# Contribution à NAUTILE : un environnement pour la compilation du silicium

Philippe Bondono

THESE

présentée par

Philippe BONDONO

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT **N**ATIONAL **P**OLYTECHNIQUE de **G**RENOBLE

*(arrêté ministériel du 23 novembre 1988)*

(**Spécialité** : *Informatique*)

==========

Contribution à NAUTILE:

# Un Environnement pour la Compilation de Silicium

==========

Date de soutenance: 8 décembre 1989

Composition du jury:

| | | |
|---|---|---|
| *J.* | *Mossière* | *Président* |
| *H.D.* | *Bonifas* | |
| *B.* | *Courtois* | |
| *A.* | *Greiner* | *Rapporteur* |
| *A.A.* | *Jerraya* | |
| *C.* | *Masson* | *Rapporteur* |

Thèse préparée au sein du laboratoire TIM3

THESE

présentée par

Philippe BONDONO

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT **N**ATIONAL **P**OLYTECHNIQUE de **G**RENOBLE

*(arrêté ministériel du 23 novembre 1988)*

(**Spécialité** : *Informatique*)

==========

Contribution à NAUTILE:

# Un Environnement pour la Compilation de Silicium

==========

Date de soutenance: 8 décembre 1989

Composition du jury:

| | | |
|---|---|---|
| *J.* | *Mossière* | *Président* |
| *H.D.* | *Bonifas* | |
| *B.* | *Courtois* | |
| *A.* | *Greiner* | *Rapporteur* |
| *A.A.* | *Jerraya* | |
| *C.* | *Masson* | *Rapporteur* |

Thèse préparée au sein du laboratoire TIM3

# INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD

Année 1988

## Professeurs des Universités

| | | |
|---|---|---|
| BARIBAUD | Michel | ENSERG |
| BARRAUD | Alain | ENSIEG |
| BAUDELET | Bernard | ENSPG |
| BEAUFILS | Jean-Pierre | ENSEEG |
| BLIMAN | Samuel | ENSERG |
| BLOCH | Daniel | ENSPG |
| BOIS | Philippe | ENSHMG |
| BONNETAIN | Lucien | ENSEEG |
| BOUVARD | Maurice | ENSHMG |
| BRISSONNEAU | Pierre | ENSIEG |
| BRUNET | Yves | IUFA |
| CAILLERIE | Denis | ENSHMG |
| CAVAIGNAC | Jean-François | ENSPG |
| CHARTIER | Germain | ENSPG |
| CHENEVIER | Pierre | ENSERG |
| CHERADAME | Herve | UFR PGP |
| CHOVET | Alain | ENSERG |
| COHEN | Joseph | ENSERG |
| COUMES | André | ENSERG |
| DARVE | Félix | ENSHMG |
| DELLA-DORA | Jean-François | ENSIMAG |
| DEPORTES | Jacques | ENSPG |
| DESRE | Pierre | ENSEEG |
| DOLMAZON | Jean-Marc | ENSERG |
| DURAND | Francis | ENSEEG |
| DURAND | Jean-Louis | ENSIEG |
| FOGGIA | Albert | ENSIEG |
| FONLUPT | Jean | ENSIMAG |
| FOULARD | Claude | ENSIEG |
| GANDINI | Alessandro | UFR PGP |
| GAUBERT | Claude | ENSPG |
| GENTIL | Pierre | ENSERG |
| GREVEN | Hélène | IUFA |
| GUERIN | Bernard | ENSERG |
| GUYOT | Pierre | ENSEEG |
| IVANES | Marcel | ENSIEG |
| JAUSSAUD | Pierre | ENSIEG |
| JOUBERT | Jean-Claude | ENSPG |
| JOURDAIN | Geneviève | ENSIEG |
| LACOUME | Jean-Louis | ENSIEG |
| LESIEUR | Marcel | ENSHMG |
| LESPINARD | Georges | ENSHMG |
| LONGEQUEUE | Jean-Pierre | ENSPG |
| LOUCHET | François | ENSIEG |
| MASSE | Philippe | ENSIEG |
| MASSELOT | Christian | ENSIEG |
| MAZARE | Guy | ENSIMAG |
| MOREAU | René | ENSHMG |
| MORET | Roger | ENSIEG |
| MOSSIERE | Jacques | ENSIMAG |
| OBLED | Charles | ENSHMG |
| OZIL | Patrick | ENSEEG |
| PARIAUD | Jean-Charles | ENSEEG |
| PERRET | René | ENSIEG |
| PERRET | Robert | ENSIEG |
| PIAU | Jean-Michel | ENSHMG |
| POUPOT | Christian | ENSERG |
| RAMEAU | Jean-Jacques | ENSEEG |
| RENAUD | Maurice | UFR PGP |
| ROBERT | André | UFR PGP |
| ROBERT | François | ENSIMAG |
| SABONNADIERE | Jean-Claude | ENSIEG |
| SAUCIER | Gabrielle | ENSIMAG |
| SCHLENKER | Claire | ENSPG |
| SCHLENKER | Michel | ENSPG |
| SERMET | Pierre | ENSERG |
| SILVY | Jacques | UFR PGP |
| SIRYES | Pierre | ENSHMG |
| SOHM | Jean-Claude | ENSEEG |
| SOLER | Jean-Louis | ENSIMAG |
| SOUQUET | Jean-Louis | ENSEEG |
| TROMPETTE | Philippe | ENSHMG |
| VEILLON | Gérard | ENSIMAG |
| ZADWORNY | François | ENSERG |

## Personnes ayant obtenu le diplôme

## D'HABILITATION A DIRIGER DES RECHERCHES

| | |
|---|---|
| BECKER | Monique |
| BINDER | Zdeneck |
| CHASSERY | Jean-Marc |
| CHOLLET | Jean-Pierre |
| COEY | John |
| COLINET | Catherine |
| COMMAULT | Christian |
| CORNUEJOLS | Gérard |
| COULOMB | Jean-Louis |
| DALARD | Francis |
| DANES | Florin |
| DEROO | Daniel |
| DIARD | Jean-Paul |
| DION | Jean-Michel |
| DUGARD | Luc |
| DURAND | Madeleine |
| DURAND | Robert |
| GALERIE | Alain |
| GAUTHIER | Jean-Paul |
| GENTIL | Sylviane |
| GHIBAUDO | Gérard |
| HAMAR | Sylvaine |
| HAMAR | Roger |
| LADET | Pierre |
| LATOMBE | Claudine |
| LE CORREC | Bernard |
| MADAR | Roland |
| MULLER | Jean |
| NGUYEN TRONG | Bernadette |
| PASTUREL | Alain |
| PLA | Fernand |
| ROUGER | Jean |
| TCHUENTE | Maurice |
| VINCENT | Henri |

## CHERCHEURS DU C.N.R.S

Directeurs de recherche 1ère Classe

| | | | |
|---|---|---|---|
| CARRE | René | LANDAU | Ioan |
| FRUCHART | Robert | VACHAUD | Georges |
| HOPFINGER | Emile | VERJUS | Jean-Pierre |
| JORRAND | Philippe | | |

Directeurs de recherche 2ème Classe

| | | | |
|---|---|---|---|
| ALEMANY | Antoine | KLEITZ | Michel |
| ALLIBERT | Colette | KOFMAN | Walter |
| ALLIBERT | Michel | KAMARINOS | Georges |
| ANSARA | Ibrahim | LEJEUNE | Gérard |
| ARMAND | Michel | LE PROVOST | Christian |
| BERNARD | Claude | MADAR | Roland |
| BINDER | Gilbert | MERMET | Jean |
| BONNET | Roland | MICHEL | Jean-Marie |
| BORNARD | Guy | MUNIER | Jacques |
| CAILLET | Marcel | PIAU | Monique |
| CALMET | Jacques | SENATEUR | Jean-Pierre |
| COURTOIS | Bernard | SIFAKIS | Joseph |
| DAVID | René | SIMON | Jean-Paul |
| DRIOLE | Jean | SUERY | Michel |
| ESCUDIER | Pierre | TEODOSIU | Christian |
| EUSTATHOPOULOS | Nicolas | VAUCLIN | Michel |
| GUELIN | Pierre | WACK | Bernard |
| JOUD | Jean-Charles | | |

## Personnalités agréees à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

ENSEEG

| | | | |
|---|---|---|---|
| CHATILLON | Christian | SARRAZIN | Pierre |
| HAMMOU | Abdelkader | SIMON | Jean-Paul |
| MARTIN GARIN | Régina | | |

ENSERG

| | |
|---|---|
| BOREL | Joseph |

ENSIEG

| | | | |
|---|---|---|---|
| DESCHIZEAUX | Pierre | PERARD | Jacques |
| GLANGEAUD | François | REINISCH | Raymond |

ENSHMG

| | |
|---|---|
| ROWE | Alain |

ENSIMAG

| | |
|---|---|
| COURTIN | Jacques |

EFP

| | |
|---|---|
| CHARUEL | Robert |

C.E.N.G

| | | | |
|---|---|---|---|
| CADET | Jean | NIFENECKER | Hervé |
| COEURE | Philippe | PERROUD | Paul |
| DELHAYE | Jean-Marc | PEUZIN | Jean-Claude |
| DUPUY | Michel | TAIEB | Maurice |
| JOUVE | Hubert | VINCENDON | Marc |
| NICOLAU | Yvan | | |

Laboratoires extérieurs :

C.N.E.T

| | | | |
|---|---|---|---|
| DEVINE | Rodericq | MERCKEL | Gérard |
| GERBER | Roland | PAULEAU | Yves |

**********

**********

***

**Je tiens à remercier:**

Mr le Professeur **J. Mossière**, Directeur de l'E.N.S.I.M.A.G., d'avoir accepté de présider le jury de cette thèse.

Mr **A. Greiner**, Professeur à l'Université Pierre et Marie Curie pour avoir accepté d'être rapporteur de mon travail.

Mr **C. Masson**, Responsable du service de C.A.O. de V.L.S.I. de la société CII BULL, pour avoir accepté d'être rapporteur de mon travail.

Mr **D. Bonifas**, Responsable du service de C.A.O. de V.L.S.I. de la société IBM de m'avoir accueilli dans son service et de m'avoir ainsi fait découvrir les impératifs du monde industriel.

Mr **B. Courtois**, Directeur du laboratoire TIM3, qui m'a accueilli au sein de son équipe et qui a dirigé efficacement mon travail.

Mr **A. Jerraya**, Chargé de recherche au C.N.R.S. pour la patience toute orientale dont il a pu faire preuve en m'encadrant.

Ainsi que tous mes collègues et amis des laboratoires TIM3 et IBM, ceux de l'aventure informatique tunisienne, et tous les autres, que je ne pouvais décemment oublier.

Résumé

NAUTILE est un environnement de génération de dessins de masques, adapté aux besoins des compilateurs de silicium. Il permet la réalisation d'outils d'assemblage et de générateurs paramétrés par la technologie.
NAUTILE facilite la décomposition d'un circuit en une hiérarchie de cellules pouvant être formalisées suivant plusieurs niveaux d'abstraction appelés "vues". Une cellule possède deux types de vues, les vues physiques et une vue de construction. Celle-ci permet de concevoir des circuits "corrects par construction", et met en œuvre des règles de composition paramétrées par la technologie.
Enfin, NAUTILE permet l'intégration d'outils et de cellules développés dans d'autres systèmes.

Abstract

NAUTILE is an environment to help designing physical layout in silicon compilers. It is aimed to produce assembly tools and technology independant module generators.
In NAUTILE a circuit is described with a hierarchy of cells, each cell may be defined within different abstraction levels, called "views". A cell has two kind of views: physical views, and a construction view. The construction view allows to design "correct by construction" circuits, by using technology dependant composition rules.
Finally, NAUTILE acts as a common repository for both cells and tools developped externally.

# 1. INTRODUCTION

## 1.1. Le point sur la conception de circuits aujourd'hui

Dans les premiers temps de l'informatique, les programmes étaient écrits directement en langage machine. Puis, la complexité des programmes croissant avec celle des machines les traitant, on vit apparaitre des langages de plus en plus sophistiqués, permettant une totale abstraction vis à vis de la machine hôte: les concepteurs de logiciels préfèrent perdre en rapidité d'exécution et en occupation de la mémoire, afin de gagner en efficacité et en sureté. Seules certaines parties trés critiques se verront encore programmer en langage assembleur, relativement proche du langage machine.
En conception de circuits, une telle démarche peut être observée, même si, plus récente, elle n'est encore pas communément admise.
Ainsi, depuis le début des circuits VLSI, la complexité de ceux-ci n'ayant cessé de croitre, le concepteur nécessite des outils permettant l'automatisation des taches répétitives et fastidieuses. Des outils de plus en plus sophistiqués voient le jour régulièrement, adressant essentiellement une automatisation partielle ou totale aussi bien au niveau de la synthèse (éditeurs, compacteurs, routeurs), que de l'analyse (simulateurs, vérificateurs des règles).
Ces outils permettent au concepteur de se détacher de l'aspect détaillé de la technologie, pour concevoir des systèmes à un niveau plus élevé: même si l'automatisation des tâches de conception se fait généralement au détriment de la qualité du produit obtenu, (notamment au niveau de l'optimisation de sa surface et de la rapidité), beaucoup considèrent que c'est le prix à payer pour une conception rapide (les impératifs économiques l'exigent) et sure (les circuits doivent être corrects du premier coup).

## 1.2. Pourquoi un environnement de conception

Face à cette demande croissante d'automatisation, on assiste à l'éclosion d'une multitude d'outils, aussi bien universitaires qu'industriels, chacun d'entre eux possédant ses propres formats d'entrée/sortie, la plupart du temps incompatibles avec ceux des autres outils.

Afin de permettre l'échange d'informations entre ces outils, diverses solutions peuvent être retenues, notamment l'emploi d' un langage intermédiaire servant d'interface commune aux différents outils, l'autre solution communément adoptée consistant à faire dialoguer les outils par l'intermédiaire d'une base de données commune.

Le système NAUTILE [Hornik 89] est un environnement de travail permettant l'intégration d'outils trés variés, combinant les avantages des deux solutions présentées ci-dessus.

## 1.3. Les objectifs du système Nautile

L'équipe d'architecture des ordinateurs a commencé à développer des systèmes de conception de circuits à la fin des années 70. A cette époque, un éditeur graphique de dessin des masques a vu le jour, le système LUCIE (Langage Universitaire de Conception de Circuits Intégrés pour l'Enseignement) [Paillotin 85].

Par la suite, il a été nécessaire d'envisager la création d'un outil plus puissant, tout en restant simple, et répondant aux besoins des concepteurs de circuits intégrés. L'idée maitresse de ce système était une description algorithmique des circuits intégrés (en langage Pascal). Dans ce système, les circuits sont décomposés hiérarchiquement de façon arborescente, les feuilles de l'arbre ainsi constitué étant des petites figures dessinées à la main, et prévues pour s'assembler entre elles. LUBRICK (assembleur de BRIques LUcie) [Schoellkopf 83] était né.

Forte de cette expérience, l'équipe TIM3 s'est associée au projet SYCOMORE, qui avait pour but les spécifications d'un environnement de conception de circuits intégrés utilisable à l'échelle nationale. Plusieurs partenaires ont participé à ce projet, TIM3 s'étant engagé à fournir un compilateur de silicium (SYCO [Jerraya 86]), ainsi qu'à participer à la définition d'un système de conception hiérarchique symbolique de dessin des masques (STYX [Rougeaux 87]).

L'expérience acquise avec STYX permit, en septembre 1986, de démarrer un nouveau projet devant en principe résoudre les problèmes non adressés par STYX. Ce projet, appelé NAUTILE (New AUtomatic and Technology Independant Layout Environment) devait notamment

permettre l'intégration au sein d'un même environnement des outils provenant de mondes trés différents, tels que IBM (GL1 [Lambert 81] et ASTAP [Astap 84]), le laboratoire TIM3 à Grenoble (LUBRICK et LUCIE), l'université de Washington (RNL [Notrott 87][Baker 80]), ainsi que d'autres. On peut citer quelques unes des idées directrices ayant présidé à la réalisation du système:

- l'utilisation de cellules déja existantes, décrites dans divers systèmes
- l'intégration d'outils utilisés par d'autres systèmes
- la possibilité de changements de la technologie à moindre frais
- une description des circuits suivant différents niveaux d'abstraction
- un traitement unifié de ces différents niveaux d'abstraction
- l'assurance que le dessin obtenu est correct.

La finalité d'un tel produit devant essentiellement être la construction de générateurs de cellules, employés dans le compilateur de silicium comportemental SYCO.

L'approche considérée a donc consisté à caractériser un langage décrivant de façon hiérarchique la structure des circuits et leur topologie, dans un environnement aisément modifiable et extensible, permettant un dessin dit "correct par construction".

## 1.4. Plan de la thèse

La première partie sera consacrée à une présentation des différents travaux dont nous nous sommes inspirés, ou qui étaient suffisamment proches des concepts que nous avons développés pour que nous ne puissions les passer sous silence.
Une deuxième partie présentera le système NAUTILE dans son ensemble, ses particularités étant développées dans une troisième partie.
Puis nous détaillerons la version actuelle du système ainsi que l'utilisation qui a pu en être faite, avant de clore avec quelques remarques de conclusion.

# 2. ETAT DE L'ART

Nous allons tout d'abord procéder à un tour d'horizon des différents outils permettant de concevoir des circuits intégrés, dans un ordre plus ou moins lié à leur apparition chronologique, avant de détailler les points nous étant apparus comme cruciaux dans un environnement de conception.

## 2.1. Les éditeurs évolués

Ils permettent la création et l'édition de cellules, de façon interactive à l'écran.
Dans ces systèmes, tels que SCAPP de Hewlett Packard, l'utilisateur peut définir des rectangles, des lignes, des polygones qui représentent les différentes parties du circuit. Il peut ensuite modifier son dessin en déplaçant, dupliquant, renommant, déformant certains de ces éléments, qui peuvent éventuellement être réutilisés, soit tels quels, soit aprés transformation, dans une autre partie du circuit, ou même dans un autre contexte.

Les ancien systèmes de conception n'offraient à l'utilisateur qu'une marge de manoeuvres réduite, mais de nombreuses améliorations ont progressivement vu le jour. Ainsi CAESAR [Ousterhout 81]) autorisait-il l'utilisation de cellules hiérarchiques, ainsi que l'union en un seul polygone de deux rectangles du même niveau.
Actuellement, les constructeurs de tels produits, s'attachent à présenter des éditeurs graphiques proposant des fonctionnalités plus évoluées, telles que vérification interactive des règles de dessin, routage automatisé entre plusieurs connecteurs, compaction du dessin obtenu. On trouve de telles fonctionnalités dans des systèmes tels que MAGIC (le successeur de CAESAR) [Ousterhout 84] ou des produits industriels tels que CAECO de la société SCS.

## 2.2. Les systèmes de conception symbolique

Leur but est de libérer le concepteur des contraintes d'une représentation graphique détaillée, en permettant d'une part d'introduire des changements non linéaires, et d'autre part d'être indépendant de la

technologie utilisée. A cet effet, on utilise un ensemble de symboles permettant de représenter tous les éléments simples du circuit, par exemple des portes logiques, des transistors, des connexions entre différents éléments, ou même des ensembles de symboles. Le concepteur ne pense alors plus en terme de distances fixes, mais plutôt en terme de relations entre les différents éléments. Le circuit peut alors être représenté sous forme de graphe, dont les nœuds sont les différents symboles et les arêtes les connexions électriques reliant ces symboles.

En général, les systèmes de représentation symbolique sont classifiés selon le type de la grille sur laquelle sont répartis les symboles. Trois types de grilles sont fréquemment utilisés:

- les grilles fixes (aussi appelées "grilles métriques" [Rougeaux 87])
- les grilles relatives
- les grilles virtuelles

Nous allons maintenant maintenant examiner ces différents types de grille plus en détail.

### 2.2.1. Les grilles fixes

Dans ce cas, on dessine les symboles sur une grille de pas constant, comme dans les systèmes SLIC [Gibson 76], SIDS [Clary 80], MASKS [Larsen 78], etc.... Une cellule sera représentée telle qu'elle sera à partir de masques abstraits, mais dessinés en dimension et en taille réelle. Le concepteur peut travailler à un niveau de raffinement trés grand, et ainsi gagner en densité d'intégration.
La principale limitation d'une telle représentation, est qu'on ne peut choisir comme espacement minimum entre deux éléments que le pire des cas: on ne peut pas adapter cet espacement en fonction du type des différents composants.
On peut tout de même améliorer ce procédé en utilisant comme pas de grille le plus grand commun diviseur (PGCD) des différents espacements: c'est le principe des grilles fixes fines. L'inconvénient d'une telle approche réside dans la complexité des symboles: leur stockage pose notamment des problèmes de capacité de la mémoire.
La figure 2.1 décrit le principe d'utilisation des grilles fixes:

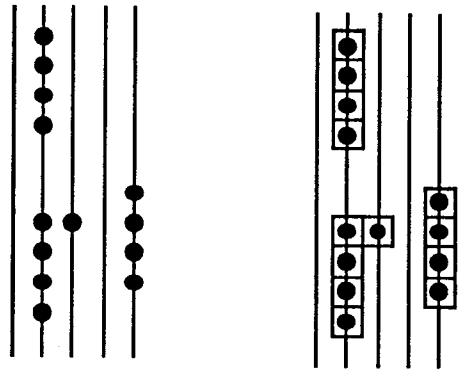


Fig. 2.1: Principe des grilles fixes

Le pas de grille correspond au pire cas d'espacement entre deux éléments. Une telle approche est forcément limitée, et ne laisse au concepteur qu'une marge de manoeuvre trés réduite: ainsi en cas de changement majeur de technologie, la plupart des cellules devront-elles être redessinées.

### 2.2.2. Les grilles relatives

Ces systèmes sont le contre-pied des précédents: ici le but est de concevoir les circuits le plus rapidement possible en libérant le concepteur des contraintes de placement fin des objets. Il se contente d'indiquer des positions relatives ainsi que le type de connexions qu'il souhaite réaliser. Une fois les cellules définies, il utilise des algorithmes de compactage pour générer le dessin des masques, et ce sont ces algorithmes qui se chargent de régler les problèmes de placement optimal, tout en vérifiant les règles de dessin.

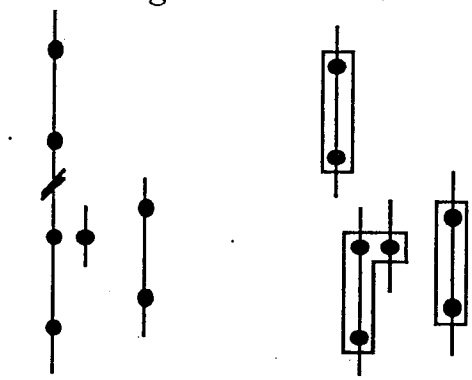


Fig. 2.2: Principe des grilles relatives

On voit sur la figure 2.2 que les éléments dessinés ne sont pas assujettis à un emplacement fixé: ils peuvent bouger les uns par rapport aux autres.

Ce principe peut encore être amélioré, par exemple en considérant que l'emplacement des connecteurs peut évoluer à l'intérieur d'une même cellule: on définit alors les limites entre lesquelles cette position peut varier.
Des systèmes comme FLOSS [Cho 77], CABBAGE [Hsueh 79], et STICKS [Williams 77] utilisent des grilles relatives.

### 2.2.3. Les grilles virtuelles

Ce type de représentation consiste à rendre un système aussi fin que dans les grilles fixes quant à la densité de conception, et aussi souple que les grilles relatives quant aux possibilités de déformation des cellules.
L'approche adoptée par exemple dans MULGA [Weste 81] ou NS [Cherry 85] a consisté à permettre aux lignes et aux colonnes d'avoir chacune un espacement propre. La figure 2.3 illustre ce principe:

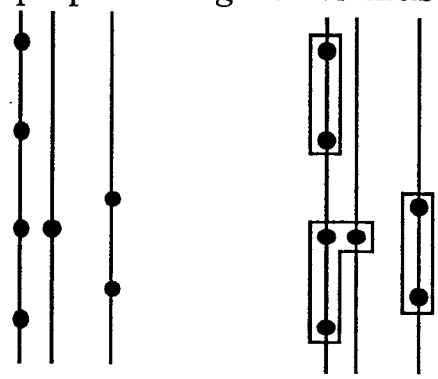


Fig. 2.3: Principe des grilles virtuelles

Dans ce type de grille, l'espace entre deux mailles peut varier suivant le contexte, de façon non uniforme. Le pas de grille est calculé en fonctions des éléments placés et de la technologie. L'espace final est calculé lors de la réalisation du dessin des masques, également à l'aide d'algorithmes de compaction.
Le problème avec les grilles virtuelles, c'est que les différents éléments d'une même colonne vont tous être espacés de leurs voisins de la même valeur, alors qu'il est parfois préférable de considérer tous les éléments comme des cas particuliers.

### 2.2.4. Utilisation des grilles virtuelles et relatives

L'utilisation de ces grilles impose généralement la construction d'un graphe des contraintes (c'est surtout vrai dans le cas de grilles relatives):

ce graphe représente les relations existant entre les différents éléments de la grille, et facilite le travail de compaction et de vérification.

Actuellement, on utilise plus volontiers les grilles virtuelles pour la conception de structures régulières, l'usage des grilles relatives (moins faciles d'emploi) étant plutôt réservé aux structures irrégulières. En fait tout le problème consiste à parvenir au meilleur compromis entre la souplesse d'utilisation et l'encombrement mémoire.

Une question se pose lorsqu'on utilise ces grilles, c'est la possibilité ou non qui est laissée à l'utilisateur de placer les éléments de son circuit de façon absolue. En effet, si on permet cette pratique, on peut en venir à violer des règles de dessin, le respect de ces règles étant assuré par l'emploi des grilles.
Sur ce sujet, les concepteurs de systèmes sont partagés: DPL [Batali 80] par exemple préfère laisser le choix à l'utilisateur, alors que ALI [Lipton 82] le lui interdit.

### 2.2.5. L'expérience symbolique à TIM3: STYX

Le système STYX [Jerraya 85] [Rougeaux 87] propose une approche générale orientée objet pour décrire des objets graphiques, permettant de concevoir des circuits VLSI.
Il a été réalisé dans le cadre du projet national SYCOMORE, dont le but était de réaliser un système intégré de conception de circuits, écrit dans un environnement portable Lisp (LeLisp [Chailloux 88]). Ce projet devait notamment consister à réaliser une "boite à outils", comportant:

- un langage de description,
- un simulateur multi-niveau,
- des outils de synthèse,
- un compilateur de silicium (appelé SYCO: voir paragraphe 2.8.4),
- un éditeur symbolique appelé STYX.

STYX est un éditeur symbolique dont les objets de base sont les articulations et les fils (pour les connexions), ainsi que les transistors (pour les éléments actifs). Les entités graphiques qu'il manipule sont définis à l'aide d'un langage immergé dans LeLisp (voir paragraphe 2.9.2).
Il propose un éditeur graphique traduisant les commandes graphiques sous la forme procédurale équivalente.

A l'inverse des systèmes graphiques axés vers une méthodologie de conception spécifique, STYX s'est fixé les buts suivants :

- définir un ensemble minimum de concepts graphiques génériques. A cette fin un langage d'assemblage a été construit. Ainsi concevoir un circuit à l'aide de STYX consiste principalement à écrire des procédures qui décrivent un circuit intégré.
- fournir un outil graphique pouvant s'adapter à différentes technologies, méthodologies et requêtes des utilisateurs. A cette fin il a été utilisé un formalisme orienté objet pour décrire les objets STYX et leur comportement, en particulier pour la description des objets présents en bibliothèque.
- unifier le traitement des cellules élémentaires et des cellules composées (les deux types de cellules sont appelés et utilisés par les mêmes mécanismes). Ce dernier point permet de simplifier les modèles de données et de garantir une vue environnement unifiée.
- avoir une correspondance étroite entre les objets et les actions sur les objets, et les utiliser aussi bien pour l'environnement graphique que pour l'environnement procédural.
- fournir au chargement des vérifications automatiques de la consistence des données.

Bien que proposant une méthode d'assemblage automatique trés puissante, le système STYX s'est cependant révélé limité pour le traitement de grosses quantités d'informations, le principe des articulations définies pour chaque connexion et pour chaque fil, faisant littéralement exploser la mémoire (par exemple pour un PLA on devait gérer des milliers d'articulations, de fils et de connexions).

## 2.3. Les systèmes procéduraux

En 1979 au Caltech, D. Johannsen [Johannsen 79] formulait la remarque suivante: "90% du circuit est obtenu en 10% du temps de conception, alors que les 10% restant requièrent 90% du temps de conception. La plupart des choix de dessin sont pris au départ, sans que les conséquences de ces choix ne soient encore apparentes". Partant de cette constatation, il proposait un système permettant d'obtenir le dessin des masques, de la même façon qu'un compilateur permet d'obtenir du langage machine à partir d'un code source. Le terme "compilateur de silicium" était né, consacrant l'étape la plus importante franchie à ce jour

en conception de circuits intégrés VLSI: l'introduction d'un langage comme support de la description des différentes entités manipulées.
Si l'utilisation d'une description textuelle plutôt que graphique des circuits intégrés fut initialement le fait du Caltech, c'est probablement du MIT (Massachussets Institute of Technology) que provient sa popularité, avec notamment les travaux sur DPL [Batali 80] puis ensuite Daedalus [Batali 81]. L'idée majeure de systèmes tels que DPL est d'étendre un langage de programmation classique (Lisp dans le cas de DPL) par un ensemble de primitives spécialisées dans la conception de circuits intégrés.
Le dessin des masques est produit par l'évaluation d'une procédure dans laquelle le concepteur décrit son circuit comme une hiérarchie de cellules assemblées entre elles.
D'autres systèmes ont également été construits autour de langages informatiques clasiques: C (L [Burich 87] et CHISEL [Karplus 82]), Lisp (SKILL [Law 87] et LUCIFER [Chailloux 83]), Pascal (LUBRICK [Schoellkopf 83]), etc...

### 2.3.1. Intérêts de l'approche procédurale

Le fait d'utiliser une description procédurale d'un circuit permet une conception basée sur les mêmes critères que l'algorithmique classique: structuration, hiérarchie et modularité, se traduisant par une réduction de la complexité des problèmes à traiter.
L'une des conséquences immédiates de l'utilisation d'une telle méthode est la faculté d'inclure dans la description d'un circuit un ensemble non exhaustif de paramètres: un même programme, s'il est bien écrit, pourra ainsi générer différentes variantes d'un même circuit (en adoptant par exemple comme paramètre la consommation ou la rapidité de ce circuit), ou même sa généralisation à une classe complète de circuits (par exemple en paramétrant le nombre d'entrées et de sorties de ceux-ci). On peut ainsi aisément généraliser l'expérience acquise sur de petits circuits, à de grands circuits du même type, pourvu qu'on ait pris la précaution d'obtenir des structures régulières[1] . Ceci facilite grandement les séquences de test et de simulation: il est plus facile de tester un multiplieur 4*4 puis de généraliser à un multiplieur 8*32, que de simuler directement ce dernier (le temps nécessaire à cette simulation serait d'ailleurs beaucoup trop important).

[1] En admettant que la généralisation soit possible!

Le tableau 2.1 donne les différents éléments variables pour la porte NAND proposée par la société Silicon Compilers Systems dans l'environnement GDT (d'aprés un article sur les générateurs de modules [Meyer 87]):

| Entrées | Sorties | Facteur de forme | Emplacement des connecteurs | Transistors de sortie | Divers |
|---|---|---|---|---|---|
| Au moins 2 | Fixées | Variable | Espacement: Variable<br><br>Niveau: Variable (ALU, M1, M2) | W/L variable | Transistors p et n varient indépendamment les uns des autres<br><br>Bus d'alimentation variable |

Tableau 2.1: Paramètres dans un générateur de porte NAND

En résumé on peut donner les avantages suivants d'un système procédural:

- indépendance technologique: la même procédure peut générer une cellule dans différentes technologies, suivant le fichier de règles utilisé.
- paramétrisation électrique: la possibilité de pouvoir paramétrer les dimensions des transistors permet de faire varier leurs caractéristiques électriques, et donc celles du circuit final.
- paramétrisation topologique: l'emploi des variables dans les procédures générant les cellules permet de paramétrer les différentes dimensions de la cellule à générer. D'autre part l'utilisation de structures conditionnelles permet de générer certaines portions du circuit "à la demande".
- modifications facilitées de cellules trop importantes pour tenir sans perte de résolution dans un seul écran.
- niveaux d'absraction multiples: une même procédure peut décrire les différentes représentations d'un circuit (il suffit pour cela d'adapter les cellules de base en conséquence).

### 2.3.2. La conception structurée

L'approche procédurale apporte une simplification des problèmes à résoudre, en facilitant l'adoption de ce qu'on appelle la méthode de conception structurée.
Dans cette méthode, le concepteur cherche à décomposer les objets qu'il manipule en une hiérarchie de sous-blocs appelés *cellules*, chaque cellule

représentant une unité fonctionnelle. Cette hiérarchie peut être représentée sous forme d'un graphe acyclique orienté, d'où son nom *d'arbre* .
Une fois cette décomposition achevée, il doit réaliser les blocs élémentaires constituant la base du circuit (les *feuilles* de l'arbre), avant de les assembler pour construire le circuit.
Pour chacune des étapes de nombreux outils sont utilisés, pour générer les cellules, les assembler, les vérifier, etc...
La nécessité d'une approche structurée est imposée par le principe même de la conception d'un circuit. Pour concevoir un circuit, le concepteur procédera de façon structurée, et devra se voir offrir les facilités correspondantes. Tout circuit conçu est divisé en cellules elles-mêmes divisées en sous-cellules, etc... Certaines de ces cellules sont mises en commun dans différentes parties du circuit: ainsi au lieu de décrire N fois une même cellule, on ne la décrira qu'une seule fois, et on appellera sa description N fois.

De plus la conception d'un circuit implique la manipulation de données en quantités trés importantes, aussi est-il trés pénalisant d'organiser celles-ci à plat (i.e. sans hiérarchie). Si modifier un point du circuit impose à chaque fois de recalculer en entier celui-ci, on arrivera rapidement à une situation impossible (temps de calcul, taille mémoire). Il faut donc qu'un maximum de données soient mises en commun, et que la modification d'une cellule se répercute automatiquement et rapidement dans toute la structure. Pour cela, il faut que la structure de données reflète exactement la structure du circuit, et la pensée du concepteur.

### 2.3.3. Exemple d'un système procédural: DPL

DPL (Design Procedure Language) [Batali 80] a été développé par le MIT (Massachussets Institute of Technology) comme un langage de conception procédurale pour les circuits intégrés.
DPL comprend une base de données orientée objet organisée hiérarchiquement et un ensemble de fonctions LISP pour la manipuler. Une procédure DPL construira une structure de données contenant une description du circuit à concevoir. Elle pourra elle même contenir des procédures de manipulations de la base de données.

Le processus de conception en DPL va passer par les étapes suivantes [Newton 87]:

- le concepteur spécifie les procédures décrivant les cellules de base du circuit.
- ces cellules sont assemblées entre elles.
- le résultat final est une structure organisée hiérarchiquement qui permettra d'obtenir le dessin des masques.

DPL spécifie un certain nombre de notions de base:

- le type (ou générateur), qui correspond à la description d'une classe d'objets générés par une fonction à paramètres (les objets seront les structures créées par la fonction).
- le prototype (ou définition), qui est une structure construite à partir d'un type:

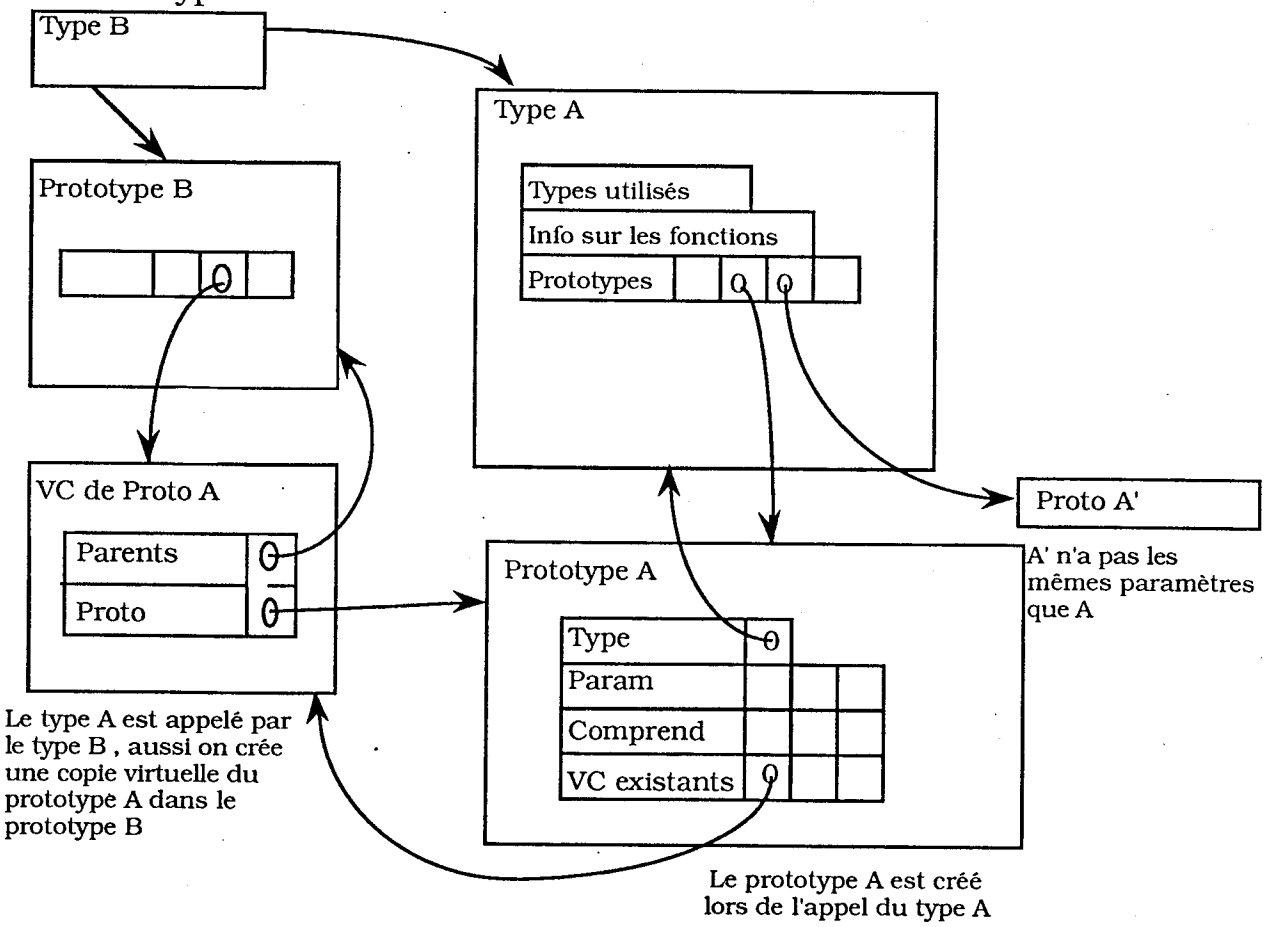



Fig. 2.4: Types et Prototypes dans DPL

- les copies virtuelles (ou instances): lors de la création d'un prototype à partir d'un type, il peut y avoir des appels à d'autres types. Dans ce cas on crée à l'intérieur du prototype une copie virtuelle (VC) du prototype résultat du type appelé. Sur la figure 2.4, par exemple, on

voit que le type B fait appel au type A (pour obtenir un prototype A), donc lors de la création du prototype B on crée une copie virtuelle du prototype A.

• les supertypes et soustypes: il est possible dans DPL de définir des "supertypes" (ou surtypes) qui contiennent d'autres types. Par exemple si on définit C comme un supertype de B, celui-ci contiendra tous les paramètres de C plus éventuellement d'autres paramètres (voir Fig 2.5).

• les instances placées qui correspondent aux différentes façons d'utiliser un prototype pour une copie virtuelle (par exemple avec une transformation géométrique). Chaque copie virtuelle contiendra une instance indiquant des données par rapport au contexte dans lequel on utilise le prototype appelé.

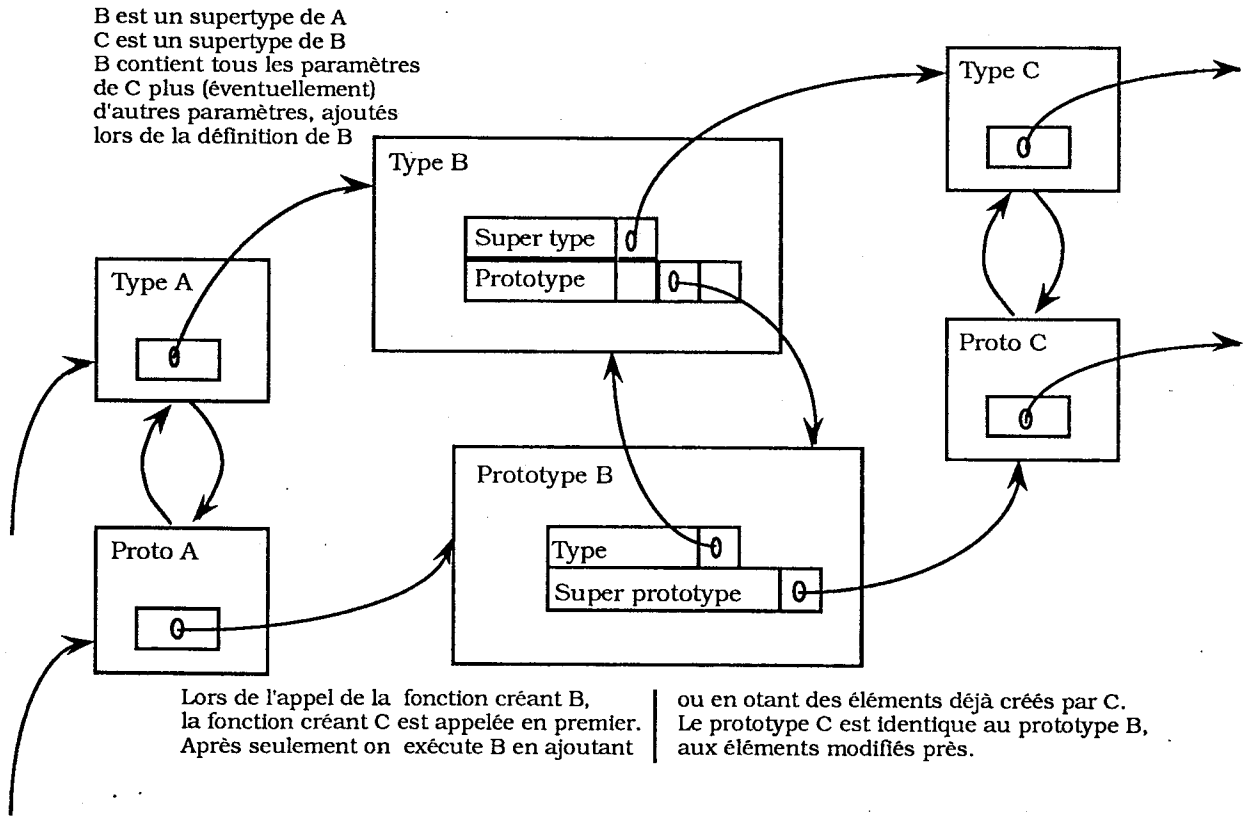



Fig. 2.5: Super-Types et Sous-Types

On peut décomposer le processus de DPL pour les appels de type de la façon suivante :

-1- évaluer les paramètres donnés

-2- résoudre les contraintes, attribuer des valeurs aux variables. Si des variables restent sans valeur attribuée, leur donner la valeur par défaut.

-3- rechercher dans les prototypes déjà existants s'il existe un prototype correspondant aux valeurs des paramètres, dans ce cas l'utiliser, sinon créer un nouveau prototype.

-4- donner une instance du prototype créé avec les transformations appropriées.

Les concepts de type, prototype, supertype, copie virtuelle et instance, relatifs au langage DPL se retrouvent aussi dans beaucoup de langages orientés objet (tel que SMALLTALK [Goldberg 83]), avec parfois des noms différents.
On verra dans le chapitre 3 la façon dont ces concepts ont été utilisés pour le système NAUTILE.

### 2.3.4. L'expérience procédurale au laboratoire TIM3: LUBRICK

Suite logique de LUCIE [Paillotin 85], un langage de description statique des masques, LUBRICK [Schoellkopf 83] est un outil d'assemblage de cellules métriques: l'assemblage est décrit par un programme Pascal, les cellules de base étant définies en LUCIE.
L'assemblage est fait sans possibilité de redimensionnement des cellules, celles-ci devant être conçues pour s'assembler par aboutement.
Les connexions sont caractérisées par des connecteurs, auxquels sont associées des directions de connexion, et qui peuvent être typés.
Les problèmes essentiels de LUBRICK proviennent de l'utilisation du langage Pascal. En effet, outre que celui-ci n'est pas interactif (ce qui implique que l'utilisateur ne peut pas manipuler directement la structure de données), il est limité par l'absence de types génériques, ainsi que de tableaux dynamiques.

### 2.3.5. Les limites de l'approche procédurale

Bien qu'elle soit séduisante à bien des points de vue, l'approche procédurale comporte un certain nombre de limitations, dont souffraient essentiellement les premiers systèmes qui l'utilisèrent.
Ainsi de nombreux systèmes ne permettent malheureusement pas de décrire de façon procédurale toutes les cellules, mais uniquement des cellules d'assemblage: seules celles-ci sont paramétrables, les cellules de base étant figées.

D'autre part, les relations entre parties graphiques et textuelles sont généralement réduites à leur plus simple expression, et peu de systèmes proposent une complète correspondance entre les deux. Dans la plupart des cas la partie graphique est le résultat de l'évaluation de la partie textuelle, plutôt qu'une entité ayant son existence propre. Les systèmes récents cherchent à pallier à cette faiblesse en proposant les deux environnements: ainsi le système MOVIE [Andersson 89] permet de formaliser graphiquement les procédures décrivant les cellules. Dans NAUTILE le problème est résolu grace à l'utilisation de la programmation orientée objet: les primitives graphiques et textuelles sont identiques, simplement il s'agit dans ce cas de méthodes dont le résultat dépendra du contexte d'appel.
Une autre limitation de l'approche procédurale est la difficulté de valider un circuit. En effet cette validation se fait généralement sur le dessin des masques, alors que plus aucune information sur la procédure ayant produit ce dessin n'est disponible. Aussi dans la plupart des systèmes récents, la vérification fait-elle partie intégrante de la conception, au lieu d'être rejetée en fin de cycle.
Une autre contrainte majeure des systèmes procéduraux réside dans l'obligation pour les concepteurs de posséder de plus grandes qualifications, notamment en programmation: privilégier une définition textuelle au détriment des habituelles descriptions graphiques peut s'avérer parfois déroutant pour quiconque de non familiarisé avec les méthodes de programmation.

## 2.4. Améliorations de la méthode procédurale

Si DPL n'a jamais été consacré par un produit commercial fini, il n'en fut pas moins à l'origine d'un certain nombre de systèmes commerciaux s'inspirant plus ou moins de l'approche procédurale. Parmi ceux-ci on peut citer NS [Cherry 85], VTI Tools de la société VLSI Technology [Trimberger 84], et surtout le langage I développé par AT&T [Johnson 82] et ses nombreux descendants: L de la société SCS (environnement GDT) [Burich 87], SKILL de la société Cadence (environnement IDS) [Law 87], SI développé dans le projet Vdd [Chu 86], IMAGES de AT&T (projet IDA) [Hill 89], etc...
Cependant, si la plupart des concepts introduits avec DPL sont encore largement utilisés par ces systèmes, de nombreuses améliorations ont vu le jour, au fur et à mesure de l'évolution des différents concepts.

Ainsi le système conçu par la société VLSI Technology permet de concevoir des circuits soit de façon graphique, soit textuellement, le concepteur décrivant ses cellules de base avant de les assembler semi-automatiquement, à l'aide des différents outils d'assemblage, routage et compaction fournis avec le système.
Ses principales limitations résident essentiellement dans l'absence d'une description électrique, ainsi que dans l'emploi du langage VIP, d'approche relativement complexe pour un utilisateur non averti.
Une autre amélioration importante de l'approche procédurale a été l'élimination des erreurs liées aux contraintes technologiques: la méthode utilisée consiste à traduire la procédure définissant le circuit dans un système symbolique, plutôt que directement en terme de masques. Un programme de compactage permettant de résoudre les différentes contraintes liées à la technologie est alors utilisé pour obtenir le dessin des masques final. C'est notamment la philosophie adoptée par le système ALI [Lipton 82].
L'Université de Washington a également publié récemment le résultat de ses recherches sur un système de gestion de vues multiples d'un circuit [Baer88], permettant la description hiérarchique d'un circuit.
Le principe énoncé ici est que la description d'un circuit en termes de niveaux de masques est trop dépendante de la technologie, et trop détaillée pour être directement exploitable. Aussi les chercheurs de cette université introduisent-ils d'autres niveaux d'abstraction pour les blocs de base de la hiérarchie: un modèle comportemental en vue de simulation fonctionnelle, un modèle "switch" (description sous forme d'un réseau de transistors) pour la simulation électrique, et un modèle schématique, nécessaire à la documentation. L'étude présentée porte donc sur le choix d'une notation, permettant de produire et de manipuler des objets traités par le système MAGIC [Ousterhout84]. Cette approche ne permet néanmoins pas directement de s'interfacer avec d'autres systèmes, et notamment de récupérer des cellules écrites dans d'autres environnements que celui propre à MAGIC.
D'autre part les simplifications extrêmes apportées à l'assemblage limitent la portée d'un outil qui par ailleurs offre une bonne modélisation du problème des vues multiples d'un circuit et des correspondances entre elles.

Nous reviendrons ultérieurement sur d'autres améliorations importantes de la méthode basées sur des techniques d'intelligence artificielle, et de programmation orientée objet (dans le paragraphe 2.7).

## 2.5. Les langages d'interface

Les outils provenant de différents contextes étant difficiles à intégrer dans un environnement donné, l'usage de langages d'interface est devenu de plus en plus courant.
Son principe, décrit dans la figure 2.6, consiste à utiliser un langage intermédiaire servant d'interface commune aux différents outils à relier:

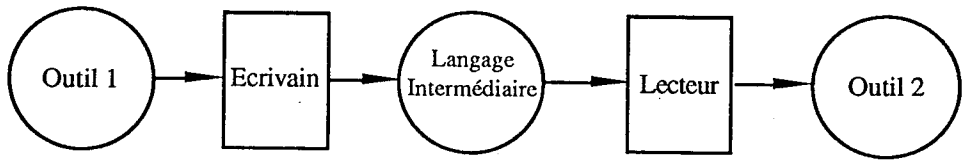



Fig. 2.6: Utilisation d'un langage d'interface

L'outil souhaitant communiquer des informations à un autre outil, utilise un programme appelé "écrivain", afin de traduire sa structure de données dans le langage choisi. Le destinataire utilise à son tour un programme appelé "lecteur", qui traduit le langage intermédiaire dans un format qui lui est directement compréhensible.
Les plus anciens de ces langages (CIF [Sproull 80], ou GDSII [Calma]) étaient limités à la description des masques, alors que les langages récents recouvrent un ensemble de descriptions beaucoup plus variés, la tendance actuelle semblant être à la standardisation: EDIF [EDIF 87] concernant les couches de description les plus basses, alors que VHDL [Lipset 89] [Vhdl 87] permet essentiellement des descriptions à un niveau assez élevé.
Le problème avec les langages d'interface, c'est que du fait de leurs limitations, les outils utilisent généralement des séquences d'échappement pour les parties critiques et optimisées du dessin. Les problèmes d'interface dans ce cas restent donc toujours non résolus.
D'autre part, hormis les séquences d'échappement qui, on l'a vu, ne sont pas trés souhaitables, il est impossible de faire évoluer les langages comme on pourrait l'espérer.
Enfin, il est vrai que si beaucoup de systèmes proposent des lecteurs EDIF par exemple, bien peu offrent en revanche des écrivains (pour des raisons commerciales évidentes), en limitant ainsi considérablement la portée.

## 2.6. Les systèmes de gestion des données

Afin de pallier aux diverses limitations d'un langage d'interface comme solution au problème de l'intégration des outils dans un environnement commun, certains auteurs préconisent d'utiliser une base de données commune à l'ensemble des outils, comme indiqué dans la figure 2.7:

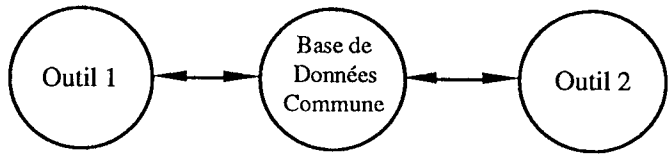



Fig. 2.7: Utilisation d'une base de données commune

Ce type de solution dans lequel les outils communiquent par l'intermédiaire de la base de données, est notamment celui proposé dans des systèmes tels que le "Version Server" [Katz 86] ou OCT [Silva 89].

### 2.6.1. L'approche initiale proposée par Katz

R.H. Katz a été l'un des premiers à se pencher sur les problèmes liés à la modélisation des données spécifiques à la conception des circuits VLSI [Katz 83][Katz 86]. En effet les approches classiques de description des données, et notamment le modèle relationnel, s'avèrent assez mal adaptées à la conception de circuits: les données à manipuler sont ici caractérisées par une structure trés complexe (en comparaison des gestionnaires de données classiques qui manipulent des grosses quantités de données trés régulières) et par l'utilisation massive de descriptions hiérarchiques, que les bases de données usuelles ne sont pas à même de traiter convenablement.
D'aprés Katz, les caractéristiques essentielles d'un système de gestion des données dans un environnement de CAO de VLSI sont:

- construction hiérarchique
- description des objets suivant plusieurs "perspectives" ou "vues"
- maintien d'un historique de la conception et de son évolution

Nous allons maintenant détailler chacun de ces concepts, dont s'inspirent la majorité des systèmes actuels de conception.

La construction hiérarchique:

Les différents objets contenus dans la base de données peuvent être décomposés en d'autres objets, qui seront à leur tour décomposés, et ainsi de suite, récursivement: on décrit ainsi une hiérarchie d'objets, représentée par un graphe acyclique orienté.
Katz distingue donc deux types d'objets: les objets de composition, composés d'autres objets, et les objets primitifs, qui sont terminaux (c'est-à-dire qu'ils ne sont composés d'aucun autre objet).
La décomposition hiérarchique des circuits permet certes une meilleure compréhension du problème à résoudre, mais elle augmente en revanche la complexité de la structure de données, et dans certains cas rend la tache des outils plus ardue (notamment la vérification des règles de dessin et la compaction).

Différentes vues pour une même objet

Katz introduit la notion d'objets "représentation", ceux-ci permettant de décrire une portion de circuit selon un mode de représentation particulier: topologique, symbolique, logique, etc...
Chaque objet de type représentation possède une partie interface, qui permet au système et aux outils de connaitre le comportement de l'objet ainsi que ses performances, sans avoir à examiner l'objet lui-même.
Les différentes représentations d'un objet sont regroupées par un objet "equivalence", permettant d'assurer la corrélation entre ces représentations: on spécifie explicitement dans la structure de données que deux vues sont équivalentes si elles décrivent effectivement une seule et même entité. Les objets "equivalence" doivent être validés avant que la base de données puisse être déclarée cohérente.

Gestion des alternatives et des versions

Comparé aux gestionnaires classiques, un système de gestion des données orienté conception de circuits, se doit de proposer un mécanisme de rattrapage d'erreurs et de maintien d'un historique. En effet, si dans un système usuel une fiche de paye est détruite en cours de

frappe, il suffit de retaper ladite feuille, et tout rentre dans l'ordre en un minimum de temps. En revanche, si un concepteur de circuits s'aperçoit que son travail de la journée est détruit à la suite d'une panne système, sans qu'il ait eu le temps de réaliser une copie de sauvegarde, les conséquences peuvent s'avérer nettement plus graves, et l'utilisateur peut être en droit de se plaindre.
Katz préconise de baser le rattrapage d'erreurs sur d'une part un historique des commandes entrées par l'utilisateur, et d'autre part la possibilité de créer différentes versions d'un même objet: le concepteur peut ainsi opérer un retour sur une version antérieure à celle sur laquelle il travaille, s'il se rend compte qu'il a fait fausse route.
Enfin la notion "d'alternative" complète celle de version: des alternatives sont différentes versions valides d'un objet. Elles permettent notamment d'analyser le comportement du circuit final, selon les différents choix de conception envisagés.

Relation entre ces différentes notions

Toutes ces notions sont étroitement liées: ainsi l'objet "générique" circuit, peut exister en différentes alternatives et versions, chacune d'elle pouvant être vue suivant différentes représentations decomposées en hiérarchies d'objets du même type que la représentation à laquelle il appartient:

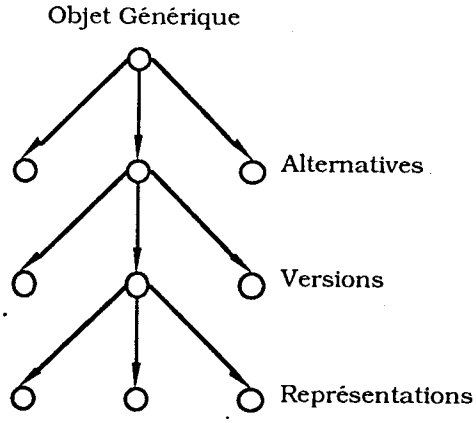



Fig. 2.8: Structure d'un objet générique chez Katz

Les problèmes essentiels de cette approche résident dans la difficulté d'assurer la cohérence entre les différents objets: cohérence entre les alternatives, entre les versions, et entre les représentations. En effet, lorsqu'on met à jour un objet, il faut s'assurer que cela ne risque pas de

mettre en péril l'édifice: une modification du schéma électrique de la cellule a de fortes chances de se répercuter sur sa topologie.
Cependant, s'il est relativement aisé de prouver que deux alternatives ou deux versions d'un même objet sont équivalentes (par exemple en n'autorisant que des opérations continues conservant l'équivalence), il est en revanche beaucoup plus ardu d'assurer l'équivalence entre deux représentations d'un même objet: celle-ci sera généralement vérifiée à la demande, à l'aide par exemple d'un extracteur.
Pour Katz le problème de la cohérence entre les différentes vues d'un même objet reste donc ouvert. Nous verrons par la suite quelle solution NAUTILE propose pour le résoudre.

### 2.6.2. Une vision parallèle: OCT

OCT [Moore 86], en conjonction avec son interface graphique VEM [Harrison 86] et le compacteur SPARCS [Burns 86], propose une base de données commune à l'ensemble des outils originaires de l'université de Berkeley.

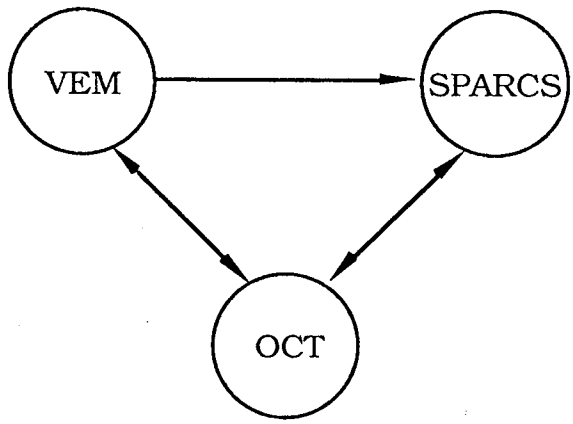



Fig. 2.9: Présentation de OCT/SPARCS/VEM [Newton 87]

Successeur de SQUID, une des premières tentatives d'intégration autour d'une base de données commune menée à Berkeley [Keller 82], OCT en a repris les principes essentiels, en les améliorant:

- le gestionnaire de données est commun à tous les outils
- les "cellules" peuvent être décrites selon plusieurs "vues"
- des facilités d'intégration d'outils futurs sont proposées
- l'accés au gestionnaire de données est simplifié et uniformisé

L'objet essentiel dans OCT est la "cellule", qui contient des "instances" d'autres cellules, et peut être définie suivant plusieurs "vues". OCT n'émet

aucune hypothèse quant au nombre et au type des vues disponibles, mais laisse ce choix à l'entière discrétion des concepteurs d'outils. Notamment des mécanismes d'intégration de nouveaux outils sont proposés.
Chaque vue peut se décomposer hiérarchiquement, et contient différentes abstractions permettant à un outil de ne pas parcourir toute la hiérarchie afin d'obtenir des renseignements sur la vue en question. Ces abstractions sont appelées des "formes de protection" ("protection frame" en anglais), les formes de protection propres à une vue étant appelées les "facettes" de la vue.
Afin de permettre des vérifications sur l'intégrité des données, OCT propose d'une part un mécanisme basé sur la chronologie des modifications (un objet en utilisant un autre doit être postérieur à celui-ci), et d'autre part maintien une liste des changements apportés à un objet: ceci permet ultérieurement qu'un outil puisse répercuter les changements opérés par d'autres outils depuis la création de cette liste.
La principale limite de OCT provient précisément de cette façon de gérer les changements: si elle est incontestablement trés souple, elle augmente en revanche la complexité de la vérification de la cohérence. En effet, celle-ci ne sera opérée que sur la demande expresse d'un outil, et non pas interactivement (comme dans NAUTILE ou MOVIE [Andersson 89]).

Récemment, les deux approches proposées par Berkeley, soit le "Version Server" de Katz, et OCT de Newton, ont été réunies dans un seul projet: le projet Octane [Silva 89], introduisant dans OCT les notions de version et de protection des données dans un environnement partagé.

### 2.6.3. Une réalisation française: COSMIC

En France, le CNET utilise actuellement des outils basés sur de tels concepts: il s'agit essentiellement du système LOF, et de la base de données COSMIC.
LOF [Berge86] est un langage de formalisation de générateurs de cellules, décrivant essentiellement des vues topologiques, même si d'autres types de vues peuvent exister (vue électrique notamment). Les deux principales entités considérées par LOF sont d'une part les "cellules de base", figées, et d'autre part les "blocs", composés de cellules de base. Ce langage, pour les besoins internes du CNET, peut également s'interfacer avec des systèmes extérieurs.

Quant à COSMIC [Jullien86], il s'agit d'une base de données orientée vers la conception de circuits intégrés . Cette base de données, accédée via l'interface DBI, sert de base commune à un ensemble d'outils développés au CNET, dont LOF.
Ces principales caractéristiques sont:

- flexibilité: il est relativement aisé de rajouter de nouveaux outils
- portabilité
- interactivité
- implantation modulaire

C'est un système complet de gestion de base de données, avec choix du support physique, rattrapage d'erreur, historique de conception, protection multi-utilisateur, etc ...

L'entité de base de COSMIC est la cellule, pouvant posséder différentes représentations, de type soit logique, soit physique. Chacun de ces types se décompose en fait en sous-types: ainsi il peut y avoir plusieurs représentations logiques différentes telles que fonctionnelle, symbolique, description en réseaux de portes, etc...
Un mécanisme de versions permet de dénoter l'état d'une représentation à un moment donné.
Les versions comportent des points de connexion externes (les "ports"), et internes (les "pins"), des composants (i.e. des instances d'autres versions), et des nœuds reliant les différents points de connexion.
Les versions et les composants possèdent de plus une vue spéciale réunissant une boite enveloppante, des commentaires, et éventuellement des éléments physiques (rectangles, polygones, etc...).
Un mécanisme de protection des données fortement inspiré des travaux de Katz est également disponible.

Ce système ne résoud malheureusement pas vraiment le problème de la gestion de la cohérence entre les différentes représentations, et d'autre part il n'offre aucun mécanisme général d'ajout d'une nouvelle vue (en revanche il fournit avec la notion "d'extent" des facilités d'intégration de nouveaux outils).

## 2.7. Utilisation de l'intelligence artificielle pour la conception

### 2.7.1. PALLADIO: un précurseur

La gestion du processus de conception par des méthodes liées à l'intelligence artificielle semble promise à un brillant avenir.
PALLADIO [Brown 83] fait figure de précurseur dans la formalisation de tels concepts, en proposant une méthode de conception basée sur l'utilisation d'une base de connaissances et de systèmes experts.
Dans l'approche proposée par PALLADIO, la conception se fait par affinage en parallèle de différentes perspectives.
Ces perspectives sont des descriptions à un niveau d'abstraction donné, en termes de composants de base et de règles régissant les interactions entre ces composants. Deux types de perspectives jouent un role prépondérant dans PALLADIO: il s'agit des perspectives de structure et de comportement.
Les perspectives de structure permettent de décrire les inter-connexions logiques entre les différents objets leur appartenant. Une perspective de structure pourrait par exemple définir le circuit au niveau transistor, alors qu'une autre le représenterait sous forme d'un ensemble de registres de transfert.
Les perspectives comportementales sont utilisées pour spécifier les variations de l'état du circuit au cours du temps. Un exemple de perspective comportementale pourrait être la description du circuit en fonction de valeurs logiques appartenant à l'ensemble {0, 1, $\phi$}. Elles sont exprimées par un ensemble de règles ayant trait au comportement des signaux.
Les perspectives de structure et de comportement peuvent être reliées de deux façons différentes, dans PALLADIO:

- soit des procédures sont utilisées pour générer une perspective de structure à partir d'une perspective de comportement (par exemple dans le cas d'un générateur de PLA)
- soit on utilise un simulateur permettant de comparer une perspective de structure avec le comportement qui en est attendu.

PALLADIO a fortement usé des techniques de programmation et de bases de données orientées objet, notamment avec l'emploi du langage LOOPS, dans lequel est écrit le système.

Si les principes introduits par PALADIO sont trés intéressants, il n'en demeure pas moins que celui-ci comporte quelques limitations, telles que la difficulté d'étendre le système, ou d'utiliser des cellules définies dans d'autres systèmes.

### 2.7.2. Des successeurs de PALLADIO: ADAM et ULYSSE

La philosophie de Palladio a notamment été reprise dans le projet ADAM mené à l'Université Sud Californienne [Knapp85] [Jain 89].
L'approche développée dans cette université est une approche de synthèse par système expert, dans laquelle les données sont communes à tous les outils. Le circuit peut être vu selon un certain nombre de vues différentes, chaque vue possédant une hiérarchie qui lui est propre, sans qu'il y ait isomorphisme entre les différentes vues. Ce concept est en soi trés puissant, mais il ne permet pas de vérification simple et peu coûteuse des relations entre les différentes vues: en fait il impose des limitations trés sévères sur les vérifications qu'il propose. D'autre part, ce système met surtout l'accent sur les problèmes rencontrés par le concepteur à un haut niveau de description (soit comportementale, soit structurelle), l'aspect physique (i.e. les spécifications électriques et topologiques) étant laissé de côté.

D'autres systèmes ont également abordé de tels concepts, notamment le système ULYSSE, décrit dans [Bushnell86] et [Bushnell 89].
Ce système peut travailler suivant plusieurs niveaux d'abstraction, y compris un niveau comportemental. Il fournit un formalisme pour décrire les interfaces avec les outils (fonctions, entrées, sorties) basé sur le principe du tableau noir ("blackboard" en anglais): les données nécessaires à un outil quelconque sont placées dans le tableau noir, jusqu'à ce que toutes les données requises soient présentes; le mécanisme d'activation de l'outil va alors pouvoir se déclencher, le résultat de l'opération étant écrit à son tour dans le tableau noir, auquel on enlève les données initiales.
Bien qu'un système tel qu'ULYSSE (ou son successeur CADWELL [Daniell 89]) ne repose que sur des représentations architecturale et structurelle, il constitue néanmoins une bonne approche pour la définition d'un "environnement pour la compilation de silicium" [Jerraya 89].

## 2.8. Les générateurs de modules

Ils représentent une catégorie d'outils de conception de circuits permettant de construire des parties de circuit définies de façon structurelle (comme des multiplieurs, des registres à décalage, des PLA, etc...), et peuvent, soit faire partie intégrante d'un assembleur, soit être utilisés seuls.

Paramétrisables fonctionnellement (nombre de bits d'entrée, chaine de bits d'une ROM, etc...), ils doivent également se montrer flexibles géométriquement, pour que les modules générés puissent s'intégrer avec les autres parties du circuit. Généralement ils proviennent du fait que la conception de circuits intégrés utilise fréquemment des cellules à structures régulières correspondant à des fonctions de logique combinatoire ou séquentielle, ou encore à des mémoires.

La première tache de l'utilisateur souhaitant les utiliser, est donc de partitionner le système à dessiner en plusieurs sous-unités ou "modules" que le programme d'assemblage devra traiter. Chaque sous-unité correspondant à un générateur de modules (chacun spécialisé pour un type de bloc donné), il est donc nécessaire de prendre en compte les différents générateurs existants lors du découpage du circuit.
L'assemblage se fait alors soit à la main (si on ne souhaite pas utiliser l'assembleur), soit en utilisant un programme de placement/routage.
Il reste alors à optimiser la surface (éventuellement à l'aide d'un programme de compaction), et à vérifier par simulation que le circuit obtenu est correct, ces deux opérations pouvant aussi être effectuées de façon procédurale.

### 2.8.1. Les différentes philosophies

Plusieurs approches sont possibles dans l'organisation d'un système proposant la génération de modules: soit il offre des générateurs tout faits, soit il donne à l'utilisateur la possibilité d'en faire d'autres, le plus facilement possible.

• dans le premier cas, le concepteur du système de CAO propose un ensemble trés varié de générateurs de modules, qu'il essaie de faire les plus généraux possibles, afin de mieux "coller" au problème de l'utilisateur

de son système (cas de GENESIL [Cheng 88] et de CONCORDE [Corbin 88]).

• dans le second cas, il estime que l'utilisateur peut lui-même construire ses propres générateurs de modules et il lui fournit les outils nécessaires à une telle réalisation (c'est le cas de systèmes tels que GDT [Burich 87] et IDS [Law 87]). En effet, partant de son expérience personnelle de la conception de circuits intégrés, il est relativement simple pour un concepteur de généraliser la description d'une cellule, au développement d'un générateur de module pour cette même cellule.
Cette méthode offre bien sur une grande souplesse en permettant à l'utilisateur de définir exactement le type de modules dont il a besoin.
Pourtant, pour enrichissante qu'elle soit, cette approche n'en reste pas moins assez limitée. En effet, elle réclame une plus grande qualification de la part des utilisateurs, ainsi qu'un changement radical dans leurs habitudes, ce qui en limite considérablement la portée, tout au moins dans un futur immédiat, et l'on assiste actuellement à l'apparition de systèmes mixtes, dont la fusion des deux sociétés produisant GENESIL et GDT était un signe avant-coureur.

### 2.8.2. Principe de l'utilisation de générateurs de modules.

Pour partitionner le système à décrire, on part de sa représentation fonctionnelle, ou de sa représentation structurelle, puis on le décompose en plusieurs sous-systèmes.
A chaque étape on vérifie l'adéquation avec le résultat souhaité, notamment à l'aide de programmes de test et de simulation.
Au niveau le plus bas, on utilise pour l'étude géométrique un générateur de modules approprié (pouvant évaluer la taille et les spécificités du sous-système considéré), ainsi que des algorithmes de placement/routage. A ce stade, le dessinateur peut émettre des hypothèses et suivre alors l'évolution du système afin de revenir éventuellement sur les choix effectués.

### 2.8.3. GDT, un environnement de générateurs de modules

GDT (Generator Development Tools) [Burich 87] est un environnement d'écriture de générateurs de modules utilisés dans des

compilateurs de silicium (voir le paragraphe 2.9), dont notamment le compilateur GENESIL [Cheng 88].
L'environnement de travail de GDT comporte:

1) les outils proposés dans les systèmes de conception "classiques":
- routeurs faisant intrinsèquement partie du langage L (ce qui présente l'avantage de lier le routage à la structure même des cellules à assembler)
- compacteurs permettant de générer des cellules indépendamment de la technologie (le système travaille dans un mode symbolique). Suivant les règles technologiques, ils étendront (transformation de segments en rectangles) puis compacteront le dessin obtenu. Il existe des compacteurs suivant les axes principaux (X, Y) et leurs bissectrices (XY et YX).
- simulateurs (et notamment un simulateur "mixte")
- générateurs de vecteurs de test (basés sur un simulateur des descriptions de modèles)

2) un langage

Il s'agit du langage L, une version du langage C adaptée à la conception de circuits. Le fait d'avoir un langage du type C à sa disposition offre toutes les possibilités de paramétrisation pour écrire des programmes de génération de cellules. Ce langage est particulièrement étudié pour écrire des compilateurs de silicium.

La méthodologie d'écriture de compilateurs de silicium à l'aide de GDT est la suivante [Burich 87]:

-1- l'utilisateur commence par choisir un langage de description fonctionnelle, soit pour des circuits spécialisés (par exemple en traitement du signal), soit pour des circuits standard. Dans le dernier cas le langage choisi sera de type RTL (langage de transfert de registres).

-2- la phase suivante consiste à développer des programmes de passage de la description fonctionnelle à une description logique, dans un format compréhensible par GDT

-3- écrire les cellules standard ou spécialisées nécessitées comme éléments de base du circuit (soit graphiquement, soit en utilisant le langage L).

-4- utiliser les outils disponibles pour le placement et le cablage. Ceux-ci génèrent automatiquement le dessin à partir de la spécification faite dans le format généré à l'étape -2-.

On peut penser qu'une telle approche est relativement simpliste, mais elle est en fait surtout utilisée pour définir des petites entités, ou alors des cellules possédant des facteurs de répétition et de régularité importants.

Remarque:

Une particularité importante de GDT est de permettre la définition "d'icones", des représentations simplifiées des cellules. A chaque icone est associé un certain nombre de caractéristiques de la cellule qu'elle représente. Ceci permet lors d'une simulation, de remplacer la cellule par l'icone correspondante, sans perte d'information.

## 2.9. La compilation de silicium

Le terme "compilation de silicium" a été introduit à la fin des années 70 par Dave Johannsen [Johannsen 79] pour décrire un assemblage paramétré de cellules.
Depuis, ce terme a été relativement galvaudé, et la plupart des systèmes apparaissant sur le marché se qualifiaient eux mêmes de compilateur de silicium.
Actuellement on regroupe sous ce terme des outils relativement différents, bien qu'on admette généralement qu'un compilateur de silicium est un outil permettant de générer le dessin des masques d'un circuit, à partir de sa description dans un langage de haut niveau (idéalement une description fonctionnelle) [Savaria 88][Jerraya 89].
Selon le diagramme en Y de Gajski [Gajski 88], le processus de conception d'un circuit est constitué d'un ensemble d'étapes de raffinements successifs classées selon les trois axes de représentation suivants: l'axe fonctionnel, l'axe structurel et l'axe géométrique:

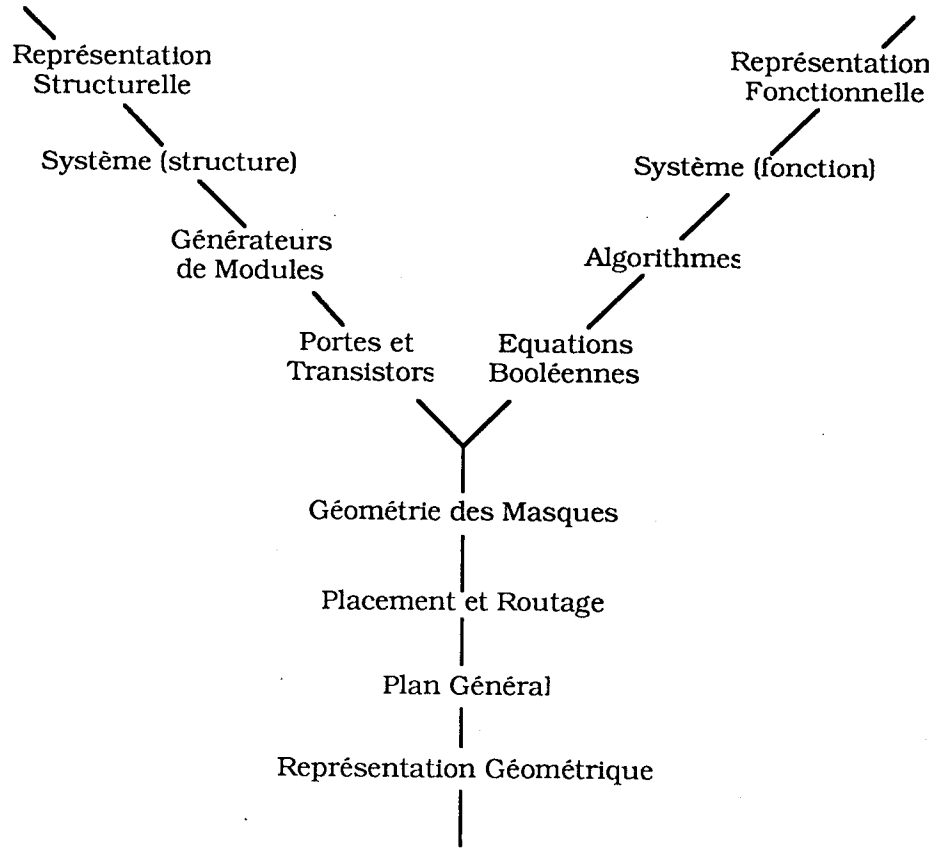
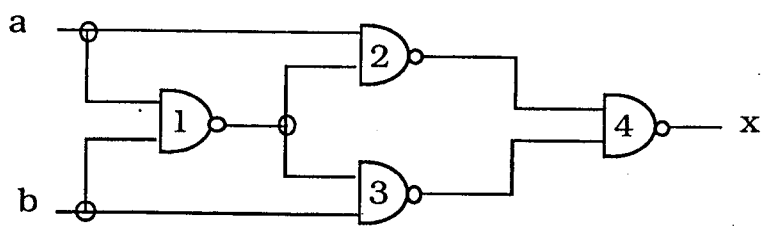



Fig. 2.10: Différentes représentations du processus de conception

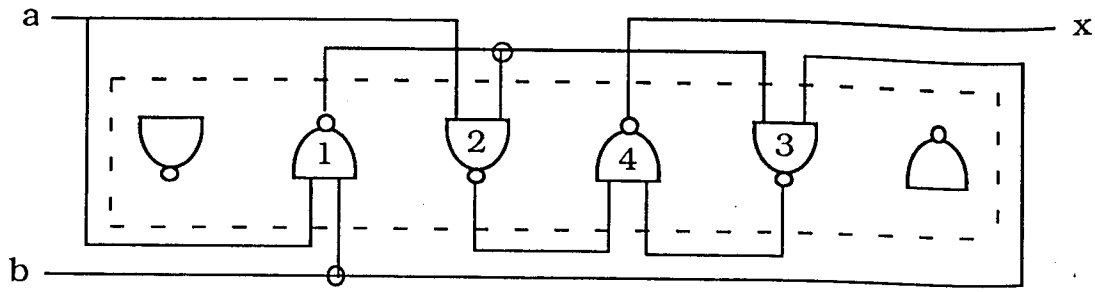
La représentation fonctionnelle (ou comportementale) concerne la fonction du système à construire, la représentation structurelle concerne les modalités de construction du système, et la représentation géométrique concerne les détails de réalisation physique du système:

$x = \bar{a}b + a\bar{b}$ after 30 ns

(a) Représentation comportementale




(b) Représentation structurelle




(c) Représentation géométrique

Fig. 2.11: Différentes représentations de la compilation de silicium

En général, le concepteur expert recourt en alternance aux trois représentations, de façon à faire les compromis les plus efficaces au bon niveau d'abstraction. Idéalement, un compilateur de silicium devrait reproduire fidèlement le comportement d'un concepteur expert en ce qui concerne cette capacité de passer d'une représentation à l'autre, et d'un niveau d'abstraction à l'autre. Cependant ces passages font intervenir des processus trés complexes, et les outils existants sont encore loin d'atteindre cet idéal [Savaria 88].
De Michelli et Jerraya classent eux les langages d'entrée des compilateurs de silicium en quatre catégories [DeMichelli 87] [Jerraya 89]:

- langage de description procédurale du dessin des masques: dans ce cas la conception d'un circuit revient à écrire un programme qui, en s'exécutant génère le dessin des masques du circuit. Ce programme peut faire appel à des cellules d'une bibliothèque et/ou à des programmes générateurs de cellules particulières.
- langage de description logique: le circuit est décrit au niveau des portes. La description peut être organisée hiérarchiquement. Les compilateurs qui partent de ce type de description sont basés sur des techniques de placement et de routage automatiques. Dans le cas général, ces compilateurs génèrent des circuits du type prédiffusé ou des circuits du type précaractérisé. Le lecteur pourra se reporter pour de plus amples informations sur ce type de compilateur dans [DeMichelli 87]
- langage de description d'architecture: le circuit est décrit par un ensemble de blocs interconnectés (où chaque bloc peut être décrit par un langage adapté à son type). La plupart des compilateurs qui partent de ce type de description permettent de générer des circuits organisés à la

demande (en anglais "ASIC"). Ils sont appelés des compilateurs d'architecture.

• langage de description de comportement: un circuit est décrit par ses entrées, ses sorties et la fonction qu'il réalise. L'organisation interne du circuit est générée automatiquement par le compilateur. Les compilateurs qui partent de ce genre de description sont appelés compilateurs de comportement et sont en général spécialisés pour des applications particulières. En fait le problème général, qui consiste à générer la meilleure architecture pour une description donnée, est difficile à résoudre, vu la grande étendue de l'espace des solutions [Director 81]. L'utilisation d'une architecture cible permet de limiter l'ensemble des solutions possibles. En revanche, un tel choix limite l'efficacité du compilateur à une classe restreinte d'applications.

Chaque type de compilateur présente des avantages et des inconvénients. Le choix d'un compilateur de silicium dépend des types de circuits que l'on veut compiler et de la qualification des concepteurs qui auront à l'utiliser.

### 2.9.1. Les compilateurs de descriptions procédurales

Ils fournissent à l'utilisateur ce qu'on appelle un environnement de programmammation de circuits, composé:

• d'un langage procédural pour la description du dessin des masques
• d'une bibliothèque de cellules prédéfinies
• d'une bibliothèque de générateurs de modules

Dans un tel environnement la conception d'un circuit revient à l'écriture d'un programme. Ces compilateurs nécessitent des utilisateurs à la fois experts en programmation et en conception de circuits intégrés. La surface perdue du fait de l'utilisation de ce type de compilateur est trés liée à l'expérience des concepteurs. En fait, comme pour la conception manuelle, la densité des dessins de masques dépend énormément de la stratégie utilisée. En revanche, avec des concepteurs possédant des compétences en programmation il est possible de réaliser des circuits aussi dense que les circuits générés manuellement.

### 2.9.2. Les compilateurs d'architecture

Ils sont formés d'un certain nombre de compilateurs spécialisés (compilateurs de parties opératives, de controleurs, d'opérateurs, ...). La conception d'un circuit à l'aide d'un compilateur d'architecture consiste à définir son architecture à partir de son partitionnement en différents blocs, chaque bloc étant réalisé par un compilateur approprié. Ces systèmes permettent une saisie graphique de l'architecture. Les compilateurs spécialisés garantissent que le dessin des masques généré est correct. De plus, ils fournissent (pour certains seulement) des informations sur les caractéristiques des circuits qu'ils génèrent (telles que des temps de propagation, des consommations, ...). Ces compilateurs sont souvent associés à des bibliothèques de cellules (jeux de plots, d'amplificateurs, de portes logiques, ...) pour réaliser des blocs particuliers, ou pour interfacer le circuit avec l'extérieur.
Ce type de compilateur peut être utilisé par des ingénieurs peu familiarisés avec les arcanes de la conception des circuits intégrés. Il est évident qu'un concepteur expérimenté réalise des circuits de meilleure qualité (des points de vue de la surface et de la vitesse). La durée du cycle de conception est trés réduite: on peut donc essayer plusieurs solutions architecturales, afin de ne conserver que la plus appropriée.

### 2.9.3. Les compilateurs de comportement

Ce type de compilateur ne réclame pas de compétence particulière en conception de circuits intégrés. En revanche ils permettent difficilement de générer des circuits efficaces. Ceci vient de la difficulté d'optimiser certains aspects du circuit tels que la surface ou la consommation en agissant uniquement sur la description de son comportement. D'autre part, la compilation de comportement doit tenir compte de quatre problèmes:

-1- L'architecture: l'espace des solutions architecturales manipulé par le compilateur doit être siuffisamment limité pour permettre une génération automatique et efficace des dessins des masques, et pour permettre des temps de compilation raisonnables. On peut par exemple laisser au compilateur le soin de fixer le nombre d'additionneurs que doit contenir un circuit. En revanche, l'architecture globale du circuit, telle que le modèle "partie opérative/partie controle", est généralement figée pour un compilateur donné.

-2- La description du comportement: elle doit permettre une description précise des circuits. Le choix du langage de description doit tenir compte de l'architecture cible afin de permettre la réalisation d'algorithmes de compilation efficaces quant au temps de compilation et aux performances des circuits obtenus.

-3- La génération automatique de l'architecture: les choix du langage de description et de l'architecture cible doivent tenir compte des algorithmes de compilation. Dans le cas où il existe une bonne correspondance entre le langage de description et l'architecture cible, les algorithmes de compilations sont relativement simples à réaliser. Dans ce dernier cas, il devient possible d'influencer le résultat du compilateur en modifiant la description d'entrée. En revanche, plus cette correspondance est forte, plus on se rapproche d'un compilateur d'architecture où l'architecture est complètement fixée par la description d'entrée.

-4- La génération automatique des dessins de masques: les concepteurs utilisent une grande variété de techniques lors d'une conception manuelle des dessins de masques. La plupart de ces techniques ne sont pas automatisables faute de formalisation.

Pour plus d'informations sur la compilation de silicium, le lecteur pourra se reporter avec profit sur les différents articles écrits par Gajski [Gajski 86], ainsi que l'ouvrage [Gajski 88]. Des compléments pourront également être apportés par [DeMichelli 87]. Ces deux auteurs détaillent notamment le fonctionnement de compilateurs de comportements tels que Mac Pitts [Southard 88], CATHEDRAL II [Rabaey 88] et le Yorktown Silicon Compiler [Brayton 88], de compilateurs d'architecture tels que CONCORDE [Corbin 88] et GENESIL [Cheng 88], et d'environnements de compilation tels que GDT [Burich 87] et les outils de Cadence [Law 87].

### 2.9.4. La compilation de silicium a l'IMAG: SYCO

Nous allons décrire ici quelques points essentiels du compilateur SYCO [Jerraya 86] [Jerraya 89] qui doit utiliser le système NAUTILE pour la génération du dessin des masques (voir paragraphe 5.5).
SYCO appartient à la catégorie des compilateurs comportementaux, il fait un certain nombre d'hypothèses quant à l'architecture des circuits obtenus:

L'architecture

Les circuits générés par SYCO (de type microprocesseur) sont conçus comme des interpréteurs de langage de commande. Un interpréteur est divisé en une pile de couches d'interprétation. Chaque couche traduit les primitives reçues par la couche supérieure en primitives pour la couche inférieure. La couche du bas exécute les commandes élémentaires. Cette couche constitue la partie opérative, les couches supérieures constituant la partie contrôle. Les différentes couches supérieures sont connectées à un bus de contrôle, leur permettant de communiquer avec le monde extérieur.

L'architecture de SYCO restreint son usage à la génération de machines séquentielles.

Le langage de description

Le circuit doit être décrit de façon procédurale en utilisant un sous ensemble du langage de description LDS (Langage de Description de Systèmes ).

L'utilisateur a la possibilité de décrire le parallélisme et le contrôle explicitement. La description algorithmique du circuit est donnée dans une syntaxe de type Pascal, et est constituée d'une hiérarchie d'appels de procédures.

L'organisation de SYCO

SYCO doit générer à partir d'une description algorithmique donnée par l'utilisateur, le dessin des masques du circuit correspondant. A cette fin il va procéder en trois étapes :

-1- la première étape produit l'architecture globale du circuit. Elle génère la description des différentes couches. Cette étape permet de donner une estimation des performances du circuit.

-2- un ensemble de compilateurs spécialisés est utilisé pour générer l'organisation interne des couches.

-3- un ensemble de générateurs de modules est utilisé pour la génération du dessin des masques. Ces modules utilisent la description de l'organisation interne des différentes couches et tiennent compte de la stratégie globale d'agencement de SYCO.

Il faut préciser que le système NAUTILE a permis d'écrire les générateurs de modules utilisés par le compilateur comportemental SYCO.

## 2.10. Les caractéristiques d'un système intégré de conception

Il faut d'abord noter qu'un "bon" système de conception automatique de circuits VLSI ne pourra qu'être un compromis entre différentes solutions, étant donné qu'un système parfait poserait des difficultés de mise en oeuvre, de rapidité d'exécution, de taille mémoire, etc...
Néanmoins, un certain nombre de caractéristiques essentielles doivent être prises en compte, et ce pour aboutir à un système offrant un maximum de souplesse d'emploi pour un minimum de contraintes.

Il semble tout d'abord certain qu'un système entièrement procédural soit d'un abord relativement rebutant pour un utilisateur non nécessairement expert en informatique. Aussi le système comportera-t-il deux parties: l'une graphique et l'autre procédurale.

### 2.10.1. La partie graphique:

Cette partie graphique concerne plusieurs visions différentes du circuit: d'une part ses différentes représentations symboliques, et d'autre part le dessin des masques et le schéma électrique. Aussi pour plus de souplesse d'emploi est-il souhaitable d'uniformiser en ne choisissant qu'un seul outil graphique pour les différentes représentations: ainsi l'utilisateur n'est-il pas dérouté lors du passage d'une représentation à une autre.

La réutilisation de cellules définies par ailleurs étant une composante essentielle de ce genre de système, il doit être possible d'opérer de façon graphique un certain nombre de transformations sur ces cellules: duplication, renommage, translation, rotation, symétrie, et pourquoi pas des fonctions non isométriques (homothéties, affinités orthogonales, ...).
Une solution peut consister à utiliser un système graphique ayant déja fait ses preuves, que l'on interfacerait avec le reste des outils[1].

### 2.10.2. Le langage de description:

Pour s'avérer réellement efficace, la partie textuelle doit présenter un certain nombre de caractéristiques:

---

[1] Une telle solution avait été envisagée pour NAUTILE, afin de ne pas axer le travail sur l'obtention d'un éditeur graphique.

• le langage de description doit être procédural afin d'être bien adapté à une conception hiérarchisée et modulaire.

• il doit être interactif, afin que l'utilisateur puisse mesurer en temps réel les modifications qu'il apporte à son circuit (ceci est grandement facilité par l'usage d'un langage interprété)

• il doit permettre la paramétrisation à tous les niveaux de la conception: ceci afin par exemple de généraliser, de simplifier, de modéliser, de simuler, etc... Ainsi un certain nombre de paramètres seront lus dans un fichier décrivant la technologie, de manière à assurer l'indépendance des niveaux élevés vis-à-vis de celle-ci.

• il doit être convivial: souplesse et simplicité d'emploi sont de mise, afin que les concepteurs ne soient pas pénalisés par une méconnaissance partielle des règles algorithmiques.

Cette dernière condition, de la même façon que pour la partie graphique, est le fruit d'une uniformisation à tous les niveaux: un langage unique sert à la fois de langage de commandes, de langage de programmation du système, et de langage de formalisation des divers générateurs de cellules. Ce langage doit dans ce cas être un langage évolué possédant son propre environnement, comme LISP, PASCAL, ou C. Le nombre de notions que l'utilisateur doit mémoriser se révèle être ainsi considérablement réduit.

F. Anceau [Anceau 84] a distingué trois possibilités d'implantation:

• l'immersion des primitives de description du dessin dans un langage algorithmique procédural et hiérarchisé (C, PASCAL, LISP, ...).
Dans ce cas on peut bénéficier de tout l'environnement existant du langage choisi (ceci permet par exemple d'avoir des débogueurs, des traceurs de fonctions, une analyse syntaxique et lexicale,... pour des langages puissants tels que LeLisp [Chailloux 88] ou SmallTalk [Goldberg 83]) et on peut programmer tous les outils dans un langage unique (celui choisi).
On peut relever cependant un inconvénient provenant de la complexité d'utilisation d'une telle solution: le concepteur doit être lui-même un programmeur.

- la paramétrisation du langage de description par ajout de possibilités algorithmiques (exemples L [Burich 87] et SKILL [Law 87]).
  Ceci impose de créer soi-même des possibilités algorithmiques dans le langage de description. Cette solution permet d'avoir un langage simple et bien adapté aux besoin des concepteurs. En revanche il est nécessaire de gérer toutes les possibilités algorithmiques. Ceci se traduira dans la plupart des cas par une limitation de la puissance de paramétrisation.

- la génération de descriptions statiques par des programmes écrits dans des langages de programmation classique: on se borne alors à l'étude des chaines de caractères générées par le langage de description du circuit (exemples Lucie [Paillotin 85] ou CIF [Sproull 80]). Cette dernière méthode permet une simplification des problèmes rencontrés, et l'utilisation d'un langage quelconque pour la génération du code de description. Cependant elle implique un volume de code important, et présente l'inconvénient de ne pas respecter l'uniformisation des différents langages utilisés.

### 2.10.3. Le lien entre parties graphique et textuelle:

Le système devant supporter à la fois du graphique et du textuel, il faut une structure de données commune, et donc compréhensible par les deux parties.

Si on souhaite pouvoir intervenir à chaque niveau de la conception, il faut qu'il y ait une parfaite adéquation entre représentations graphique et procédurale. On doit pouvoir passer de l'une à l'autre sans problème, et ce de façon interactive. Cela signifie qu'il doit y avoir une correspondance étroite établie entre ces deux représentations.
Une solution intéressante est de posséder les mêmes primitives dans les deux environnements. Ainsi le passage de l'un à l'autre sera-t-il relativement aisé.

### 2.10.4. La structure de données:

Une fois effectué les choix d'une représentation structurée et hiérarchique (nous avons vu les simplifications que de tels choix apportaient), la structure de données doit posséder un certain nombre de caractéristiques, afin de satisfaire à ces conditions:

• la base de données doit permettre la représentation hiérarchique des circuits

• le circuit peut être exprimé suivant différentes vues, des facilités d'ajout de nouvelles vues devant être présentes dans le système.

• la cohérence de l'ensemble du système doit être assurée, en particulier en ce qui concerne les diverses représentations d'un même objet, ou ses versions successives.

• le système doit être capable de fournir aux outils les données déja présentes dans la base de données, et insérées par d'autres outils, ou déductibles de données existantes.

• enfin il doit être capable de proposer des mécanismes d'ajout de nouveaux outils

La compatibilité avec les autres systèmes de conception est essentielle, d'une part si l'on souhaite utiliser des bibliothèques de cellules déja existantes, d'autre part dans le cas où la réutilisation de programmes déja écrits (comme des simulateurs, des compacteurs ou des routeurs) est envisagée, ce qui permettrait un gain notoire de temps au niveau de la conception du système.
Il faut alors prévoir les interfaces en conséquence, ou éventuellement, utiliser des formats standard d'échange de données, du type EDIF ou CIF , par exemple.

Enfin, il convient de remarquer que la représentation en base de données doit permettre d'exprimer à la fois les descriptions procédurales et graphiques, suivant le contexte d'utilisation.

### 2.10.5. La possibilité d'extension

Les outils et leur environnement vont évoluer au cours de la durée de vie d'un projet. Ainsi est-il souvent souhaitable de rajouter de nouveaux outils à l'ensemble de base déja installé, au fur et à mesure de leur disponibilité.

Aussi l'environnement de conception se doit-il d'être suffisamment flexible pour permettre de telles extensions. Il convient d'insister sur cette nécessité d'un environnement "ouvert", dans lequel toutes les informations sont accessibles. De cette façon, de nouveaux outils peuvent aisément venir compléter les anciens, et de nouvelles formes de représentations peuvent étendre celles déja disponibles dans la base de données.
Ces conditions sont nécessaires pour qu'un environnement puisse s'adapter à de nouvelles méthodes de conception.

# 3. PRESENTATION DU SYSTEME NAUTILE

## 3.1. Introduction-Généralités.

Le système NAUTILE s'appuie sur une structure de données orientée objet, adaptée à la conception de circuits intégrés. L'élément de base de cette structure s'appelle la *cellule*. Une cellule peut être construite à partir d'autres cellules: on dit alors qu'il s'agit d'une *cellule de composition*, les cellules dont elle est composée étant appelées ses *sous-cellules*. Chaque sous-cellule pouvant être à son tour décrite à partir d'autres cellules, on dit qu'une cellule de composition forme une *hiérarchie de cellules*, ou également *arbre de composition* de la cellule. Les feuilles de cet arbre, qui ne peuvent donc utiliser d'autres cellules, sont appelées des *motifs*. Les motifs sont donc uniquement formés d'*éléments primitifs*, ou bien consistent en des pointeurs sur des *éléments externes* au système NAUTILE (qui peuvent d'ailleurs dans leur système respectif être définis de façon hiérarchique).
Les cellules peuvent être décrites selon différents niveaux d'abstraction appelés *vues*, certaines de ces vues étant des descriptions *physiques* de la cellule (telles que des descriptions en terme de dessins de masques, de réseaux electriques ou logiques, etc...), alors que d'autres définiront l'interface de la cellule avec le monde extérieur, ou encore la façon dont elle est construite, s'il s'agit d'une cellule de composition. Enfin, une cellule peut être générée par l'évaluation d'un fragment de code écrit dans un langage informatique (ici du Lisp): on dit alors que ce code forme un *générateur de module*.

Nous allons maintenant détailler ces différentes notions, aprés un rappel sur les techniques adoptées pour simplifier la conception d'un circuit intégré.

## 3.2. Les techniques de simplification de la conception

Le problème actuel de la conception des circuits est la gestion de l'ensemble des données générées, leur volume devenant de plus en plus important, avec une intégration de plus en plus poussée. Il est donc nécessaire d'adapter les concepteurs et les outils de travail à cette

évolution pour maitriser la conception des circuits intégrés de demain [Rougeaux 87].
La méthodologie de dessin structurée introduite par Mead et Conway [Mead 80] a ouvert la porte à toute une nouvelle génération d'outils de conception. La complexité des circuits est maitrisée par la recherche de la régularité et de la structuration. La complexité des systèmes est gérée par l'utilisation de techniques de dessin hiérarchique. A la structuration hiérarchique des dessins s'associe la réalisation d'outils travaillant sur des structures hiérarchiques, seul moyen d'obtenir des algorithmes performants [Sequin 83] et [Katz 83].

### 3.2.1. La régularité

L'utilisation de la régularité permet de diminuer à la fois la taille des données à manipuler, et le temps nécessaire à leur traitement.
Son principe consiste à ne définir un objet qu'une seule fois: celui-ci peut être utilisé à de multiples reprises, sa description n'apparaitra qu'à un seul endroit de la base de données. C'est uniquement lors de l'utilisation de l'objet que seront précisées les conditions de son utilisation: ces conditions dépendent à la fois d'éléments additionnels et du contexte courant.
Cependant cette méthode a pour inconvénient de forcer le concepteur à prévoir à l'avance l'assemblage de ses cellules régulières. En effet, lorsque l'on définit une cellule unique (i.e. non régulière), l'interaction entre cette cellule et ses voisines est bien connue: il n'est nul besoin de penser à l'avance les différents cas d'assemblage qui pourraient se présenter pour celle-ci. En revanche, dans le cas de cellules régulières, on ne sait pas a priori comment sera réalisé leur assemblage. Le cout d'un assemblage de cellules non prévues pour s'assembler entre elles étant d'environ 20% supérieur au cas contraire, il y a donc nécessité de s'astreindre à une certaine discipline si on souhaite tirer profit de la régularité. Il conviendra notamment de se doter d'une stratégie d'assemblage prenant en compte les différents cas possibles.

### 3.2.2. La hiérarchie

En conjonction avec l'emploi de la régularité, les concepteurs de circuits intégrés utilisent généralement le concept de hiérarchie pour réduire la complexité de leur dessin.

Son principe consiste à représenter le circuit sous forme de cellules de base (les "motifs"), et de hiérarchies de cellules de composition. La description de circuits par des arbres binaires ou des algorithmes est alors possible.
Un exemple de décomposition hiérarchique d'un circuit est donné dans la figure 3.1:

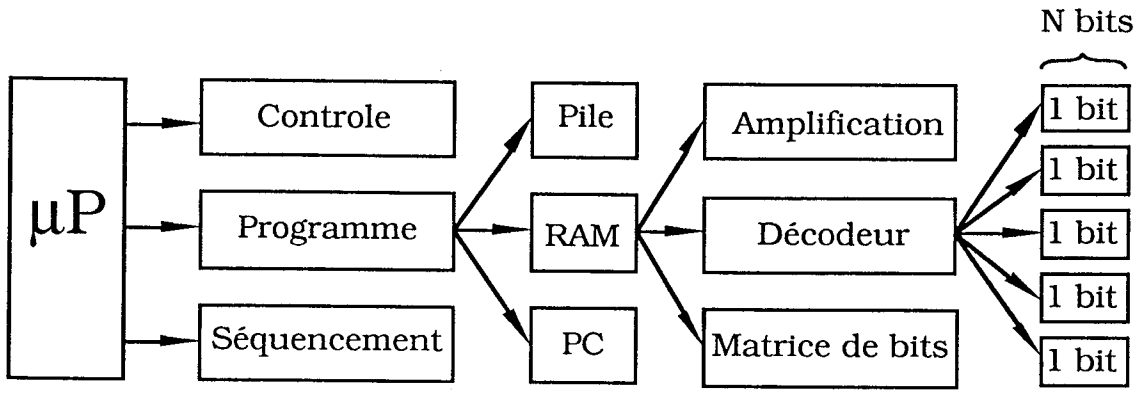



Fig. 3.1: Décomposition hiérarchique d'un microprocesseur

Un motif est atomique, n'a pas de structuration hiérarchique interne, et peut être décrit suivant différents niveaux d'abstraction.

Une cellule de composition est formée de motifs et d'autres cellules de composition. Elle n'a pas de description interne propre, mais comporte essentiellement un répertoire de cellules et d'interconnexions entre ces cellules, ainsi que des informations spécifiant son comportement externe.

La hiérarchisation du problème est indissociable de la notion de modularité. Celle-ci consiste à partitionner le circuit en blocs, chaque module possédant une spécification fonctionnelle trés précise.

## 3.3. Présentation générale de NAUTILE

Le système NAUTILE est composé essentiellement de deux parties (voir figure 3.2):

• une structure de données orientée objet pluraliste (c'est-à-dire pouvant intégrer des données externes à NAUTILE) contenant les différents objets composant les diverses représentations d'un circuit. Cette structure est manipulée par l'intermédiaire d'un ensemble de primitives formant le *gestionnaire de données*.

• un *langage procédural* appelé NIL (Nautile Interactive Language), servant à la fois à la définition de nouveaux outils, à l'interfaçage avec des outils déjà existant, et à la conception des générateurs de module.

Des ensembles d'outils et de générateurs de modules peuvent alors accéder aux données contenues dans la base de données, à travers les primitives du langage NIL, et du gestionnaire de données.

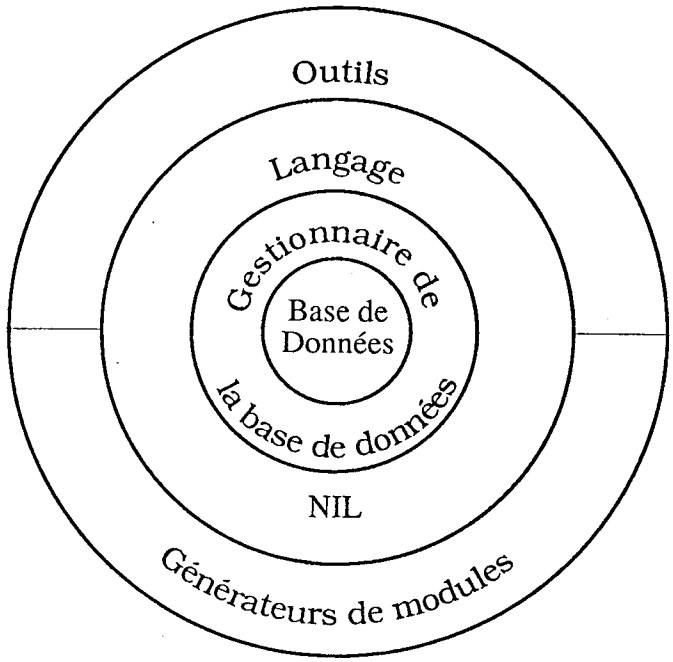



Fig. 3.2: Organisation Générale du Système NAUTILE

Nous allons maintenant nous attacher à détailler ces différentes parties, en commençant par la structure de données.

### 3.3.1. La gestion des données

L'objet de base manipulé par le système est la cellule, chaque cellule représentant généralement une unité fonctionnelle (bien qu'il puisse y avoir des cas où une cellule n'ait pas de représentation fonctionnelle particulière).

Une cellule peut aussi bien être vide[1] que constituer un circuit intégré complexe.
La représentation adoptée est hiérarchique, chaque cellule pouvant en utiliser d'autres pour sa propre définition, qui elles-mêmes utiliseront d'autres cellules, etc... La notion de découpage hiérarchique des cellules autorise la décomposition de la fonction réalisée par une cellule en différentes sous-fonctions réalisées par ses sous-cellules.
Au niveau le plus bas de la hiérarchie (les "feuilles de l'arborescence"), les cellules sont appelées des motifs. Les motifs sont donc des éléments terminaux de la hiérarchie, qui ne possèdent pas de sous-cellule.

### 3.3.2. Cellules de composition et motifs

Les cellules de composition[2] sont construites à partir de motifs et d'autres cellules, le système autorisant le concepteur à mélanger des appels de cellules et de motifs à l'intérieur d'une même cellule.

Une cellule est définie par:

- un ensemble de caractéristiques décrivant son comportement vis-à-vis de son entourage. Il s'agit par exemple de son encombrement ou de ses possibilités de communication avec l'extérieur.
- un ensemble de définitions locales, composant une bibliothèque locale: c'est un ensemble de cellules et de motifs qui sont définis (et utilisés, de préférence) à l'intérieur de la cellule. Ceci permet de cloisonner les différentes parties d'un circuit, afin notamment d'éviter des conflits, essentiellement sur les noms des cellules.
- un arbre d'appel décrivant la hiérarchie. C'est lui qui définit la façon dont les sous-cellules d'une cellule sont agencées à l'intérieur de celle-ci. Il est formé d'une suite d'*instances* de cellules et de motifs, c'est-à-dire d'une liste de noms de cellules et de motifs, ainsi que des propriétés qui leurs sont associées.
- des représentations éventuelles selon certains niveaux d'abstraction, appelés vues. Nous reviendrons sur la notion de vue au cours du paragraphe 3.4.

---

[1] Dans le cas où le concepteur souhaite utiliser une cellule sans définir explicitement sa réalisation, mais en n'indiquant par exemple que la vision externe de cette cellule

[2] Par la suite, nous confondrons les notions de cellule et de cellule de composition

Les motifs, en revanche, ne comportent ni bibliothèque locale, ni arbre d'appels, puisque ce sont des éléments terminaux de la hiérarchie.
Ils sont définis à l'aide d'éléments primitifs ou éventuellement par des pointeurs sur des cellules externes à NAUTILE[1]. Ce que nous appelons des éléments primitifs sont des objets NAUTILE ne possédant de signification que dans un type de représentation bien précis. Ainsi des rectangles ou des polygones seront des éléments primitifs attachés à un contexte géométrique, alors que des capacités ou des nœuds ne seront utilisables que dans une optique électrique. Nous verrons plus avant que les motifs aussi bien que les cellules possèdent par contre des descriptions dans plusieurs vues ou types de vues.

Différents types de motifs peuvent être construits dans NAUTILE, ces types ne se retrouvant pas forcément tous dans un même circuit, leur utilisation étant plutot fonction du "style" de dessin du concepteur:

- Les motifs prédéfinis dans le système (on parle alors de "motifs système"). Il s'agit d'éléments de base de la technologie, tels que transistors, fils ou contacts. Ces motifs sont définis par des primitives du langage NIL présentes au chargement du système (il existe donc des primitives "transistor", "fil" et "contact").

- Les motifs définis manuellement par l'utilisateur, appelés "motifs utilisateur": ce sont par exemple des portes élémentaires, des cellules simples, etc...
  Ces motifs peuvent être soit importés d'une bibliothèque déjà existante, soit créés pour les besoins immédiats. En général les motifs utilisateurs décrivent des entités assez simples, comme des portes logiques élémentaires ("inverseur", "et", "non-ou", ...). Ceci ne constitue cependant pas une règle absolue: un concepteur pourra trés bien choisir d'optimiser certaines parties critiques de son circuit, qu'il définira alors comme des motifs, court-circuitant ainsi la hiérarchie.

---

[1] La notion d'utilisation d'éléments externes au système est essentielle, et sera développée au paragraphe 4.3.

- Les motifs créés par des outils: par exemple les routeurs génèrent lorsqu'ils sont appelés un motif contenant le résultat du routage entre les différentes cellules impliquées (en fait les caractéristiques du canal de routage).

- Les motifs résultant de l'évaluation d'un générateur de modules (tels des générateurs de PLA, de chemin de données, etc ...). En effet, dans le cas de PLA, il n'est pas forcément judicieux d'utiliser une décomposition hiérarchique des cellules: il peut s'avérer préférable d'employer des générateurs permettant de générer concuremment les modèles logiques, électriques et temporels du PLA. Quant au modèle topologique, il sera plus compact et facile à manipuler. Ces générateurs peuvent soit provenir de systèmes externes, soit être directement écrits en NIL.

La notion de motif recouvre donc une grande quantité d'objets différents, et l'on peut parfois se demander quel peut être le role dévolu aux cellules. Bien qu'aucune réponse définitive ne puisse être apportée à cette question, il convient de rappeler que l'emploi des motifs est entièrement laissé à la discrétion du concepteur. Ainsi un concepteur souhaitant optimiser son dessin utilisera-t'il probablement des motifs complexes, alors que dans un cas où les problèmes de performances et de surface seront moins cruciaux, c'est l'aspect rapidité d'exécution du dessin qui l'emportera, et le concepteur s'en remettra alors au système pour obtenir le circuit final. Dans ce dernier cas, la plupart des motifs utilisés seront alors des générateurs, des motifs de base, et des motifs résultants de l'application d'outils.

### 3.3.4. Relations entre cellules et motifs

On retrouve les notions de cellules et de motifs dans d'autres systèmes, mais avec généralement des limitations par rapport à NAUTILE: ainsi LOF [Bergé 86] distingue les "blocs" (les cellules de NAUTILE) des "cellules de base" (nos motifs). Cependant NAUTILE va plus loin, en étendant la notion de motifs aux outils disponibles dans le système. En effet, pour NAUTILE, le résultat de certains outils tels que les routeurs est considéré comme motif.

D'autre part, des systèmes tels que GDT [Burich 87] utilisent la notion de motifs, mais ceux-ci sont fixés dans le système et ne peuvent être changés par l'utilisateur.
Enfin on peut considérer que dans les systèmes utilisant des cellules standard, celles-ci sont assimilables à des motifs. Cependant si ces systèmes sont trés bien adaptés pour l'utilisation qui en est faite, ils sont en revanche trés limités, notamment en ce qui concerne des possibilités de paramétrisation et d'intégration.

Méthodologie d'utilisation:

La distinction entre cellules et motifs a été introduite en NAUTILE, afin de ne pas mélanger des appels de cellules (ou de motifs), avec l'utilisation d'objets primitifs (tels que des rectangles). Le concepteur utilisera donc des cellules pour la partie description hiérarchique, et des motifs pour la partie spécification physique. Cette séparation des deux notions permet de faciliter les vérifications, et notamment de simplifier des algorithmes tels que ceux de vérification des règles de dessin (DRC), qui n'ont alors plus qu'à vérifier les motifs. Ces concepts d'élimination des vérifications pour les cellules, seront approfondis dans les paragraphes 4.1 et 4.4.

Cependant, si le concepteur ne peut mélanger des appels de cellules (ou de motifs) avec des éléments primitifs, nous avons préféré lui laisser la liberté de combiner des cellules et des motifs à l'intérieur d'une même cellule. Ce choix est criticable car il peut être génant d'un point de vue fonctionnel de mélanger ainsi des éléments terminaux avec d'autres qui ne le sont pas: ainsi l'ajout d'un simple transistor (qui est un motif prédéfini) au sommet de la hiérarchie risque de compromettre tout l'édifice. Cependant, même si l'on interdit une telle facilité, il sera toujours aisé de contourner l'interdiction: ainsi il suffirait pour rajouter un transistor à n'importe quel niveau de hiérarchie de créer une cellule qui n'appellerait que ce transistor, puis de l'instancier à l'endroit choisi! C'est la raison pour laquelle nous avons choisi de ne pas imposer cette limitation.

### 3.4. La notion de vue

Dans un système de description des circuits intégrés, il convient de prendre en compte les différent aspects d'un objet: représentation

topologique, électrique (en vue de simulation ou de vérification), temporelle (détermination des temps de réaction d'un circuit), etc...

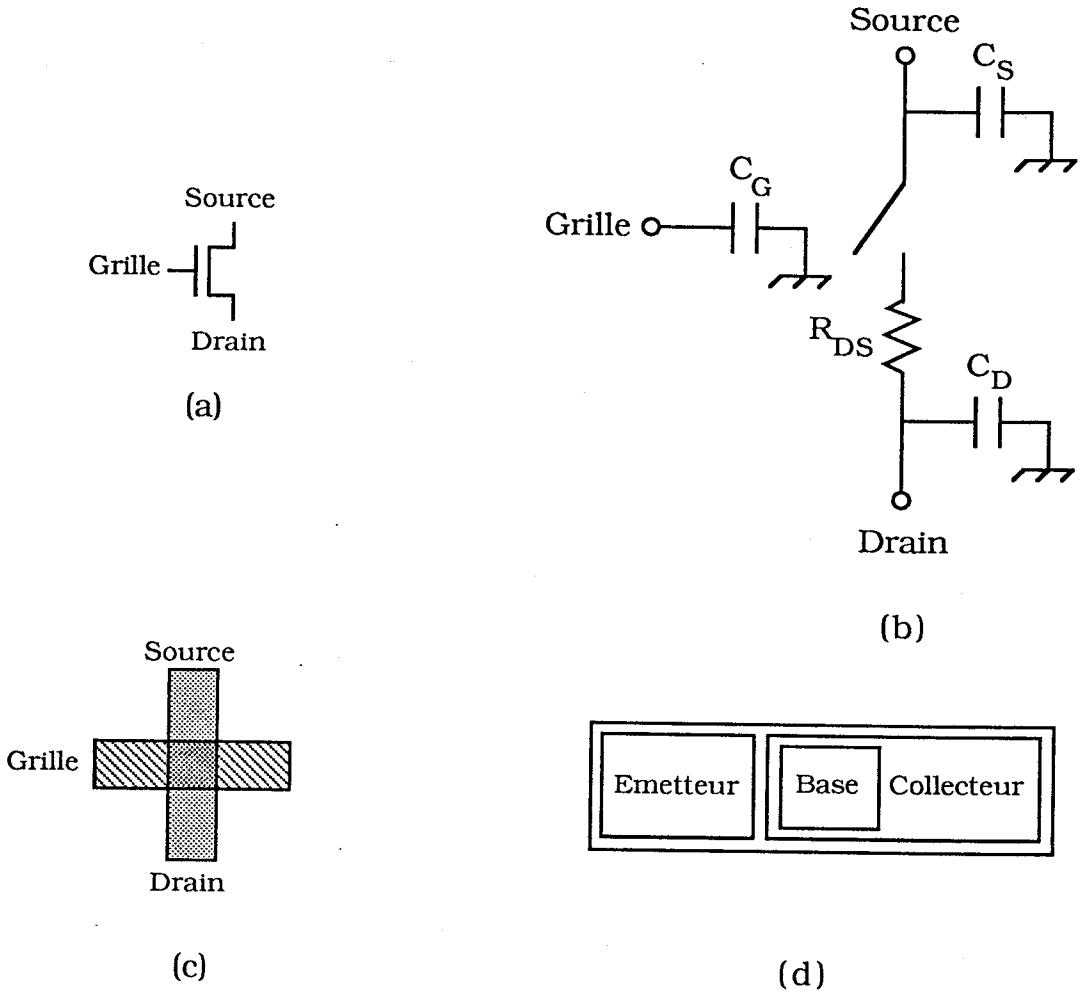



Fig. 3.3: : Différentes représentations d'un transistor

La figure 3.3 présente quatre vues possibles pour un transistor:

(a): une vue schématique symbolique
(b): une vue schématique électrique en vue de simulation
(c): une vue topologique MOS
(d): une vue topologique bipolaire.

Dans le système NAUTILE, les différentes représentations d'un objet s'appellent des *vues*.

Deux types de vues existent en fait dans NAUTILE: il s'agit d'une part de *vues physiques*, propres à un mode de représentation bien défini, et d'autre part de deux vues générales, constituant un noyau commun à toutes les vues, appelées la *vue environnement* et la *vue de construction*.

### 3.4.1. Les vues physiques

Les vues physiques décrivent les cellules de façon particulière à chaque type de représentation. Il existe d'origine un certain nombre de vues prédéfinies dans NAUTILE (actuellement une vue topologique, décrivant la cellule en termes de rectangles et de polygones, et une vue électrique pour laquelle ce seront des réseaux de transistors, résistances et capacités), mais l'une de ses principales originalités est de permettre aisément l'ajout de nouvelles vues. Des mécanismes sont ainsi proposés permettant d'enrichir l'environnement de base: ces mécanismes seront détaillés dans le paragraphe 4.3.
Il convient également de remarquer qu'il peut y avoir plusieurs vues d'un même type: ainsi peut-on considérer une vue topologique décrivant le circuit en terme de dessins de masques, une vue topologique schématique, ou bien encore une vue symbolique utilisant uniquement des symboles.
De même, différentes vues électriques peuvent coexister, suivant le degré de précision souhaité, et l'usage auquel elles sont destinées: un concepteur n'a pas forcément les mêmes besoins pour effectuer une simulation électrique, que pour la documentation de son projet. Il pourra alors être nécessaire de considérer une vue de type réseau de transistors, les motifs étant représentés par des transistors, et une vue électrique plus fine décrivant le circuit en terme de résistances, capacités, inductances, etc...

Cette possibilité de diversifier les vues, et d'accepter pour celles-ci n'importe quel format (pourvu que l'on ait défini les primitives de manipulation de la base de données), constitue une des grandes originalités de NAUTILE: celui-ci n'impose aucune restriction au départ, et permet suivant l'application considérée de choisir la bibliothèque de base la plus adéquate.
Il convient donc de distinguer deux niveaux d'utilisation de NAUTILE:

- dans une première phase il est nécessaire de définir toutes les vues que l'on souhaite utiliser, ainsi que les outils qui les manipuleront. On n'est alors limité d'aucune façon sur le choix et le nombre des vues: c'est la phase de personnalisation du système.
- la deuxième phase dépend de la méthodologie de conception, mais en général le nombre de vues est limité, et les relations entre les différentes vues sont définies une fois pour toute: cette phase est la phase d'utilisation.

### 3.4.2. La vue construction

Nous avons vu qu'une cellule était formée d'un assemblage hiérarchique de cellules de base (les motifs), et d'autres hiérarchies (les cellules de composition). C'est la vue de construction qui va permettre de décrire cet assemblage. Cette vue comporte une liste d'*appels* (ou *instances*) d'autres cellules, motifs, et même éventuellement d'outils. Un appel consiste à mémoriser le nom de la cellule appelée, complété par le contexte d'appel de cette cellule. Ce contexte représente la somme d'informations nécessaires pour placer l'instance dans la cellule appelante, telle que sa position, les transformations qu'elle subit, la valeur de ses différents paramètres et, s'il y a lieu une référence à l'outil qui l'a générée. La vue construction est donc un ensemble d'instances de cellules, de motifs et d'outils.

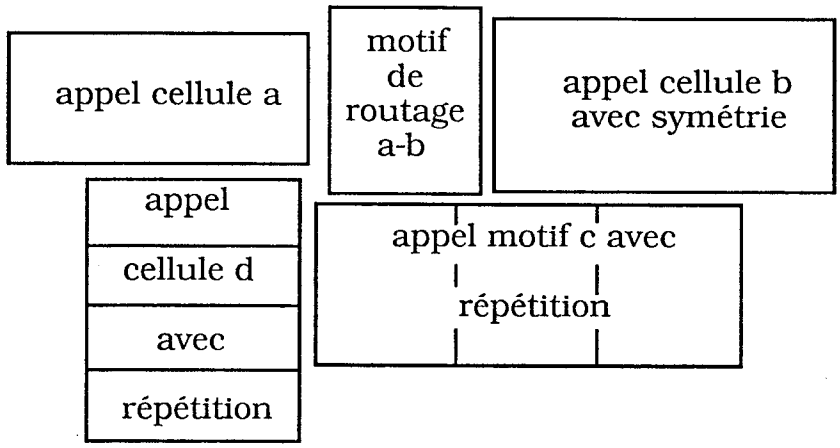



Fig. 3.4: Exemple de cellule produit d'un assemblage

Une description de la suite d'actions nécessaire à la construction de l'exemple de la figure 3.4 pourrait être:

```
(defcell 'abcd)
    ;définition de la cellule de nom "abcd"
(instance 'd 'd1 0 0 '((rep 4 nord)))
    ;appel de d en (0,0) répété 4 fois verticalement, appelé d1
(direct 'est 'd1 'c 0 10 '((rep 3 est)))
    ;aboutement de c par rapport à l'instance d1 avec décalage de (0,10)
(direct 'nord 'd1 'a -10 0 () 'a1)
    ;aboutement de a par rapport à l'instance d1 avec décalage de (-10,0)
(rout 'est 'a1 'b 0 0 '((sym x)))
```

```
    ;appel du routeur pour ajouter b avec symétrie par rapport à a1
(fin-acces)
    ;fin de définition de la cellule
```

La vue de construction résultante consisterait alors à mémoriser pour chaque nouvelle cellule ajoutée le nom de la cellule ainsi que son contexte d'appel. Ainsi pour l'appel:

```
(direct 'est 'd1 'c 0 10 '((rep 3 est)))
```

on mémoriserait:

le nom de la cellule appelée: c
sa position: (0,10)
le nom de l'instance qui lui sert de référence: d1
la liste de transformations à lui appliquer: ((rep 3 est))
son nom d'instance (ici généré automatiquement: c13 par exemple)
l'outil qui l'a générée: direct

Cette dernière information n'est en fait pas contenue telle quelle, mais sous forme de type: l'instance en question appartient au type "direct".

Cet exemple illustre bien le mode de fonctionnement de NAUTILE: on peut voir que la vue de construction de la cellule représentée utilise aussi bien des appels de cellules (a, b et d), que de motifs (c) ou même d'outils (le motif de routage entre les cellules a et b). En effet, NAUTILE considère que chaque outil construit des motifs utilisables comme n'importe quels autres motifs.

Les primitives de construction de base:

Les primitives d'assemblage sont essentielles dans NAUTILE, puisque ce sont elles qui définissent la façon dont sont produites les instances.
Il en existe quatre d'origine dans NAUTILE, rassemblées dans la figure 3.5:

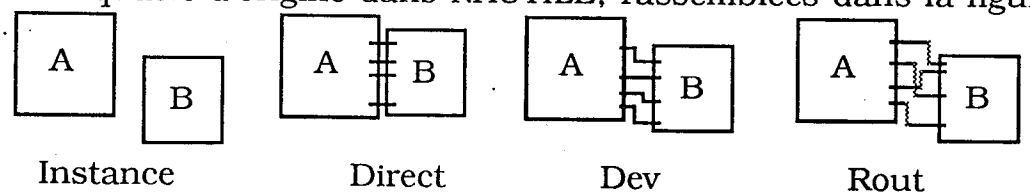



Fig. 3.5: Les différentes primitives d'assemblage

"instance" place l'instance de l'entité choisie à une position donnée
"direct" opère un routage élémentaire sur les connecteurs en vis-à-vis
"dev" créé un motif de routage monocouche avec obstacles
"rout" créé un motif de routage bicouches avec obstacles

Ces différentes fonctions permettent un placement:

- absolu, par exemple: (instance X Y)
- en fixant la référence, par exemple: (instance ΔX ΔY)
- relatif, par exemple: (direct 'nord)

Stratégies de connexion

Dans tous les types d'assemblage énumérés ci-dessus, excepté pour la primitive "instance", le système doit effectuer des connexions entre un certain nombre de connecteurs. Il est donc nécessaire de savoir ce qu'il faut indiquer au routeur, afin qu'il puisse effectuer ces connexions. Cependant, il ne sera nécessaire de préciser les connecteurs à relier entre eux que dans le cas de la primitive "rout". Dans le cas des primitives "dev" et "direct", les connecteurs à relier sont implicitement connus: pour la primitive "direct" il s'agit des connecteurs en vis-à-vis, alors que pour la primitive "dev" on opérera des connexions entre les connecteurs dans leur ordre d'implantation géométrique. Il est toutefois toujours possible d'indiquer explicitement l'ensemble des connecteurs qui doivent être reliés entre eux, pourvu qu'ils respectent les conditions du routage que l'on souhaite leur appliquer (i.e. pour un assemblage direct il faut que les connecteurs soient en vis-à-vis, et pour un routage par dévoiement, il faut que celui-ci puisse être réalisé avec une seule couche de métallisation). Par ailleurs, lorsque l'on souhaite utiliser la primitive "instance", il sera nécessaire d'utiliser explicitement une fonction générant des connexions entre les connecteurs que l'on souhaite relier (la fonction "connect"): cette primitive n'opère en effet aucune connexion implicite ou explicite.

Un autre problème (non résolu par NAUTILE) introduit par ces primitives est celui du contexte de leur utilisation. En effet, suivant ce contexte il peut être intéressant d'opérer un aboutement simple (primitive "direct"), ou avec dévoiement (primitive "dev"). Or si le contexte d'appel change, il peut être nécessaire de changer le type d'assemblage, ce qui n'est malheureusement pas fait dans NAUTILE. La seule solution que nous puissions donner à ce problème, est l'utilisation systématique de la primitive "rout", qui opère un routage général. Cette solution n'est malheureusement pas trés acceptable dans la plupart des cas, car elle peut conduire à un routage non optimal dans des cas simple, et surtout elle peut s'avérer trés couteuse en temps, ce qui est pénalisant dans le cas d'assemblage direct (le type le plus fréquent d'assemblage, en fait).

L'utilisation d'un système expert dans le routage, permettrait probablement de résoudre ce problème, mais en introduisant certainement des dégradations des temps de calcul.

Une connectique guidée par l'assemblage

Dans la version actuelle de NAUTILE c'est la connectique qui est guidée par l'assemblage, et non le contraire. En effet, s'il peut paraitre intéressant de laisser le système calculer l'emplacement relatif des différents blocs d'aprés les connexions qui les relient, cela nous a semblé entrainer des décisions relativement complexes, et nous avons préféré garder cette possibilité pour des développements futurs. Actuellement, la méthodologie adoptée consiste donc à imposer le placement des blocs, les connexions étant réalisées une fois le placement terminé (le seul paramètre pouvant varier étant la largeur du canal de routage).

### 3.4.3. La vue environnement

La vue environnement représente "l'interface" de la cellule, telle que définie dans EDIF (en revanche, EDIF regroupe toutes les autres vues sous le terme "implementation", alors que nous distinguons la vue de construction des vues physiques).
Cette vue environnement exprime les relations de la cellule avec son environnement: c'est la cellule "telle qu'elle est vue pour son utilisation". Elle doit donc contenir suffisamment d'informations sur la cellule, pour que celle-ci puisse être utilisée sans connaissance approfondie de sa réalisation physique.
Elle documente la cellule et présente explicitement les informations de connectivité, et comporte:

- des paramètres (tels que facteurs de taille, de répétition,...), qui sont de deux types: les paramètres systèmes, lus dans un fichier (technologique par exemple) et les paramètres utilisateurs, qui décrivent les différentes possibilités offertes à une classe de cellules. Nous reviendrons sur la définition et l'utilisation des paramètres au cours du paragraphe 3.7.
- un facteur mesurant l'encombrement ou "frontière" de la cellule
- des connecteurs (les moyens de communication de la cellule avec l'extérieur) ainsi que leurs propriétés. Ceux-ci possèdent un nom, un type, une direction privilégiée, un nom d'équipotentielle, et sont

placés sur la périphérie de la boite enveloppante. Les connecteurs possédant le même nom d'équipotentielle doivent être reliés entre eux d'une façon ou d'une autre à l'intérieur de la cellule.

- éventuellement une icone permettant de remplacer la cellule par une représentation symbolique simplifiée de celle-ci.

Pour des soucis de simplification, les cellules sont considérées de l'extérieur comme des boites, avec des connecteurs positionnés sur le pourtour de ces boites, ce qui permet de les manipuler sans tenir compte de ce qu'elles contiennent.
Certains pourraient trouver excessif le fait de typer les connecteurs, et de leur attribuer un certain nombre d'informations (comme la position ou des informations sur le niveau technologique dans lesquels ils sont réalisés). Cependant ceci permet de procéder à des vérifications élémentaires, et facilite le pilotage des différents outils d'assemblage (notamment les routeurs).
Les noms des équipotentielles peuvent aussi bien être locaux que globaux. Tous les connecteurs dont le nom d'équipotentielle est global sont implicitement connectés. Vdd et Gnd sont typiquement des noms d'équipotentielles globales.

Les différents éléments composant la vue environnement peuvent aussi bien être calculés que donnés par l'utilisateur (c'est notamment le cas pour les motifs, le système NAUTILE ne sachant pas dans la version actuelle extraire d'un motif les informations nécessaires à la vue environnement). Ainsi pour la boite enveloppante: cette boite est calculée automatiquement par NAUTILE (dans le cas des cellules). Cependant il peut s'avérer intéressant de changer cette boite, ce que peut faire le concepteur à n'importe quel moment de la conception. Un tel cas est présenté dans la figure 3.6:

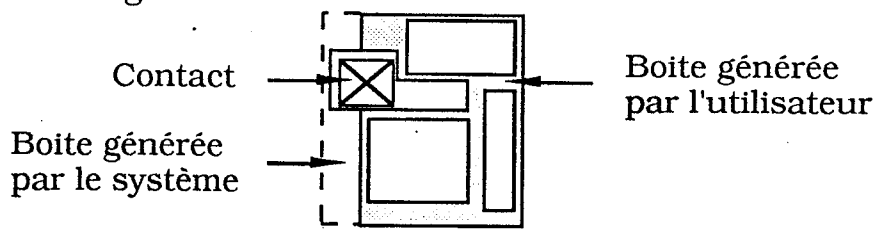



Fig. 3.6: Cas d'une boite générée automatiquement à modifier

On voit ici qu'il est plus judicieux de ne pas inclure le contact en entier dans la boite enveloppante. L'utilisateur redéfinit alors la boite afin de tenir compte de cette caractéristique spéciale.

La vue environnement: des informations communes à toutes les vues

Il est important de remarquer que contrairement à d'autres systèmes (OCT [Moore 86] et ADAM [Knapp 85] entre autres), NAUTILE considère que la plupart des paramètres et des attributs (notamment les connecteurs), sont communs à toutes les vues: ceci permet, outre une simplification du problème d'assurer des liens entre les vues, et surtout le maintien de la cohérence entre elles. De plus cela permet aux différentes primitives et outils NAUTILE de posséder des sémantiques multiples: ils peuvent agir indifféremment sur toutes les vues, c'est le contexte de leur appel qui va fixer la fonction qu'ils réalisent (nous reviendrons sur ce point dans le paragraphe 5.4 consacré à l'environnement de travail).

Le problème d'une représentation simplifiée de la cellule

Cette représentation peut se faire de deux manières: soit en indiquant uniquement les limites de la cellule (la boite enveloppante) et éventuellement des points d'entrée/sortie (les connecteurs), soit en adoptant une représentation symbolique de celle-ci permettant une simplification et une standardisation du schéma représentatif du circuit: c'est ce qu'on appelle une *icone*.
Nous allons maintenant détailler ces deux modes de représentation.

La façon la plus simple de représenter la frontière d'une cellule, est de délimiter celle-ci par un rectangle, englobant tous les éléments constituant la cellule:
Cette solution, facile à mettre en œuvre, est celle qui est adoptée dans la version actuelle de NAUTILE: en effet elle simplifie la réalisation tout en permettant un gain important au niveau des temps de traitement (si on interdit le recouvrement des cellules entre elles).
En revanche, elle présente l'inconvénient de pertes importantes de surface.

Cependant, on peut souhaiter améliorer la lisibilité du dessin ainsi que sa compréhension, en se définissant une représentation de la cellule prenant en compte la fonction de cette cellule: c'est notamment le principe des systèmes de représentation symbolique.
Nous avons souhaité doter NAUTILE de telles possibilités, sans cependant recréer un véritable environnement de conception symbolique (et surtout

les outils complexes qu'il nécessite). Aussi avons-nous préféré utiliser la notion d'icone: celle-ci représente une abstraction de la cellule permettant de remplacer celle-ci au cours dans des descriptions du type logique, ou électrique.
Pour l'instant il n'existe qu'une seule icone par cellule, donc commune aux différentes vues de la cellule. Cependant, rien n'empêche d'imaginer que dans le futur toute cellule possédera une icone par vue, permettant de se substituer à la cellule dans la vue en question.
Il pourra alors devenir nécessaire d'associer à ces icones un certain nombre d'informations, et notamment une description électrique permettant de simuler uniquement avec des icones.

### 3.4.4. Relation entre les différentes vues

La figure 3.7 décrit le principe de fonctionnement des différentes vues dans le système NAUTILE:

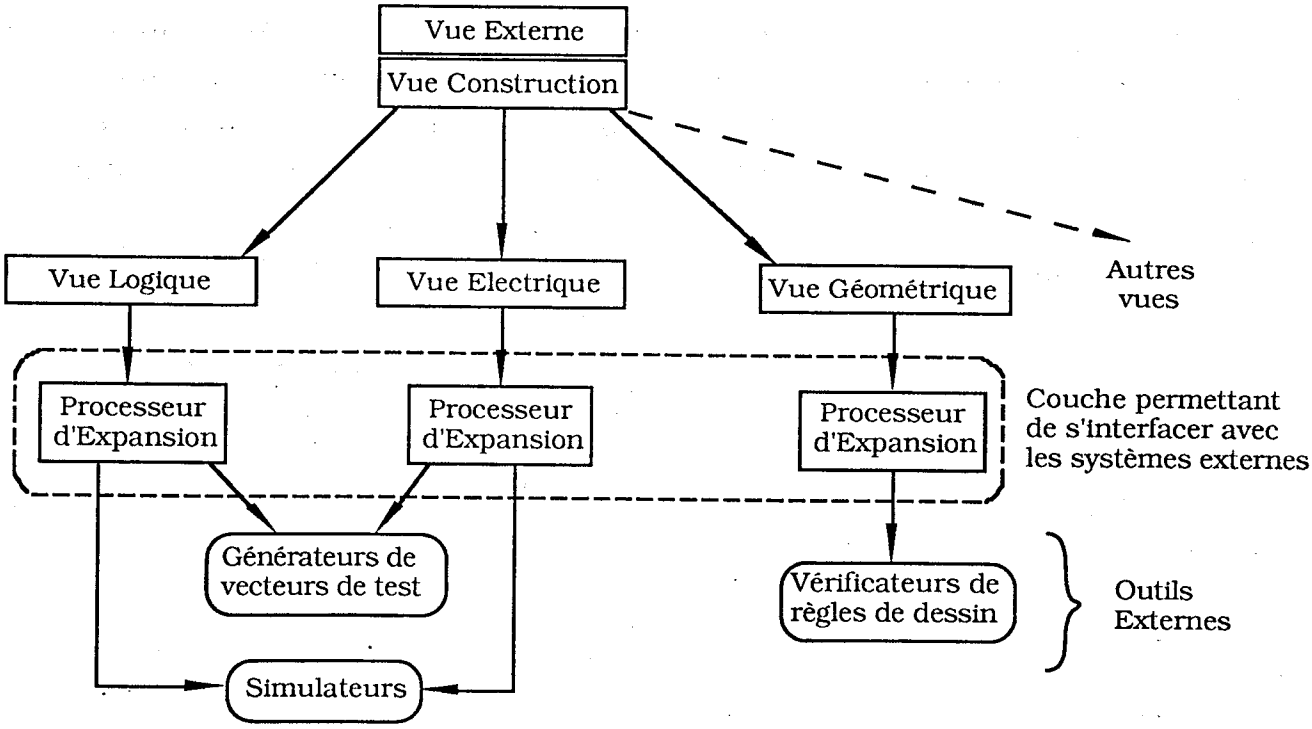



Fig. 3.7: Relations entre les différentes vues en NAUTILE:

Le système NAUTILE est organisé autour d'un certain nombre de vues, chacune d'entre elles pouvant être décrite aussi bien textuellement que graphiquement, ou encore être issue de l'évaluation d'un outil spécialisé ou encore d'un générateur. Le nombre de vues physiques n'est pas limité.

Seules les cellules de composition possèdent une vue de construction, les motifs possèdant uniquement la vue environnement, et des vues physiques. Aussi est-il indispensable lors de la construction d'un circuit de s'assurer que le concepteur possède bien toutes les vues pour chaque motif, et tous les motifs nécessaires à la définition de ce circuit, faute de quoi il ne pourra pas extraire toutes les vues possibles de son circuit.

Pour les cellules, les vues physiques peuvent être soit générées automatiquement à partir de la vue de construction, soit spécifiées explicitement: elles proviennent alors généralement d'une mise à plat de l'arbre d'appels dans une vue donnée.
D'autre part, si le concepteur a toujours la possibilité de définir pour une cellule donnée une vue physique quelconque, un tel schéma de conception n'est évidemment pas trés sûr: la vérification de cohérence entre la vue physique calculée à partir de la vue de construction, et la vue physique rapportée, devra être effectuée par des outils externes classiques.

Les vues physiques d'une cellule ou d'un motif peuvent être soit définies dans le langage NIL, soit importées d'autres systèmes en utilisant une primitive d'importation: la fonction "charge". Réciproquement, une vue décrite en NAUTILE peut être traduite dans un système quelconque à partir du moment ou l'on définit la primitive d'exportation (la fonction "génère"). Ces problèmes d'interfaces avec les systèmes externes seront traités plus en détail dans le paragraphe 4.7 intitulé "interfaçage avec les systèmes externes".

Enfin, on peut se demander pourquoi des possibilités de passage d'une vue à l'autre ne sont pas présentes dans NAUTILE: cette caractéristique ne nous parait pas en effet relever d'un besoin prioritaire. Quoi qu'il en soit, de nombreux outils existants permettent déjà de résoudre de tels problèmes, la plupart du temps avec succés.

## 3.5. Les mécanismes associés à la gestion des données

La gestion des données est assurée par l'ensemble des primitives formant le gestionnaire des données. Elles ne sont accessibles qu'à travers ce gestionnaire, et en aucun cas directement.
Les primitives de gestion des données comportent essentiellement:

- des mécanismes d'ajout et de suppression de données

- des mécanismes de consultation de la structure
- des mécanismes d'accés et de sauvegarde, permettant d'amener en mémoire centrale une hiérarchie, ou au contraire de la sauvegarder dans le système de fichier hote.
- des mécanismes de nommage des objets

Le nommage des objets est fait de façon structurée et algorithmique, afin de permettre un meilleur partitionnement de l'espace des noms (nous examinerons plus en détail la portée des noms dans le paragraphe 5.4.1).
D'autres primitives de gestion de la base de données sont également disponibles, telles que des primitives de recherche par filtrage (soit sur une clef, soit sur un type), ainsi que des fonctions de gestion de la cohérence (nous reviendrons sur les problèmes liés à la gestion de la cohérence au cours du paragraphe 4.1).

## 3.6. Le langage NIL

Le langage NIL est immergé dans CEYX [Hullot 84], qui est lui-même une extension orientée objet du langage Le-Lisp, un dialecte français de Lisp [Chailloux 88].
Son role est de manipuler la représentation des objets, alors que les primitives du gestionnaire de données s'appliquent, elles, à la structure des objets.

NIL se compose de trois parties distinctes:

- un ensemble de facilités procédurales permettant d'effectuer des constructions algorithmiques classiques (notamment des structures de controle): des caractéristiques notamment absentes de langages d'interface tels que EDIF.

- un ensemble de primitives d'assemblage permettant de composer entre eux les cellules et les motifs

- un ensemble d'extensions diverses, chacune d'entre elles dédiée à une vue précise (par exemple la primitive "addrect" qui permet d'ajouter un rectangle dans un motif est une primitive spécifique de la vue topologique, alors que la primitive "addcapa" ne sera valable que dans un environnement électrique)

L'ensemble de ces primitives permet d'obtenir un langage à la fois souple et riche, ayant pour contre-partie un caractère verbeux certain.
Le concepteur peut utiliser NIL de deux manières différentes:

- soit il construit sa cellule progressivement et de façon interactive en utilisant les primitives NIL et éventuellement les quelques facilités graphiques disponibles

- soit il écrit dans un fichier la suite d'instructions NIL lui permettant de créer la cellule souhaitée, reportant l'évaluation et donc la création de la cellule à plus tard. Il lui est alors entièrement loisible de rajouter des paramètres et des fonctions de controle sophistiquées.

Une cellule sauvegardée sous cette forme s'appelle un générateur de module. C'est l'évaluation du générateur de module qui va créer la cellule elle-même dans la structure de données, une fois que tous les paramètres nécessaires ont été fixés.

C'est également à travers NIL que les outils vont manipuler la structure de données. Ces outils peuvent soit être écrit directement en NIL, soit être des outils extérieurs: dans ce cas, seule la partie interface de l'outil avec NAUTILE est écrite en NIL.

## 3.7. Les paramètres

Il est souvent intéressant de définir des paramètres tels que facteurs de taille (notamment pour les transistors), de répétition, etc... Ces paramètres interfèrent évidemment avec la surface totale du circuit, mais également avec ses performances (fiabilité, rapidité, consommation,...). Il est donc trés important de se définir un compromis correspondant à ce qui est souhaitable.

La taille d'un circuit s'accroit généralement avec sa complexité: il peut être intéressant de définir comme paramètre la taille maximale que l'on souhaite donner au produit obtenu. Un tel résultat peut parfois être rendu réalisable grace à des outils de compactage.

La gestion des paramètres

Deux solutions ont été envisagées pour prendre en compte la gestion des paramètres:

- une solution statique: l'utilisateur écrit des générateurs, c'est-à-dire des fonctions dans un fichier de type texte, en insérant dans ces fonctions des paramètres et des variables. L'évaluation de ces fonctions va créer les cellules souhaitées dans la structure de données, une fois les valeurs des paramètres et variables fixées.
- une solution dynamique: les paramètres font partie intégrante de la structure (c'est-à-dire que l'on garde dans la structure d'une cellule les paramètres qui lui sont propres, puis lors de l'évaluation de cette cellule on demande à l'utilisateur de fixer la valeur de ces paramètres). Le problème dans ce cas est qu'il faut pouvoir bloquer l'évaluation de la cellule, et garder trace dans la structure de tout ce qu'il faudra évaluer ultérieurement.

La solution adoptée par NAUTILE est en fait un mélange des deux: ainsi les paramètres ne sont pris en compte qu'au niveau des générateurs (ceci est fait pour avoir à chaque fois des bases solides, notamment pour le calcul des boites), mais en revanche l'appel à certaines primitives (comme notamment les primitives de construction) est conservé dans la structure de la cellule, pour être évalué ultérieurement, lors de l'utilisation de cette cellule, suivant le contexte (i.e. l'environnement de la cellule appelante, mais aussi la vue courante et le mode de description souhaité: graphique ou textuel).

Remarque:

S'il pourrait parfois être intéressant de préciser au préalable un certain nombre de caractéristiques électriques du circuit telles que consommation, dissipation, délais, etc... , dans tous les cas il est relativement difficile d'adopter ces valeurs comme paramètres. En effet s'il est relativement aisé de déduire ces grandeurs des facteurs de taille des transistors, le contraire est en revanche relativement ardu, et des systèmes tels que GDT [Burich 87] proposent comme solution l'emploi de tableaux donnant les correspondances entre les différentes quantités (ce qui n'est pas simple à mettre en œuvre dans le cas où les cellules de base ne sont pas fixées). Cependant, l'usage des tableaux est relativement

limité: en effet, si ceux-ci décrivent la façon de définir des UAL avec 1, 2, 4, 8 et 16 bits, on ne pourra pas avoir d'UAL 13 bits!
Une autre solution à ce problème a été proposée dans STYX [Rougeaux 87]: elle consiste à mémoriser les différents résultats obtenus lors des appels de générateurs. On cherche alors à chaque nouvel appel d'un générateur si on n'a pas déja rencontré une telle situation. Si c'est le cas on récupère alors le résultat, sinon on le génère et on le mémorise.

## 3.8. Les outils

Un système intégré de conception de circuits tel que NAUTILE doit pouvoir accepter un ensemble d'outils divers et variés. La plupart de ces outils ne sont pas développés dans NAUTILE: il s'agit le plus souvent de s'interfacer avec des outils déja existant [Jerraya 89].

Parmi les outils, on distingue essentiellement:

- des outils de manipulation des formats graphiques et textuels: essentiellement des éditeurs, l'éditeur graphique ne permettant dans la version actuelle du système que la visualisation des cellules
- des outils de simulation logique et électrique travaillant soit au niveau interrupteur, soit au niveau transistor[1]
- des outils de test basés sur des principes tels que l'analyse logique, la génération de vecteurs de test, ou l'observation par microscope électronique.
- des outils de génération, permettant à partir de spécifications données de produire le dessin des masques. Il peut s'agir dans ce cas aussi bien d'outils d'aide à l'assemblage (tels que routeurs, compacteurs ou générateurs de plan de masse), que de générateurs de modules particuliers, tels que des générateurs de PLA (PAOLA [Chuquillanqui 82]) ou même de cellules plus complexes (par exemple un générateur de chemin de données comme APOLLON [Jamier 85]).
- enfin des processeurs d'expansion permettant de générer différents formats de description (LUCIE [Paillotin 85], RNL [Notrott 87], GL1 [Lambert 81], ...).

---

[1] Il est nécessaire de considérer dans cette optique l'existence d'extracteurs permettant de générer à partir du dessin des masques le graphe électrique (au cas où le concepteur n'a pas spécifié de représentation électrique).

## 3.9. Conclusion

Le principe de description d'un circuit en NAUTILE consiste à utiliser une hiérarchie de cellules et de motifs, pouvant être décrits selon de multiples représentations: les motifs de façon explicite, et les cellules par le biais de la vue de construction. Celle-ci mémorise les instances utilisées dans une cellule donnée, et ne dépend ni du contexte d'utilisation, ni des différents types de représentation: les cellules de composition sont donc définies par une hiérarchie unique, appelée l'arbre d'appel. Une vue externe permet de connaitre l'interaction entre une cellule et ses voisines. Enfin, un langage est proposé, permettant d'écrire entre autres des générateurs de module, dont l'évaluation produira soit des cellules, soit des motifs.

# 4. GESTION DES DONNEES DANS NAUTILE

NAUTILE présente de nombreuses originalités dans sa façon de gérer les données: la plupart des concepts introduits l'on été pour des raisons de sureté de conception, et afin d'obtenir des circuits "corrects par construction". Ainsi NAUTILE permet-il de maintenir la cohérence entre différentes vues d'un même circuit (paragraphe 4.1). Il autorise également un certain nombre de changements de la technologie, grace notamment à l'utilisation d'un fichier technologique comportant entre autre des règles de construction (un jeu par vue: paragraphe 4.2). L'interface avec des systèmes externes étant vitale pour un système de conception moderne, NAUTILE propose des mécanismes permettant d'intégrer des outils et des cellules provenant d'autres systèmes (paragraphe 4.3). Enfin, un certain nombre de caractéristiques permettant de s'affranchir des problèmes d'assemblage liés à l'utilisation d'éléments communs à un ensemble de cellules seront précisées (paragraphe 4.4).

## 4.1. Gestion de la cohérence

### 4.1.1. Le problème du maintien de la cohérence

Un problème fondamental en conception des circuits intégrés est de maintenir la correspondance entre les différents aspect d'une même cellule. En effet les concepteurs conçoivent généralement leurs circuits en partant d'une représentation architecturale ou comportementale, afin d'être moins dépendants des niveaux de représentation les plus bas. La conversion de vues abstraites en vues moins abstraites peut être effectué de différentes façons, comme par exemple par l'utilisation d'un compilateur de silicium. Cependant, une fois la conversion opérée, il n'y a plus aucun lien entre les différentes vues.
L'exemple de la figure 4.1 illustre ce problème:

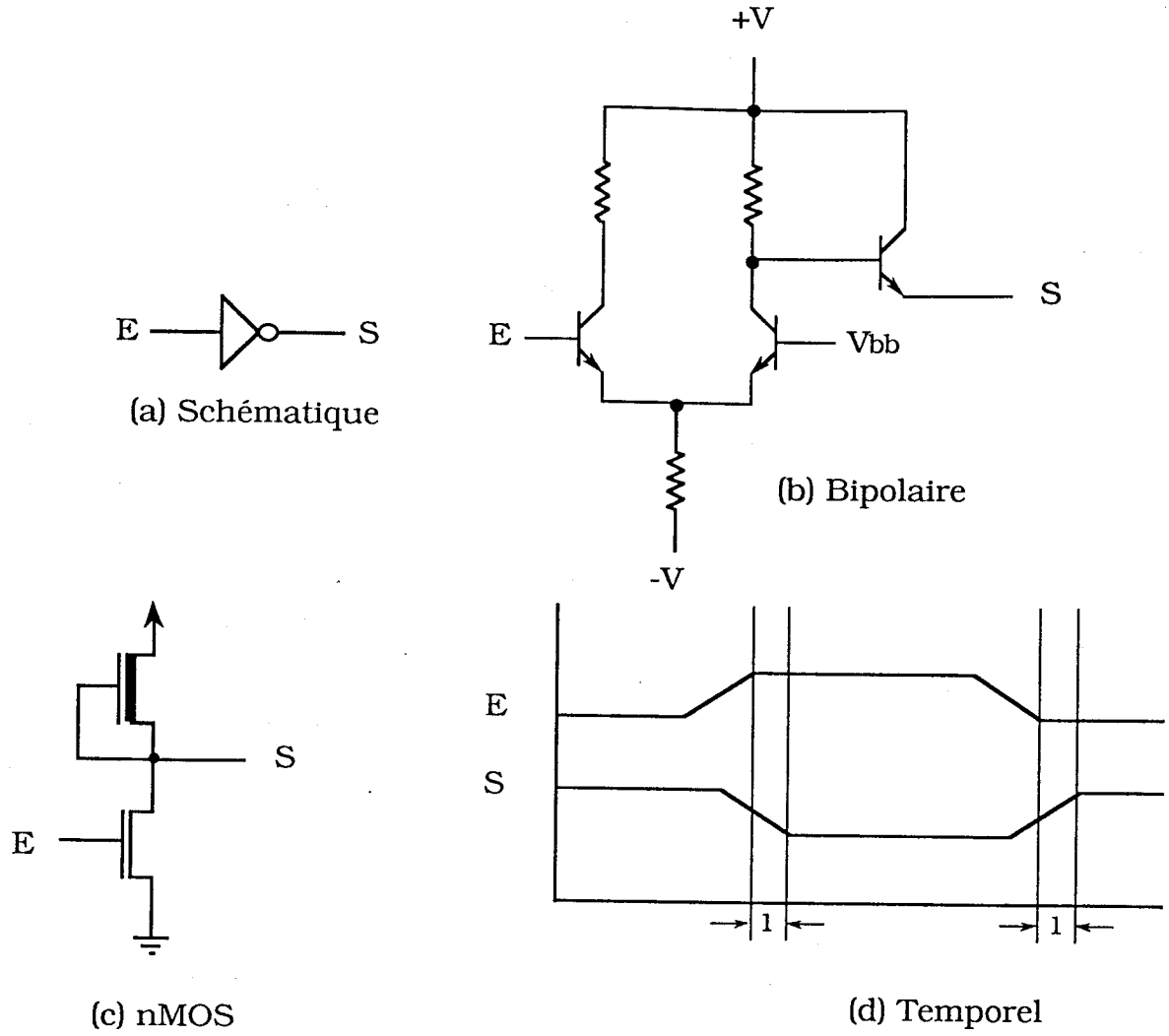



Fig. 4.1: Différentes vues pour un inverseur [Rubin 87]

Il n'est pas trés difficile de convertir la représentation comportementale de l'inverseur en une représentation schématique, en maintenant la correspondance entre les deux. En revanche, la description des transistors comporte de multiples éléments, ce qui introduit des difficultés supplémentaires. Ces difficultés sont essentiellement liées au problème complexe d'assurer une correspondance fonctionnelle entre plusieurs éléments et un seul: les transistors, avec leurs connexions à la masse et à l'alimentation, ainsi que tout leur cablage interne, sont équivalents à un seul composant dans la représentation schématique. Un changement de ce composant schématique va occasionner un changement de tous les composants du transistor correspondant. Mais que va-t-il se passer si un changement est opéré à l'intérieur d'un seul transistor? Il risque de n'y avoir aucun équivalent schématique pour la cellule résultante. Il est donc essentiel qu'un système intégré puisse assurer que deux descriptions données correspondent bien au même objet: il doit pouvoir

maintenir la correspondance entre les différentes représentations d'une cellule [Buchanan 80].

### 4.1.2. Les solutions généralement adoptées

La plupart des systèmes actuels ignorent le problème de la cohérence entre les différentes vues. Soit ils ne proposent aucun moyen de conversion entre les différentes vues, soit ils permettent une telle conversion, mais sans assurer de vérification de cohérence, et encore moins son maintien. Une fois que le circuit a été traduit dans une nouvelle vue, il devient un nouveau circuit, sans rapport avec le circuit d'origine. C'est un problème pour les concepteurs qui souhaitent travailler sur des représentations différentes d'un même circuit, mais sans pour autant violer les spécifications originales de ce circuit.
La solution qui est préconisée par un outil tel que DDS [Knapp 85] [Winslett 89] consiste à conserver suffisamment d'informations sur une vue pour vérifier que le changement d'un de ses composants ne va pas perturber le reste de la cellule. L'inconvénient d'une telle approche est de forcer la mise à plat de certaines portions du circuit dans les cas critiques. Une autre approche a été proposée par Chou et Kim dans le système ORION [Chou 86] et [Chou 88]. Ce système est basé sur des mécanismes de "notification en cas de changement", qui consistent à affecter à chaque composant un indicateur permettant de savoir s'il a été changé ou non. Dans l'affirmative, des messages sont envoyés au concepteur responsable de ces composants. Malheureusement, pour intéressante que soit cette approche, elle oblige l'utilisateur à intervenir pour modifier les composants incriminés, pouvant ainsi remettre en question tout le circuit.
Nous allons maintenant détailler la solution adoptée dans NAUTILE.

### 4.1.3. La cohérence vue par NAUTILE

Dans NAUTILE, la cohérence d'un circuit peut être vue à deux niveaux. En effet, on peut considérer qu'un système intégré doit vérifier:

- d'une part la cohérence entre les différentes vues: si on a en mémoire deux vues d'une même cellule, on veut espérer qu'elles sont bien équivalentes, c'est-à-dire que la réalisation physique de cette cellule aura les caractéristiques des deux vues (cohérence d'une vue électrique avec une vue topologique, par exemple)

- d'autre part la cohérence des appels de cellules: si on crée une instance d'une cellule en respectant les règles d'assemblage, puis que l'on modifie cette cellule, il faut pouvoir être sûr que les règles sont toujours vérifiées, et que les modifications apportées ne viennent pas perturber le reste du circuit. Nous appelons ce type de cohérence la *cohérence hiérarchique.*

### 4.1.4. La cohérence entre les différentes vues

la solution retenue dans NAUTILE pour assurer le maintien de la cohérence entre les différentes vues, a consisté à se doter d'un ensemble de processeurs vérifiant un certain nombre d'invariants (ex: une masse ne peut toucher un fil d'alimentation). Ces invariants sont modélisés sous forme de règles (appelées *règles de construction*), qui sont localisées dans un fichier spécial: *le fichier technologique* (en effet, la plupart de ces règles sont liées à une technologie donnée). Nous reviendrons sur l'utilisation du fichier technologique ainsi que sur les différentes règles de construction au cours du paragraphe 4.2.

Pour assurer ce maintien de la cohérence entre les différentes vues, le système NAUTILE tranche sous certaines conditions: si à un instant donné l'objet incriminé est dans un état cohérent, alors il conviendra de vérifier que les transformations qui lui sont appliquées ne modifient pas cet état.
Ce postulat est décomposé par les vérifications élémentaires suivantes:

**si** les motifs sont corrects par définition
**si** le circuit est décrit hiérarchiquement avec la vue de construction
**si** le concepteur n'a modifié aucune vue physique du circuit
**alors** le circuit est correct par construction

Dans la plupart des autres cas, des outils de vérification classiques sont nécessaires, comme ils le sont pour vérifier la cohérence au niveau des motifs.

De ces différentes contraintes, la première est relativement faible: en effet il est relativement aisé de prouver sur n'importe quelle station de travail un peu évoluée, la correspondance entre les vues physiques d'une cellule simple.

En revanche, la contrainte de n'opérer des modifications que sur la vue construction, jamais sur une vue physique, est une contrainte trés forte pour les concepteurs, qui l'acceptent difficilement. Elle ne peut cependant être contournée en général qu'au prix de longues et fastidieuses vérifications, à moins que l'on ne puisse valider un ensemble de primitives permettant de conserver la cohérence. Dans NAUTILE elle constitue le prix à payer pour une conception sûre.

### 4.1.5. La cohérence hiérarchique

Un exemple de problème de cohérence hiérarchique pourrait être le suivant: supposons qu'une cellule A fasse appel à une cellule B. Que se passe-t-il si une modification de B intervient aprés une modification de A? En effet, en pareil cas, l'environnement de B dans A peut trés bien avoir à changer: rien ne garantit par exemple que l'encombrement de la cellule B soit resté invariant...
Le système doit tolérer ce type de situation afin de conserver une certaine souplesse d'utilisation, mais il faut qu'il les reconnaisse et les signale à l'utilisateur. Aussi le système NAUTILE permet-il de gérer le maintien de la cohérence en pareil cas.

Le mécanisme de maintien de la cohérence hiérarchique est principalement basé sur la chronologie des appels, méthode déjà utilisée par d'autres systèmes, tels que STYX développé dans le cadre du projet SYCOMORE [Rougeaux 87].
Chaque cellule comporte une date de dernière vérification de la cohérence, une date de création, une date de modification, et le nom de celui qui l'a créée (éventuellement ce peut être un outil, ou le système lui-même).
En ce cas, on peut affirmer qu'une vue physique est cohérente avec la vue construction si et seulement si:

-1- la vue physique provient de la vue construction

-2- la vue physique n'a jamais été modifiée

-3- la date de création de la vue physique est postérieure aux dates de modification de la vue construction et de toutes les sous-cellules qu'elle utilise.

A chaque fois que l'on modifie la vue construction d'une cellule, le système NAUTILE effectue les vérifications suivantes:

- une vue physique doit être cohérente avec la vue construction: si la vue physique provient de la vue construction il est nécessaire de la recalculer. En revanche si c'est une vue externe importée par le concepteur, celui-ci devra être averti d'un manque de cohérence à ce niveau.
- une vue physique doit être cohérente avec les autres vues physiques: ceci est un corollaire de la vérification précédente. En effet, pour les cellules, toutes les vues physiques doivent être cohérentes avec la vue construction. Elles seront donc fatalement toutes cohérentes entre elles.
- une cellule doit être cohérente avec toutes ses sous-cellules: c'est typiquement la vérification de la cohérence hiérarchique.

## 4.2. Indépendance technologique

Au cours de la vie d'un circuit, de nombreux changements de technologie dus à l'évolution des procédés de fabrication, sont malheureusement à déplorer. Aussi les systèmes récents cherchent-ils à minimiser les modifications que le concepteur doit apporter à son circuit lorsqu'un changement de technologie survient, l'idéal étant qu'il ne soit nécessaire de procéder à aucune modification du tout.
Pour arriver à ce résultat, il faut pouvoir paramétrer la technologie, un changement de technologie ne se traduisant alors que par une modification de la valeur de quelques paramètres , ne remettant pas en cause les outils et les cellules existants.

### 4.2.1. Le fichier technologique

Dans NAUTILE, l'indépendance vis-à-vis de la technologie est assurée par un fichier technologique, constitué:

- d'un ensemble de constantes (espacement, largeur, recouvrement,...).
- de règles de construction définissant les contraintes d'assemblage des cellules entre elles, ces règles étant partitionnées par domaine d'application (un jeu de règles par vue physique).
- d'un ensemble de motifs de base (un transistor est définit de telle manière, en utilisant tels niveaux, tels paramètres, etc...).

L'ensemble des motifs de base d'une technologie minimale comporte le transistor, le fil et le contact, bien qu'en réalité les motifs de base nécessaires à la conception de circuits soient plus variés et complexes.

### 4.2.2. Les règles de construction

Un principe bien connu des concepteurs aussi bien de logiciels que de circuits, affirme que plus une erreur est détectée tardivement dans la phase de conception, plus la correction de cette erreur coutera cher.
A cette fin, on peut soit se doter d'un ensemble de mécanismes permettant de détecter les erreurs de conception le plus tôt possible, soit imposer l'utilisation de primitives "sures", ne permettant pas d'introduire d'erreurs de conception: c'est cette solution qui a été retenue pour NAUTILE.
A cet effet, NAUTILE propose un jeu de primitives d'assemblage, qui permettent à l'utilisateur de s'affranchir de la plupart des vérifications.
Ces primitives utilisent un ensemble de règles, qui permettent d'assurer que l'assemblage obtenu est *correct par construction*. On utilisera ainsi des règles d'assemblage topologique pour éviter les vérifications de règles de dessin (DRC), et des règles d'assemblage électrique pour éviter les vérifications de règles électriques (ERC).
La figure 4.2 présente le mode de fonctionnement des primitives d'assemblage:

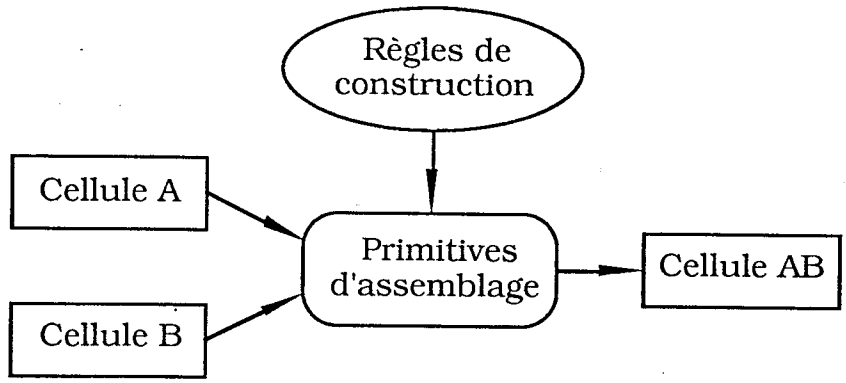



Fig. 4.2: Utilisation des règles de construction

Lorsque l'on souhaite assembler deux cellules entre elles (l'assemblage est pour l'instant limité à deux entités), il suffit d'indiquer le nom de chacune des cellules à assembler, ainsi que l'outil d'assemblage que l'on souhaite utiliser. Aucune vérification n'est alors effectuée. C'est seulement lors de l'utilisation de la cellule résultante que l'outil prévu pour l'assemblage va réellement *construire* l'assemblage, et ce en fonction du contexte courant.

Il utilisera alors le jeu de règles défini par le contexte pour achever cet assemblage, et uniquement ce jeu: cela permet de ne pas effectuer toutes les vérifications relatives à l'assemblage à la fois.

Exemple: pour l'exemple de la figure 4.2, supposont que l'on souhaite visualiser dans le mode électrique la cellule AB obtenue par assemblage de A et de B. Le système sait alors que l'outil utilisé pour cet assemblage est par exemple le routeur ("rout"). Il va donc faire appel à "rout" pour opérer l'assemblage, et c'est ce routeur qui va vérifier avant de procéder à l'assemblage que les types des connecteurs à relier sont bien cohérents (électriquement parlant). Enfin, le résultat du routage en question sera passé comme paramètre au processeur d'affichage qui se chargera de la représentation graphique à l'écran.

Remarque:

S'il est relativement simple de modéliser des règles de construction topologiques, et un peu moins simple pour des règles électriques, il est en revanche beaucoup plus ardu de définir de façon formelle des règles fonctionnelles, ou encore des règles ayant trait aux caractéristiques et au comportement des circuits: contraintes de délais, de vitesse, de consommation, de surface, etc...
Une solution heuristique a été proposée par Knapp & Al. [Knapp 83].
L'ensemble des règles disponibles dans NAUTILE est donc pour l'instant limité à des règles géométriques et électriques.

#### 4.2.2.1. Les règles topologiques de construction:

Ces règles, qui relèvent des vérifications élémentaires de validité du dessin des masques, seront appliquées par les différents outils de dessin, de compactage et d'assemblage.
Elles concernent esssentiellement les erreurs locales, et vérifient surtout le non recouvrement des frontières lors de l'assemblage (excepté dans le cas de l'utilisation d'éléments partagés, comme décrit au paragraphe 4.4).
Ainsi, lors d'un assemblage automatique, l'outil d'assemblage va utiliser les règles topologiques de construction pour placer la ou les cellules, alors qu'au cours d'une instanciation, ces mêmes règles serviront à prévenir l'utilisateur en cas de placement illicite.

#### 4.2.2.2. Les règles de construction électriques:

Le but essentiel de ces règles est de garantir que l'interprétation électrique de l'assemblage génèrera une vue électrique correcte, sans pour autant qu'elles soient efficaces à 100%. Elles ne remplaceront donc pas une vérification complète à l'aide d'un vérificateur de règles électriques: des erreurs assez délicates comme des boucles illégitimes sont difficiles à détecter et réclament une simulation une fois le circuit terminé.
En fait la plupart de ces vérifications consistent en des vérifications élémentaires de cohérence: elles sont basées sur les propriétés des connecteurs. En effet ceux-ci possèdent des informations telles que leur type (entrée, sortie, entrée/sortie, horloge, alimentation, masse, etc ...), leur direction, leur niveau, etc... Toutes ces informations sont optionnelles, et peuvent être fournies à n'importe quel moment de la conception. Néanmoins, le circuit ne pourra être déclaré dans un état correct que si toutes ces informations sont présentes (certaines d'entre elles pouvant être fixées par des outils, tels que le routeur).
Les vérifications effectuées porteront donc essentiellement sur la cohérence entre ces différentes informations: les connecteurs d'une même équipotentielle doivent être de types compatibles, de même direction (mais éventuellement de sens contraire), et de niveaux compatibles (il peut alors s'avérer nécessaire de rajouter un via).
Ainsi dans l'exemple de la figure 4.3 l'assemblage ne pourra se faire, car relier les deux connecteurs les plus au nord reviendrait à faire un court-circuit entre l'alimentation et la masse:

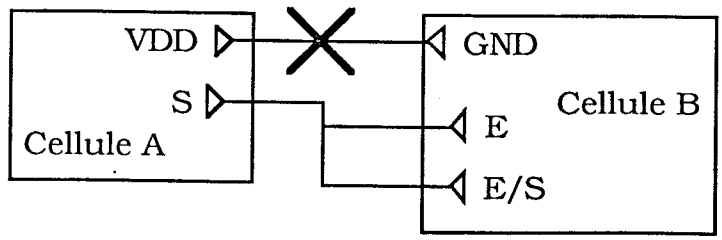



Fig. 4.3: Exemple d'assemblage

Des exemples de règles électriques élémentaires sont donnés ci-dessous, d'autres exemples pouvant être trouvés dans l'annexe 4:

Règle 1: Il ne doit exister aucun chemin direct reliant l'alimentation à la masse, ce qui prévient les court-circuits entre VDD et GND.

Règle 2: Il ne doit pas exister un seul transistor entre la masse et l'alimentation (sinon lorsque ce transistor serait passant il y aurait un court-circuit entre VDD et GND).

Règle 3: Tous les transistors doivent avoir au moins une possibilité de connexion.

Règle 4: La grille d'un transistor de transmission ne peut être connectée qu'à la sortie d'une porte logique à restauration: il est interdit de commander un transistor de transmission à l'aide d'un autre transistor de transmission.
Cette règle provient du fait que le niveau d'un signal se dégrade lorsqu'il traverse un transistor de transmission. Cette dégradation s'accentue lorsque la sortie du dit transistor commande un autre transistor de transmission, ceci pouvant altérer suffisamment le signal pour qu'il devienne non significatif si on répète cette opération plusieurs fois.

### 4.2.3. Le changement de technologie

En cas de changement de technologie, un certain nombre d'éléments du système devront changer. Nous allons donc maintenant détailler élément par élément les changements qui devront être introduits, du fait d'une nouvelle technologie.

Les motifs: si les changements sont minimes (par exemple pour passer d'une technologie Cmos à une technologie Cmos assez proche), seuls quelques paramètres devront être changés (tels que certaines règles de dessin et constantes localisées dans le fichier technologique) et éventuellement quelques motifs (tels que les motifs de base).
Dans ce cas les motifs utilisateurs peuvent aussi avoir à être redéfinis par le concepteur. Quant aux motifs produits par des outils, il suffira de relancer l'exécution de l'outil les ayant générés pour les mettre à jour.
Par contre, en cas de changements majeurs, il sera nécessaire de redéfinir tous les motifs.

Les outils: normalement les outils sont paramétrés par des constantes technologiques permettant des modifications minimes en cas de changements simples de la technologie (c'est le cas notamment des

routeurs). Dans le cas contraire, il pourra être nécessaire de réécrire tout ou partie des outils.

Les cellules: si elles sont définies par leur vue de construction, elles ne changeront pas avec la technologie. En revanche, si elles sont décrites dans une vue physique quelconque de façon manuelle, cette vue deviendra trés certainement obsolète.

Les générateurs: en général ils utilisent une liste de motifs assemblés de façon à obtenir des structures régulières.
Dans leur cas, si la liste de motifs a été mise à jour, et si l'assemblage est paramétré par les constantes technologiques, alors il ne sera pas nécessaire de les écrire de nouveau.

### 4.3. Relation avec les systèmes externes

Une entreprise achetant un nouveau système d'aide à la conception de circuits, aura souvent un passé conséquent derrière elle. Ceci se traduira par des bibliothèques de cellules déja décrites à l'aide d'autres systèmes, ainsi que par des outils déja existants. Il est donc indispensable que le concepteur ait toujours à sa disposition ces cellules, et puisse utiliser les outils qui lui sont habituels. Pour cela le nouveau système doit comporter des interfaces lui permettant:

- la traduction automatique des anciennes cellules dans le nouveau système et vice-versa

- l'utilisation de hiérarchies entières de cellules sous forme de squelettes, puis la génération de l'ensemble du circuit (comportant les anciennes cellules ainsi que les nouvelles) sous forme compréhensible par l'ancien système (ceci permettant de conserver opérationnels les outils disponibles avec ce système).

#### 4.3.1. Utilisation de processeurs d'interface

L'un des points forts du système NAUTILE est de pouvoir manipuler des données issues de n'importe quel système externe: une cellule peut avoir ses différentes vues décrites dans n'importe quel système (y compris le système NAUTILE!). La seule limitation est que tous les motifs utilisés pour

sa construction soient définis dans toutes les vues qu'elle possède. Chaque vue peut alors être expansée dans une représentation externe utilisant à chaque fois un processeur d'expansion spécialisé (voir les figures 3.7 et 4.4):

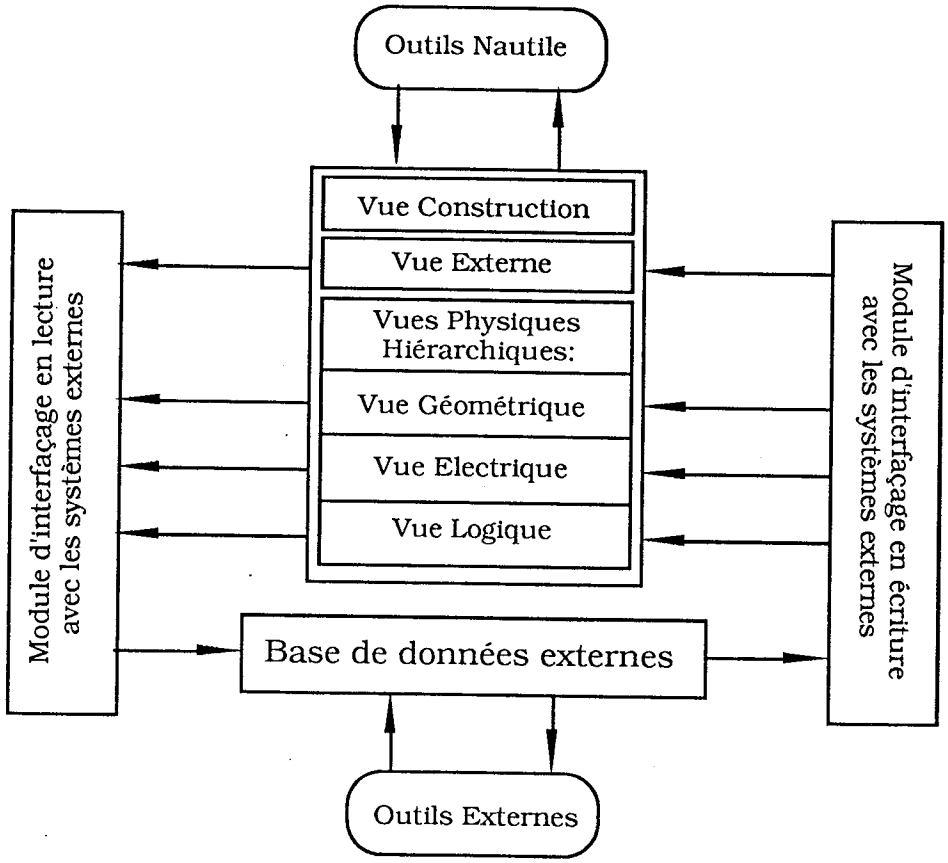



Fig. 4.4: Interfaçage de NAUTILE avec un système externe

La figure 4.4 permet de préciser le mode de fonctionnement de NAUTILE, vis-à-vis des systèmes externes. On voit sur cette figure que NAUTILE utilise un certain nombre de processeurs spécialisés, pour traduire la vue construction d'une cellule en des vues externes pouvant être reprises dans d'autres systèmes. Les outils disponibles dans ce système peuvent alors être utilisés pour éventuellement enrichir les descriptions ainsi obtenues.

### 4.3.2. Mécanismes d'expansion dans une vue externe

La génération de vues dans un système externe est étroitement liée aux possibilités du système externe en question.
En fait, les paramètres décisifs sont:

- la possibilité d'avoir des appels de cellules externes
- le support de la hiérarchie

On peut ainsi distinguer trois cas différents, selon la façon dont le système externe gère la hiérarchie, et les cellules externes. Supposons ainsi que l'on souhaite traduire la cellule "titi" écrite dans NAUTILE dans un système externe "ext":

-1- le système "ext" supporte la hiérarchie et les appels de cellules externes: on se trouve alors dans un cas idéal (c'est par exemple le cas avec les systèmes LUCIE et RNL). La façon de traduire la cellule "titi" est alors on ne peut plus simple: NAUTILE traduit directement le squelette de "titi" dans le système externe, en remplaçant les appels de motifs par des appels externes à des cellules écrites dans "ext". Dans ce cas, le système NAUTILE ne manipule que des noms de fichiers, et des vues environnement pour chaque motif.

-2- le système "ext" ne supporte pas la hiérarchie: dans ce cas NAUTILE "met à plat" la cellule "titi" (c'est-à-dire qu'il parcourt la hiérarchie complète, afin de connaitre la position de chaque motif dans la cellule "titi"). Le résultat est alors envoyé au système "ext".

-3- le système "ext" ne supporte pas l'appel de cellules externes (c'est le cas du GL1, par exemple). Ici, deux solutions auraient pu être envisagées: soit on aurait pu récupérer tous les motifs utilisés pour la cellule "titi" et les analyser de façon à pouvoir les inclure dans la cellule finale, soit on traduit directement les motifs en NIL, et on fournit un mécanisme permettant d'obtenir une cellule dans le système "ext" à partir d'une cellule NIL. C'est la seconde solution qui a été adoptée (bien que ce choix soit criticable), car c'est celle qui nous a parue la plus simple à mettre en œuvre. On utilise ici de véritables "traducteurs", permettant de décrire *toutes* les primitives du système externe en NIL (alors que dans le premier cas, seules les primitives d'assemblage sont à traduire).

On peut en outre remarquer que le langage NIL correspond au premier cas de figure: il existe ainsi des possibilités de translation de cellules décrites en NIL dans la structure interne NAUTILE.

Enfin, un cas un peu à part est celui de systèmes tels que ASTAP. En effet, ASTAP supporte l'appel à des cellules externes, *à condition que celles-ci soient présentes en mémoire centrale.* Dans ce cas, NAUTILE ne gère donc pas de noms de fichiers, mais il se contente d'espérer que le

concepteur aura bien chargé tous les motifs ASTAP qui lui sont nécessaires pour effectuer l'assemblage avant de simuler.

### 4.3.3. Ajout d'une nouvelle vue externe

L'un des atouts majeurs de NAUTILE est de permettre aisément la création d'une nouvelle vue, telle par exemple qu'une vue spéciale utilisée pour une simulation particulière.

En pareil cas les objets à redéfinir seront alors:

- les motifs devant être utilisés pour cette vue particulière
- les règles de construction définies dans le fichier technologique
- une fonction "charge" permettant d'importer une cellule décrite dans cette vue
- une fonction "génère" qui produit à partir de la vue de construction d'une cellule sa représentation dans la nouvelle vue. Cette fonction est en fait une méthode qui doit être applicable à toutes les primitives de construction de base. Ainsi il sera nécessaire de définir par exemple les primitives suivantes:

```
{instance}:génère
{direct}:génère
{dev}:génère
etc...
```

Les primitives "génère" de l'exemple précédent sont appelées des *méthodes* (voir paragraphe 5.1.1). Ces méthodes peuvent s'appliquer différemment suivant leur contexte d'appel.

Enfin, il conviendra éventuellement de modifier également les outils devant manipuler cette nouvelle vue.

## 4.4. Le problème des frontières: la transparence, les partagés

Du fait de la représentation hiérarchique d'un circuit, celui-ci utilisant des cellules qui en appellent d'autres et ainsi de suite, il apparait nécessaire de ne pas représenter une cellule dans son intégralité lorsqu'on se trouve à un niveau élevé de la hiérarchie. En fait, il est même préférable d'adopter une représentation minimale de la cellule, correspondant à un seul niveau de hiérarchie.

Cette représentation se fait naturellement en indiquant uniquement les limites de la cellule: c'est la notion de *frontière.*

### 4.4.1. Les frontières

La notion de frontière, inhérente à celle de vue topologique, permet de ne représenter d'une cellule que ses contours extérieurs, tout en ignorant parfaitement de quoi est composée cette cellule, les seules liaisons permises étant les connexions avec les cellules voisines.
La façon la plus naturelle de représenter la frontière d'une cellule, consisterait à la délimiter par un polygone dont les arêtes forment toutes des angles multiples de 90° entre eux (de tels polygones sont dits "Manhattan"). Une telle représentation es donnée dans la figure 4.5:

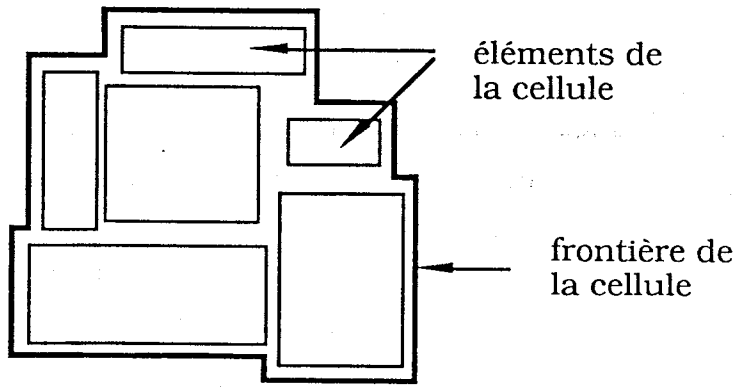



Fig. 4.5: Représentation de la frontière sous forme Manhattan

Cependant, pour enrichissante que soit ce type de représentation, il n'en est pas moins générateur de problèmes extrêmement complexes à résoudre. C'est notamment le cas lors de l'assemblage de cellules, les algorithmes devant alors être mis en jeu réclamant des temps de calcul inconciliables avec une approche interactive de la conception des circuits.
Aussi, afin de simplifier ces algorithmes, la solution communément retenue par les systèmes classiques est-elle de circonscrire la cellule dans un rectangle, appelé boite enveloppante de la cellule. Une telle configuration est décrite par la figure 4.6:

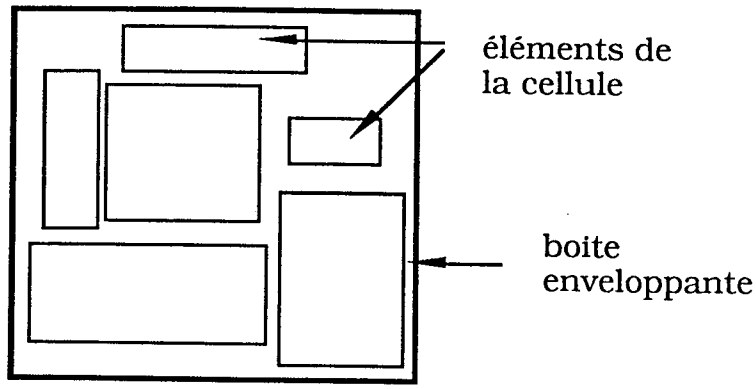



Fig. 4.6: Représentation de la cellule sous forme d'un rectangle

Cette solution, qui constitue une simplification de la précédente, est celle qui est adoptée par NAUTILE dans un premier temps: en effet elle facilite la réalisation tout en permettant un gain important au niveau des temps de traitement, si on interdit le recouvrement des cellules entre elles. De plus, adopter une telle solution permet de simplifier la vérification hiérarchique des règles de dessin: en effet, une fois qu'une cellule est validée, il n'est pas nécessaire de l'examiner de nouveau. Et si on a pris soin de choisir des boites enveloppantes appropriées, il est seulement nécessaire de vérifier les règles entre les différentes boites (les règles sont alors uniquement des règles de non recouvrement).
En revanche, elle impose des pertes importantes de surface.
Aussi, afin de pallier aux inconvénients d'une représentation par trop simpliste, ou bien au contraire par trop complexe, nous avons choisi d'introduire une possibilité suplémentaire, permettant de bénéficier des avantages des deux solutions, sans leurs inconvénients. L'utilisation de formes spéciales appelées des fantomes, permettant à deux cellules de se chevaucher constitue cette solution. La figure 4.7 montre un exemple de définition de fantomes:

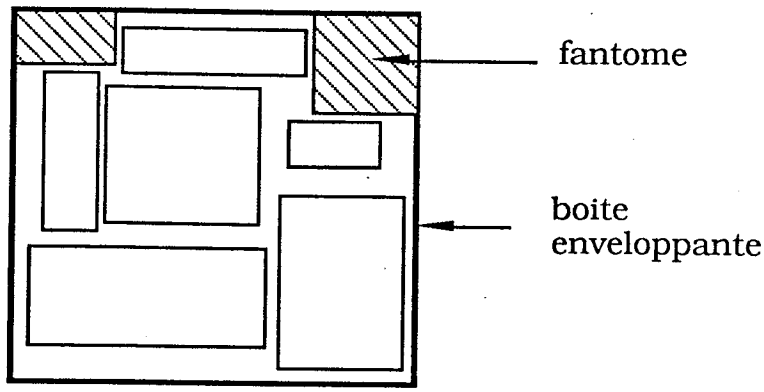
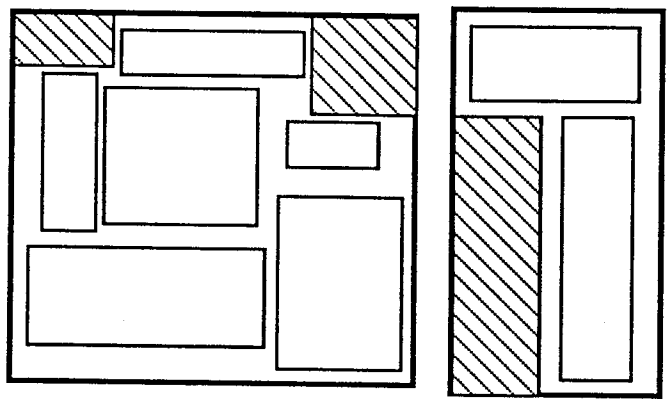
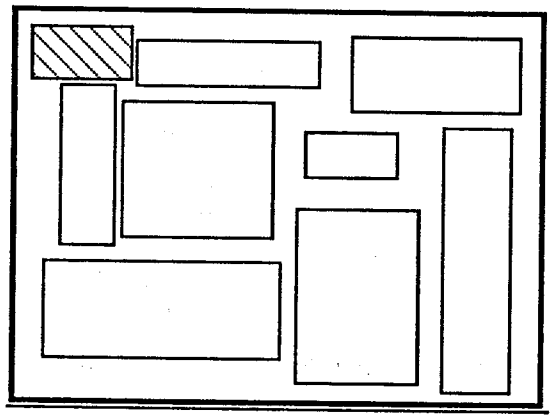



Fig. 4.7: Déclaration de fantomes

On voit sur cet exemple que le fantome constitue un affinement de la boite enveloppante, sans atteindre la complexité d'une frontière de type Manhattan.
La figure 4.8 permet de comprendre le schéma de fonctionnement des fantomes:



(a) Les cellules avant assemblage



(b) La cellule résultant de l'assemblage

Fig. 4.8: Utilisation des fantomes

Dans la figure 4.8 (a) on a deux cellules possédant chacune leurs fantomes respectifs. L'assemblage de ces deux cellules permet d'utiliser les fantomes afin d'optimiser le placement des différents blocs.

Nous reviendrons sur la définition des fantomes dans le paragraphe 4.4.3.

### 4.4.2. Notions de partagés et de fantomes

Afin de faciliter la tache des outils opérant sur le dessin des circuits, les frontières des cellules sont arbitrairement simplifiées: la plupart du temps elles se réduisent même à leur plus simple expression, un rectangle. Afin de pallier à cette représentation par trop restrictive, les

concepteurs de systèmes de dessin ont pensé définir des "zones d'utilisation" des cellules dessinées.

Diverses façons de procéder sont envisageables:

- soit on définit sur la cellule des zones protégées, appelées "cadres de protection" (Protection Frames en anglais) consistant généralement en des polygones inscrivant ces zones, étant donné qu'un polygone correspond à un niveau technologique. Ces cadres de protection s'accompagnent alors de cadres de connexion ("Terminal Frames"), rectangles délimitant les zones où peuvent se trouver les connecteurs (l'intérêt de l'utilisation de ces cadres de connexion étant de pouvoir préciser non seulement des valeurs inférieures de largeur des connecteurs, mais également des valeurs supérieures, ce qui permet une plus grande souplesse d'emploi lors de l'assemblage). Cela permet également de définir des contraintes imposées par l'utilisateur, ainsi que d'introduire la notion d'équivalence électrique: des connecteurs équivalents sont des connecteurs appartenant à un même ensemble de cadres de connexion.

Cette approche est celle adoptée par des systèmes comme SPARCS [Burns 86], SQUID [Keller 82], et PYTHON [Bales 82].

- soit on utilise des "ombres de transparence", indiquant les zones non protégées. Cette notion permet notamment d'améliorer la gestion des connecteurs.

C'est l'approche choisie pour STYX [Jerraya 85], et également pour le système NAUTILE (les "ombres de transparence" sont appelés ici des fantomes, afin de ne pas opérer de confusion avec la définition qui est généralement admise pour les ombres, et que l'on trouve dans [Wolf 87]).

- soit on utilise des "ombres" telles que définies dans [Wolf 87]: ces ombres permettent de définir la façon dont un objet interagit avec ses voisins. La figure 4.9 présente le principe des ombres tel que définit par W. Wolf:

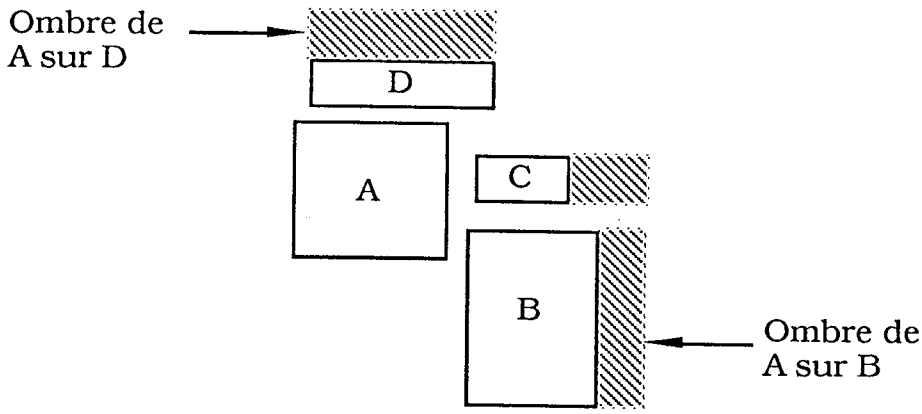



Fig. 4.9: Principe des ombres

On voit sur cette figure le principe de fonctionnement des ombres: A porte sur B, C et D des ombres, dans lesquelles on peut placer un nouvel élément E. On sera alors certain que E ne viole aucune règle de dessin, par rapport à A.

- soit enfin, on utilise des niveaux spéciaux permettant de traverser des cellules (notamment pour le passage de bus) comme dans le système IDS de la société CADENCE [Law 87].

### 4.4.3. Les fantomes - Les prolongés

Les fantomes sont des éléments attachés à une cellule, et permettant à une autre cellule de recouvrir celle-ci à l'endroit où passe ce fantome.
Les fantomes définissent donc à l'intérieur d'une cellule une "zone permise", où le recouvrement par une autre cellule est autorisé.

Exemple de définition de fantomes :

Pour construire un ensemble de cellules on s'aperçoit que l'on a besoin de faire passer deux bus du niveau Alu dans ces cellules : on pourra créer une cellule contenant ces bus de niveau Alu sous forme de fantome (grâce à des segments dits prolongés de niveau technologique Fantome_Alu) et on l'appellera dans la définition des autres cellules.

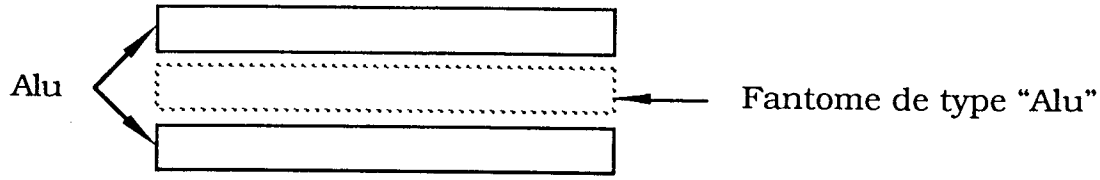



Fig. 4.10: Exemple d'un fantome de type "Alu"

L'exemple de la figure 4.10 montre bien l'intérêt de se définir plusieurs types de fantomes, suivant leur appartenance à une couche technologique ou à une autre.
En effet, les règles de dessin ne sont pas forcément les mêmes suivant le niveau technologique envisagé: dans l'exemple de la figure 4.11, il est clair que les fantomes de type "Alu" et les fantomes de type autre que "Alu" devront être différents:

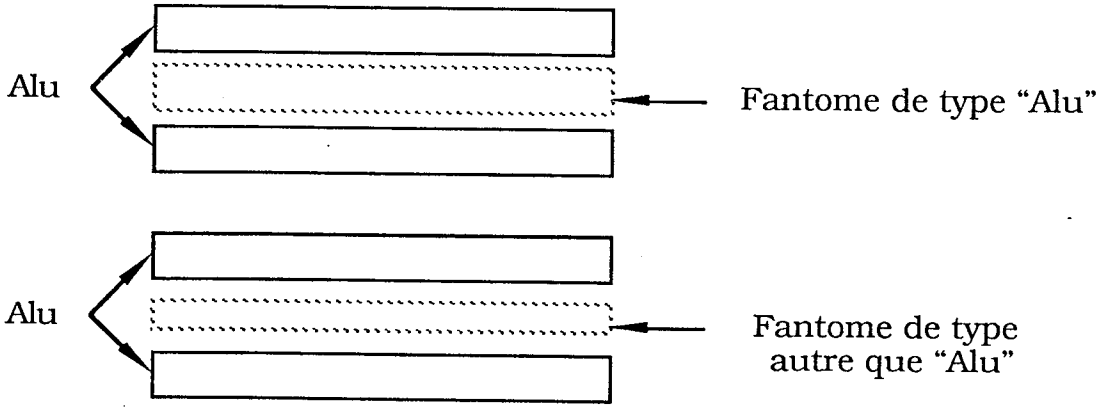



Fig. 4.11: Différents types de fantomes

### 4.4.4. L'utilisation des fantomes

L'utilisation des fantomes n'est possible qu'avec une déclaration préalable, permettant un gain de temps et de place. Un certain nombre de fantomes seront donc prédéfinis, notamment des fantomes permettant le passage de fils de métal (bus, alimentation et masse).

Problème d'héritage d'un fantome:

Le fantome d'une cellule A appelant une cellule B ne peut "recouvrir" B que s'il existe à l'intérieur de celle-ci un fantome correspondant, ou éventuellement du vide. De toute façon, il pourrait être intéressant de se définir systématiquement un fantome associé à une

cellule donnée, afin de ne pas se poser la question d'héritage des fantomes.
On pourrait alors avoir:

$$fantome_i = vide_i + \bigcup_{j=1}^{i-1}(fantome_j) - \bigcap_{j=1}^{i-1}(fantome_j)$$

Problème de modification d'une cellule:

Le problème se pose de savoir quelle stratégie adopter lorsque l'on souhaite modifier une cellule.
En effet, en pareil cas, que se passe-t-il pour le fantome associé à cette cellule? Doit-on le modifier lui aussi, ou bien faut-il créer un nouveau fantome, ou encore en utiliser un déja existant?
Dans la version actuelle de NAUTILE, c'est à l'utilisateur de préciser la manière dont le choix sera effectué.

### 4.4.5. Les éléments partagés

Ce type d'éléments sert en fait à pallier aux inconvénients d'une représentation externe des cellules à l'aide d'une boite enveloppante de forme rectangulaire.
En effet, il est alors impossible de décrire une situation pour laquelle un élément appartient à la fois à deux cellules: c'est le cas classique du contact, qui opère précisément une jonction non aboutive entre deux objets:

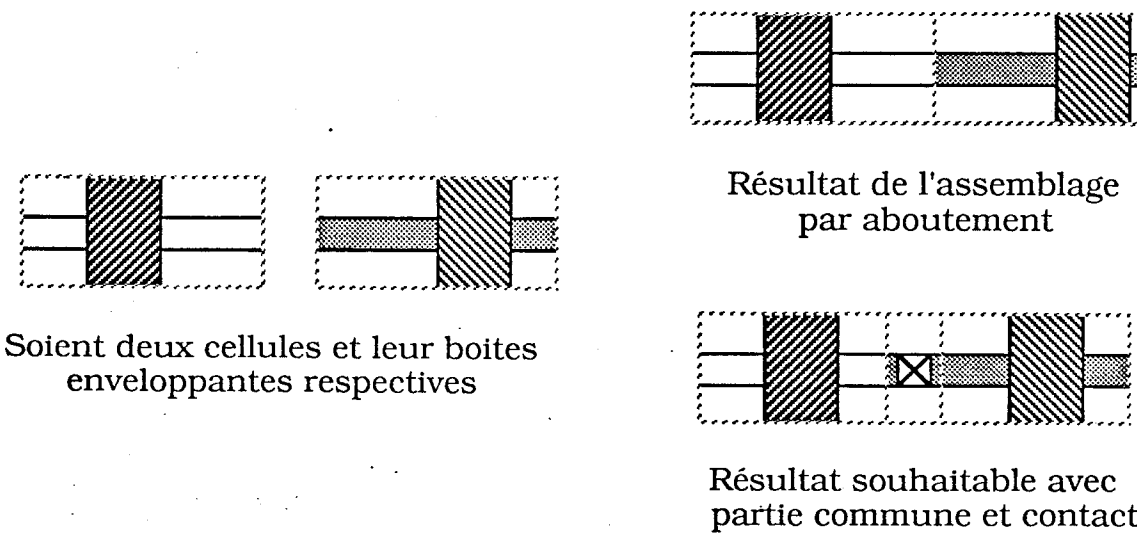
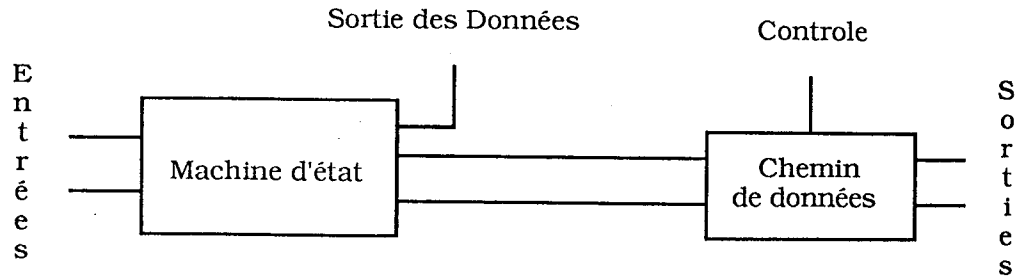


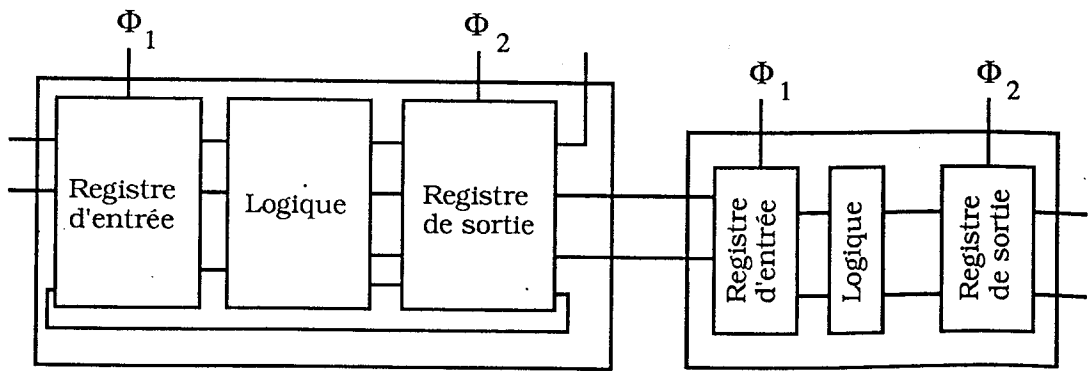

Fig. 4.12: Un contact partagé

Lorsqu'un concepteur se trouve devant un cas tel que celui présenté dans la figure 4.12, il définit généralement des portions de cellule: dans l'exemple en question, il s'agirait de moitiés de contact. Ceci pose malheureusement des problèmes de vérification, notamment de la fonctionnalité: on ne peut en effet pas dire qu'un demi-contact possède une fonctionnalité en soi (si tant est qu'un contact entier en possède une!). C'est uniquement en conjonction avec un autre demi-contact qu'il aura une signification. Une situation assez génante peut d'ailleurs se produire si lors de l'assemblage le demi-contact en question se retrouve sans vis-à-vis: on se trouvera alors immanquablement dans une situation de violation de toutes les règles.

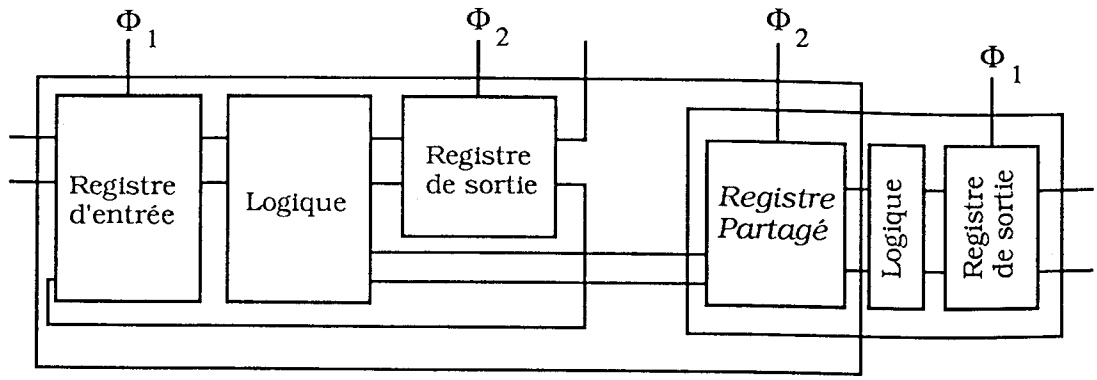
Cependant, le problème des éléments partagés ne se pose pas que dans le cas d'éléments géométriques.
L'exemple de la figure 4.13 décrit un cas d'éléments partagés dans un contexte logique:




(a) Description sous forme de chemin de données




(b) Description sous forme de registres (vue logique)




(c) Même type de description, mais avec un registre partagé

Fig. 4.13: Exemple d'un registre partagé [Brown 83]

Ici, c'est un registre qui est partagé par deux cellules indépendantes l'une de l'autre, afin de minimiser le cout de chaque cellule.

NAUTILE autorise donc la déclaration d'éléments partageables (dans l'exemple de la figure 4.12 ce serait le contact et la zone l'entourant), permettant de résoudre ce genre de problème: un élément de cellule déclaré comme type partagé permettra à une autre cellule d'utiliser cet élément.
Ainsi, une cellule pourra recouvrir partiellement (ou totalement) une autre cellule, si et seulement si l'intersection des deux cellules donne un ensemble d'éléments partagés.

Limitation:
Il convient cependant de remarquer que malgré sa puissance intrinsèque, le concept d'éléments partagés peut s'avérer relativement complexe à gérer dans le cas d'algorithmes classiques, tels que des algorithmes de vérification de règles de dessin. Il est ainsi nécessaire de choisir avec précaution quelle stratégie adopter: soit limiter l'emploi des éléments partagés (ce qui revient à perdre en surface, mais à gagner en efficacité), soit au contraire privilégier les améliorations de la surface par l'utilisation des éléments partagés, mais au détriment de la rapidité d'exécution des algorithmes de parcours de la hiérarchie.

## 4.5. Le concept d'alias

L'introduction des alias est due à la nécessité de pouvoir définir pour une vue donnée de la cellule plusieurs représentations dans une autre vue pour cette même cellule.
Ainsi, à une vue électrique peuvent correspondre plusieurs vues géométriques, et à une vue logique plusieurs vues électriques.
Cependant, la question se pose alors de savoir s'il faut conserver toutes ces représentations, ou bien s'il faut en choisir une particulière, et dans ce cas, sur quels critères? Est-on obligé de définir autant de cellules différentes qu'il existe de vues différentes d'un même type?

NAUTILE contourne le problème en autorisant dans ce cas l'usage des "alias". On peut rapprocher la notion d'alias de celles des "versions" (des améliorations ou des corrections d'un objet), et des "alternatives" (différentes possibilités de réalisation d'un objet) définies notamment par R. Katz [Katz 86].
On défini l'alias d'une cellule A comme étant une cellule B, dont toutes les vues exceptée une seule, pointent sur les vues de la cellule A.

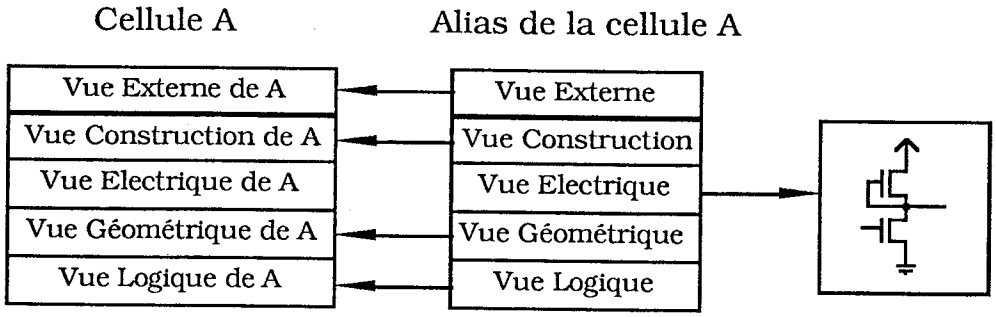



Fig. 4.14: Alias d'une cellule

Les alias permettent à l'utilisateur de se donner des visions différentes d'une même cellule, comme par exemple une version construite à l'aide des différents outils offerts par le système, et une version dessinée manuellement.
Cependant le système ne permet aucune gestion automatique des alias, et notamment il ne produit aucune vérification de fonctionnalité entre différents alias. Ces vérifications sont laissées à la charge de l'utilisateur. C'est ce qui différencie les notions d'alias et d'alternative: la gestion des alias telle que proposée dans NAUTILE est grandement simplifiée en comparaison de ce que propose R. Katz.

# 5. REALISATION, EVALUATION, RESULTATS

Les concepts introduits par NAUTILE ont été validés par la conception d'un prototype écrit dans une extension orientée objet du langage Lisp (voir paragraphe 5.4).
La version actuelle comporte l'ébauche d'une structure de données orientée objet (paragraphes 5.1 et 5.2), manipulée par l'intermédiaire du langage NIL (paragraphe 5.3).
Bien que cette version soit encore trés limitée, elle nous a tout au moins permis d'obtenir certains résultats, dont la conception de la partie controle du microprocesseur généré par SYCO (paragraphe 5.7).
Elle a également permis des comparaisons avec les systèmes classiques mettant ainsi l'accent sur les avantages, mais également les limitations du système NAUTILE (paragraphe 5.6).

## 5.1. La structure de données

R.H. Katz [Katz86] a défini les critères suivants pour une base de données VLSI:

- structuration des données
- hiérarchie
- conception orientée objet
- possibilité de description suivant plusieurs "perspectives"
- cohérence des différentes représentations et versions

La plupart de ces concepts ont été repris et développés durant la conception de la structure de données NAUTILE.

Ainsi, tout le système est basé sur l'utilisation d'une structure de données permettant une description hiérarchique et multi-vues des cellules. Cette structure de données est identique pour toutes les cellules, ainsi que toutes leurs vues. Une telle unicité permet une plus grande cohérence entre les différentes représentations d'une cellule, ainsi qu'une meilleure utilisation des outils, ceux-ci travaillant tous sur une structure unifiée (se reporter au paragraphe 4.1 pour la gestion de la cohérence entre les différentes vue d'une cellule).
Enfin, dans un souci de clarté et afin de faciliter les extensions futures du système, nous avons choisi de réaliser cette structure dans un formalisme orienté objet.

Nous allons maintenant examiner les différents composants de cette structure, aprés avoir rappelé quelques notions de programmation orientée objet.

### 5.1.1. Rappels sur la programmation orientée objet

Dans un système orienté objet, toutes les entités sont modélisées par des *objets*. Un objet est constitué d'une portion de mémoire, qui mémorise l'état de l'objet, cette portion de mémoire étant constituée de valeurs appelées *attributs* de l'objet.
Le comportement d'un objet est encapsulé dans des *méthodes*, qui sont des portions de codes qui manipulent ou retournent l'état de l'objet. Les méthodes attachées à un objet font partie de la définition de cet objet, mais elle ne sont pas accessibles en dehors de lui: les objets communiquent entre eux par l'intermédiaire de *messages*, qui constituent donc les interfaces de ces objets. A chaque message compris par un objet correspond une méthode qui applique le message. Un objet réagit à l'envoi d'un message en exécutant la méthode correspondante, et en retournant un objet.
Afin de réduire la taille des informations nécessaires pour les différents objets, ceux-ci sont regroupés en *classes*. Tous les objets appartenant à une même classe partagent attributs et méthodes, répondent aux mêmes messages, et sont appelés *instances* de la classe.
Les classes peuvent être divisées en *sous-classes*, qui sont des raffinements de la classe dont elles sont issues, celle-ci étant alors appelée *super-classe*. En plus des messages propres à leur classe, les objets comprennent tous les messages spécifiques des classes supérieures: on dit qu'ils *héritent* des propriétés et des méthodes de leurs super-classes.

Pour plus de détail la programmation orientée objet, le lecteur pourra se reporter entre autre à l'ouvrage de A. Golberg sur SMALLTALK [Goldberg 83].

### 5.1.2. La programmation orientée objet appliquée à NAUTILE

La structure de données décrivant les objets NAUTILE est orientée objet. Ainsi les objets répondent à des messages qui leurs sont adressés, sans que les messages en question connaissent la nature des objets auxquels on les destine: la plupart des primitives définies dans NAUTILE

possèdent des *sémantiques multiples*, ce qui signifie qu'elles s'appliquent de façon identique à différents types d'objets. Ces sémantiques sont en fait multiples à plusieurs niveaux: elles s'appliquent aussi bien aux différentes vues, qu'aux différents environnements (graphique ou textuel) ou aux différents types de cellules (cellules composées ou motifs).
En clair, cela signifie qu'il n'y aura, par exemple, qu'une seule fonction de routage, applicable aussi bien aux cellules qu'aux motifs, dans n'importe quelle vue, et utilisable de façon similaire en modes graphique et textuel.

Exemple: la primitive "rout" permet la définition d'une cellule de routage reliant deux cellules. L'appel suivant:

```
(rout 'est 'A1 'B 'B1 <ensemble-de-connexions>)
```

va générer une instance appelée B1 de la cellule B, qui sera placée à droite d'une instance appelée A1 dans la cellule courante.
L'interprétation logique de ce résultat sera l'ensemble des connecteurs générés, alors qu'une interprétation géométrique en serait le placement de B à l'est de A1, et la génération du canal de routage avec les différents fils joignant les différents connecteurs.

En revanche, il existera toujours dans le système des primitives spécifiques à une vue donnée: c'est par exemple le cas de la primitive "addcapa" (ajout d'une capacité), qui n'offre d'intérêt que dans un contexte électrique.

Nous allons maintenant décrire les différents objets composant cette structure, avant d'étudier les différentes primitives du langage NIL permettant de les manipuler.

## 5.2. La structure détaillée

Toute la structure de données est composée de classes et de sous-classes, avec des méthodes correspondant à chaque classe.
Dans la suite, nous présentons les caractéristiques de chacune des classes essentielles, en indiquant à la fin la description exacte de ces classes et de leurs attributs respectifs (on pourra également se reporter à [Hornik 89] pour plus de détail sur la structure de données, ainsi qu'à l'annexe 3 pour le texte complet de la structure).

### 5.2.1. La structure NAUTILE: le contexte de travail

Cette classe comporte un certain nombre d'informations concernant le contexte de travail courant:

La liste des répertoires accessibles par le logiciel NAUTILE:
L'ensemble des objets de type "vue" est réparti dans des "répertoires", un répertoire correspondant dans la version actuelle du système à un "directory" UNIX. Le système mémorise donc un répertoire par type de vue (ou plutot par système externe), ainsi qu'un répertoire courant.

La liste des hiérarchies accessibles
La figure 5.1 précise la structure des hiérarchies accessibles:

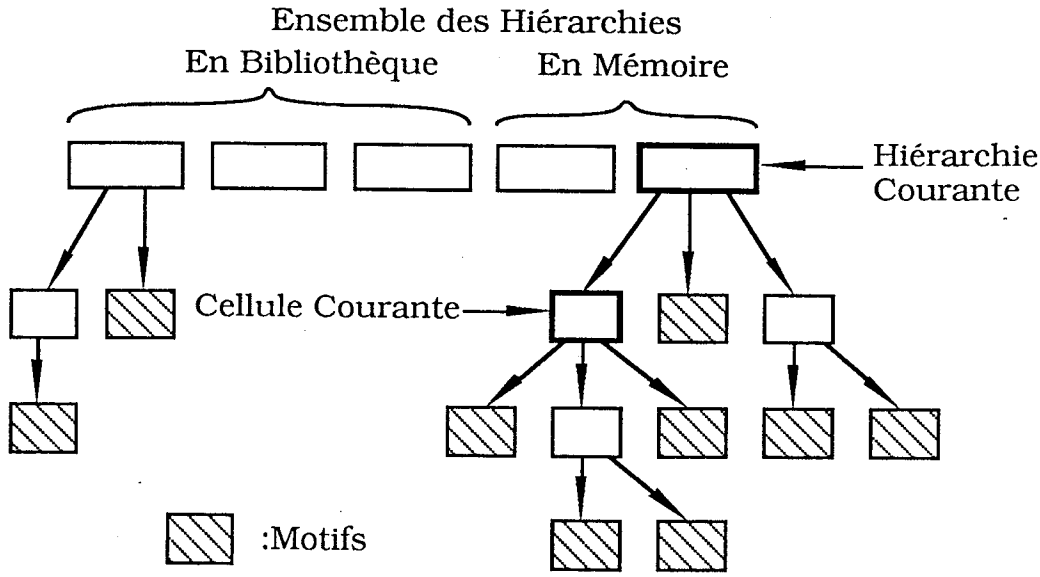



Fig. 5.1: Découpage de l'espace des hiérarchies

La liste des hiérarchies accessibles, est divisée en deux parties distinctes:

- les hiérarchies gardées en mémoire centrale
- les hiérarchies conservées en bibliothèque

Dans l'ensemble des hiérarchies, l'une d'elles est la hiérarchie courante. Parmi les cellules formant la hiérarchie courante, l'une d'elles est la cellule courante, qui peut être soit composée d'autres cellules (c'est le cas de la figure 5.1), soit d'éléments primitifs ou de pointeurs vers des cellules externes (s'il s'agit d'un motif).

La cellule courante
C'est la cellule sur laquelle on est en train de travailler.

Le mode de travail

Le mode de travail permet de connaitre la vue courante. Cette information sert essentiellement aux processeurs de parcours de la hiérarchie: processeurs d'affichage graphique, et processeurs de mise à plat de la hiérarchie. C'est en effet du mode de travail que dépend le contexte de parcours de la hiérarchie pour ces différents processeurs.

Des informations diverses sur l'affichage graphique:

Comportant notamment des informations liées aux niveaux de masques (telles que leur nom et leur couleur associée), à la profondeur d'exploration de la hiérarchie (lors de l'affichage d'une cellule), à l'affichage des noms, etc...

Le tableau suivant résume ces différentes informations:

| **STRUCT** | |
|---|---|
| *Champs* | *Type* |
| Ldir | (list string) |
| Lhier | (list symbol) |
| Cell-cour | CELL |
| Mode | symbol |
| Dessin | Ens-Var |

### 5.2.2. Les cellules et motifs

Les cellules et les motifs possèdent des structures relativement similaires, même si ils sont représentés par des classes distinctes. Leurs différences essentielles tiennent au fait que les motifs sont des éléments terminaux, et qu'ils ne peuvent donc appeler de cellules (ou d'autres motifs). Ils ne peuvent également pas posséder de bibliothèque de cellules locales.

Les tableaux suivants comparent la structure des motifs et des cellules:

| **MOTIF** | |
|---|---|
| *Champs* | *Type* |
| Nom | symbol |
| Etat | ETAT |
| Env | ENVIRONNEMENT |
| Corps | (list VUE) |

| CELLULE | |
|---|---|
| *Champs* | *Type* |
| Nom | symbol |
| Etat | ETAT |
| Env | ENVIRONNEMENT |
| Corps | (list VUE) |
| Filles | (list symbol) |
| Const | CONSTRUCTION |

On peut remarquer sur ces tableaux que la structure des cellules constitue un raffinement de la structure des motifs. On pourrait en déduire hativement qu'il suffirait alors que la classe "cellule" soit une sous-classe de la classe "motif" pour refléter cet état de fait. Cependant ce serait une erreur: en effet de nombreuses méthodes ne s'appliquent qu'aux motifs (telles que les primitives d'ajout d'un élément de base, comme un rectangle par exemple). Or, si la classe "cellule" était une sous-classe de la classe "motif", toutes les méthodes applicables aux motifs le seraient également aux cellules. Aussi, afin de tirer partie de ces analogies sans pour autant faire des cellules une sous-classe des motifs, nous avons introduit une classe supplémentaire qui est une super-classe à la fois des cellules et des motifs: la classe "cell" (le nom n'est peut-être pas trés judicieux).
Nous allons maintenant détailler chacun de ces différents attributs:

• Le nom: toutes les cellules et tous les motifs possèdent un nom propre, dont la portée est limité à la cellule où ils sont déclarés, et à sa descendance. Ceci permet de construire des noms complexes non redondants (voir paragraphe 5.4). Ce nom est donné soit par l'utilisateur, soit par défaut par le système. Ce dernier cas est notamment celui des outils générant des motifs (comme les routeurs): ceux-ci doivent en effet fréquemment attribuer des noms aux motifs qu'ils créent, car l'utilisateur ne le fait généralement pas de façon explicite.

• Les filles: chaque cellule comporte une liste des cellules qui sont déclarées dans son environnement (la hiérarchie de déclaration). Ceci, ajouté à la structuration des noms permet également d'éviter la redondance des cellules.

• La vue environnement: elle décrit les relations de la cellule ou du motif avec l'extérieur. Sa structure sera étudiée dans le paragraphe 5.2.4.

• L'état: il permet de mémoriser des information d'ordre chronologiques, afin de procéder à des vérifications de la cohérence verticale (voir les paragraphes 4.1 sur la cohérence, et 5.2.7 sur la structure de l'état)

• Le corps: il mémorise la liste des vues physiques disponibles. Pour les motifs il est absolument nécessaire que ce corps soit non vide (faute de quoi le circuit ne correspondrait à rien). En revanche, le corps des cellules est généralement vide: les seuls cas où le concepteur précise une vue physique particuliere pour une cellule est pour briser la hiérarchie, ou pour imposer une vue précise que le calcul hiérarchique n'aurait par exemple pas su optimiser.
Il convient de remarquer que l'organisation actuelle de la structure de données limite pour l'instant les vues d'un même type à un seul représentant: il ne pourra par exemple pas exister deux vues topologiques différentes pour une même cellule (en réalité il peut y avoir plusieurs vues différentes d'un même type, mais seule la première sera prise en compte).

• La vue de construction: propre aux cellules, elle mémorise l'arbre d'appels (la liste des cellules instanciées). Sa structure sera précisée dans le paragraphe 5.2.3.

### 5.2.3. Les appels et la vue de construction

Les cellules sont organisées sous forme d'arbre, un arbre d'appel définissant une hiérarchie, et peuvent être décrites suivant les différentes vues, bien que généralement la seule vue qui soit donnée explicitement soit la vue de construction.
Celle-ci peut comporter:

des appels de motifs
des appels de cellules
des appels d'outils
des appels de fantomes

Si le concepteur souhaite expanser la cellule dans une vue donnée, le système commencera donc par regarder si la cellule possède un corps défini dans celle-ci, et si ce n'est pas le cas, il utilisera la vue de construction pour une "mise à plat" dans la vue demandée. En fait, l'expansion d'une cellule consistera à évaluer les différentes fonctions composant cette cellule dans la vue souhaitée.

La représentation dans la structure de données des appels comportera:

Pour les appels de cellules:

- nom général de la cellule appelée
- nom d'instance
- position de l'instance
- liste de transformations à appliquer à l'instance[1]

Pour les appels d'outils:

- nom de l'outil appelé
- nom d'instance
- liste de paramètres

Pour les appels de fantomes:

- nom du fantome appelé
- nom d'instance
- position de l'instance
- liste de transformations à appliquer à l'instance[1]

Nous allons maintenant décrire pour chacune des classes énumérées ci-dessus la représentation exacte en structure de données.
La vue de construction:

| **CONSTRUCTION** | |
|---|---|
| *Champs* | *Type* |
| Lappel | (list APPEL) |

[1] Les différentes transformations disponibles sont des compositions de rotations (angles multiples de 90°), et de symétries par rapport aux axes. De plus des répétitions sont possibles.

Les appels de cellules:

| AP_CELL | |
|---|---|
| *Champs* | *Type* |
| Position | Coord |
| Nom_Def | symbol |
| Nom_Inst | string |
| Transfo | fix (i.e. entier) |
| Repet | REPETITION |

Ces appels de cellules peuvent être décomposés en trois sous-classes, les classes "instance" (positionnement absolu dans la cellule), "direct" (positionnement par aboutement) et "'appel de fantome". La classe "direct" possède les mêmes attributs que la classe "appel", complétés par les attributs résumés dans le tableau suivant:

| DIRECT | |
|---|---|
| *Champs* | *Type* |
| Dir | string |
| Ref | symbol |
| Lconn | (list CONNECT) |

Les appels d'outils:

| AP_OUTIL | |
|---|---|
| *Champs* | *Type* |
| Nom | symbol |
| Lparam | (list) |

### 5.2.4. La vue environnement

Pour une plus grande uniformité, ainsi que le maintien de la cohérence entre les différentes vues, il nous a paru souhaitable de regrouper un certain nombre de paramètres dans une vue particulière, et donc commune à toutes les autres vues.
Ainsi en est-il de la frontière, des entrées/sorties, et des paramètres de certains motifs (comme les transistors), qui sont communs aux différentes représentations, même s'ils ne sont pas tous utilisés par chacune d'elles.
La vue ainsi définie est la vue environnement, la cellule (ou le motif) n'étant vue à un niveau élevé de la hiérarchie (en fait par toute cellule

l'appelant) que sous forme de "frontières" et de "connecteurs", le détail de la réalisation de la cellule n'étant pas spécifié.
Cette vue comporte donc une description de la boite enveloppante, la liste des connecteurs, une liste de paramètres, ainsi qu'une icone représentative.
Cette icone permet d'améliorer la lisibilité du dessin ainsi que sa compréhension, en offrant une vision minimale de la cellule. Une telle vision permettra éventuellement de remplacer dans les dessins la cellule qu'elle symbolise. Cependant, si la notion d'icone n'est pas encore vraiment formalisée dans le système, nous espérons arriver à des résultats dans un avenir proche, afin par exemple de pouvoir simuler un circuit uniquement à partir d'une représentation par icones. Il sera alors nécessaire d'associer à ces icones un certain nombre d'informations, et notamment la description électrique correspondante.

Le tableau suivant résume la structure de la vue environnement:

| **ENVIRONNEMENT** | |
|---|---|
| *Champs* | *Type* |
| Connects | (list CONNECT) |
| Lparam | (list) |
| Boite | Rect |
| Icone | ICONE |

### 5.2.5. Le problème des connecteurs

La représentation interne des connecteurs ainsi que les problèmes liés à l'instanciation d'un cellule sont particulièrement bien traités dans [Hornik 89]. Nous ne donnerons ici qu'un résumé de l'étude faite dans ce document, le lecteur intéressé pouvant s'y reporter pour de plus amples informations.
Avant de détailler les différents problèmes inhérents à la gestion des connecteurs, il convient de préciser que dans NAUTILE *les connecteurs sont distincts des équipotentielles*: un connecteur correspond en fait à une position bien déterminée dans la cellule, et non pas à une vision connectique de cette cellule: ainsi deux connecteurs sont égaux si et seulement si ils ont le même emplacement dans le circuit.

### 5.2.5.1. La structure des connecteurs

Les connecteurs sont en fait découpés en deux classes distinctes: la classe des connecteurs "remontés" (c'est-à-dire provenant d'une cellule instanciée), et les connecteurs "normaux" (c'est-à-dire non remontés!).
Les deux classes possédant des attributs en commun, ces attributs ont été regroupés dans une super-classe appelée "connect". La classe "connect" possède donc deux sous-classes, les classes "connect-n" (connecteurs normaux) et "connect-r" (connecteurs remontés).
Nous allons donc présenter les attributs commun aux deux classes (donc les attributs de la classe "connect"), avant de détailler les spécificités de chacune des classes.

Les informations communes à tous les connecteurs sont les suivantes:

- le nom: dans NAUTILE le nom d'un connecteur doit être unique dans une même cellule. C'est en effet la seule façon de les différencier (pour cette raison il est préférable que le système nomme lui-même les connecteurs). Cette unicité des noms est indispensable avec l'introduction des tables de connecteurs remontés [Hornik 89]. L'utilisateur dispose de moyens variés pour retrouver des connecteurs: par leurs coordonnées, leur appartenance à une équipotentielle, leur type,... Ces recherches sont effectuées à l'aide de fonctions de filtrage.

- "l'occupation": lors de la création d'une instance de cellule, se pose le problème de sélectionner les connecteurs à remonter dans la hiérarchie (voir paragraphe 5.2.5.3). Un indicateur en position vrai ou faux déterminera si le connecteur doit être remonté ou non. Dans la plupart des cas cet indicateur montrera si le connecteur a déjà été utilisé.

- l'équipotentielle: elle définit la notion de connexion entre les différents connecteurs. Il est possible d'assimiler un ensemble de connecteurs en contact électrique à une même famille. Un certain nombre d'informations doivent être accessibles ou calculables: à quelle équipotentielle appartient tel connecteur ou encore quels sont les connecteurs appartenant à cette équipotentielle?

- le type: chaque connecteur peut se voir associer un type particulier. On peut ainsi avoir un type *entrée*, *sortie*, *alimentation*,... Ces informations servent notamment à la vérification de la cohérence lors des routages d'assemblage (à l'aide des règles de construction), ou même à diriger des assemblages élémentaires.

Les informations spécifiques des connecteurs normaux sont les suivantes:

- la direction: chaque connecteur a une direction privilégiée vers laquelle il peut s'étendre. Ceci permet de mieux piloter les routeurs et les mécanismes d'assemblage, en leur précisant uniquement des directions d'assemblage, l'outil retrouvant simplement les connecteurs dans une direction. Ces directions peuvent être nord, sud, est ou ouest.

- le niveau: chaque connecteur est prévu pour s'étendre sur un niveau particulier. Ici encore cette information facilite la tache de certains outils d'assemblage. On retrouve pour ces niveaux les classiques : Aluminium1, Aluminium2, Diffusion, etc. Cependant, l'utilisateur peut toujours employer ses propres notations, sachant qu'il existe des fonctions effectuant la traduction si deux types différents de notations sont utilisés: il conviendra alors de modifier le fichier technologique, et notamment les règles de composition électriques, qui dépendent étroitement de tels paramètres.

- la taille: chaque connecteur a une taille qui lui est propre. Lors des héritages de connecteurs cette taille restera fixe. Cette taille correspondra à la largeur du fil de connection arrivant sur le connecteur.

Les informations spécifiques des connecteurs remontés concernent leur provenance. En effet, ceux-ci ne sont pas créés dans la cellule mais proviennent de l'instanciation d'autres cellules (voir paragraphe 5.2.5.3). Aussi, afin d'offrir des facilités dans la gestion de la cohérence, est-il nécessaire de savoir d'où proviennent ces connecteurs. Leur structure contiendra donc des informations sur:

- le nom de l'instance qui les contient

- un numéro d'ordre permettant en cas de répétition d'une cellule de repérer le numéro exact de l'instance où se trouve le connecteur

- le nom du connecteur dont ils sont issus (c'est-à-dire le nom du connecteur de la cellule instanciée dont la remontée va produire le connecteur en question).

Enfin, il convient de remarquer qu'un attribut un peu spécial est disponible pour les deux classes de connecteurs: il s'agit de la position du connecteur. Cet attribut est spécial, car dans le cas d'un connecteur normal il fait partie intégrante de la structure, alors que dans le cas d'un connecteur remonté il s'agit d'un méthode qui donne la position du connecteur par calcul récursif sur l'arbre d'appel d'où il provient. Ceci permet de rendre transparent à l'utilisateur le fait que la position soit calculée ou pas: celui-ci n'aura en effet qu'à envoyer le message "position" au connecteur qu'il souhaite examiner pour avoir le résultat. C'est le connecteur qui décidera alors suivant son type de la procédure à adopter (soit lire directement sa structure, soit calculer le résultat).
Dans tous les cas, lors de la demande d'information de la position d'un connecteur, celle-ci est calculée dans le repère courant, c'est-à-dire par rapport au point origine de la cellule courante.

Nous indiquons maintenant dans l'ordre respectif: la structure de la super-classe "connect", et les structures des classes "connecteur normal" et "connecteur remonté".

| **CONNECT** | |
|---|---|
| *Champs* | *Type* |
| Nom | string |
| Occupe | cons (i.e. booléeen) |
| Equi | string |
| Type | string |

| **CONN-N** | |
|---|---|
| *Champs* | *Type* |
| Dir | string |
| Niveau | string |
| Position | Coord |
| Taille | fix |

| CONN-R | |
|---|---|
| *Champs* | *Type* |
| N-inst | string |
| N-Connect | string |
| Num | fix |

### 5.2.5.2. La gestion des connecteurs et des équipotentielles

Les choix effectués dans ce cadre sont parmi les plus criticables de NAUTILE. En effet, afin de simplifier la gestion des connecteurs et des équipotentielles, nous avons pris comme décision de ne pas maintenir de listes croisées de connecteurs et d'équipotentielles. Ceci nous évite ainsi des mises à jour excessives de l'une de ces listes lorsque l'autre liste est modifiée. On ne pourra ainsi pas tomber sur l'un des deux dilemnes qui se présentent lors de la gestion des connecteurs et des équipotentielles: c'est soit la suppression des connecteurs, soit celle des liens entre eux qui s'avère excessivement complexe, et même souvent impossible.
Ainsi n'existent dans le système que des objets de type *connecteur*, ces objets comportant un champ *équipotentielle* qui conserve le nom de l'équipotentielle à laquelle ils sont rattachés. Les équipotentielles seront donc extraites dynamiquement en parcourant l'ensemble des connecteurs, à chaque fois que le besoin s'en fera ressentir. De cette façon, si on supprime un connecteur, il n'y aura aucune mise à jour des équipotentielles à effectuer.
La limitation actuelle provient du fait qu'un connecteur ne peut appartenir qu'à une seule équipotentielle. Ceci impose que lorsqu'on souhaite relier deux connecteurs ensemble, ils doivent posséder des valeurs identiques pour leurs champs "équipotentielle" respectifs (ou bien l'une de ces valeurs peut être nulle). Le tableau 5.1 résume les différentes possibilités:

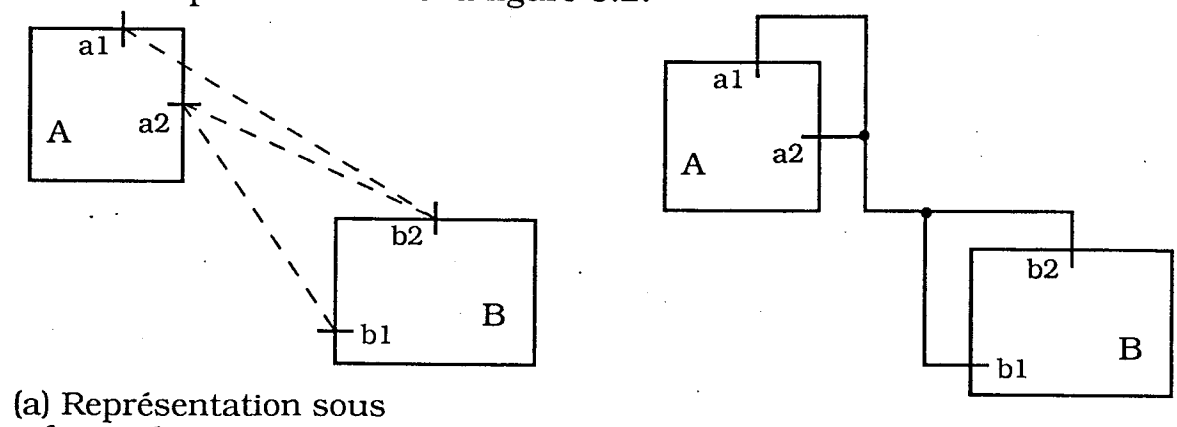
| Connecteur 1 | Connecteur 2 | Résultat de l'union |
|---|---|---|
| vide | vide | possible |
| vide | équipotentielle 1 | possible |
| équipotentielle 1 | vide | possible |
| équipotentielle 1 | équipotentielle 1 | possible |
| équipotentielle 1 | équipotentielle 2 | impossible |

Tableau 5.1: Les différentes possibilités de connexion

Accepter le dernier cas reviendrait à procéder à une union de deux équipotentielles, ce qui poserait un certain nombre de problèmes relativement complexes à résoudre: notamment pour savoir quelle est l'équipotentielle qui l'emporte sur l'autre (*équipotentielle absorbante*), ou encore pour mettre à jour l'ensemble des connecteurs appartenant à l'une ou l'autre de ces équipotentielles. Sans compter que dans ce cas, la suppression de l'un des deux connecteurs en question devra rétablir le situation précédant leur union, ce qui se révèle être relativement complexe à gérer.

Une solution à ce problème pourrait consister à conserver non pas une équipotentielle, mais la liste des équipotentielles auxquelles appartient le connecteur. Cependant cette solution ne permet pas de prendre en compte les problèmes liés à "l'absorption" d'une équipotentielle par une autre (par exemple l'alimentation doit pouvoir absorber toutes les autres équipotentielles). Ce problème est celui de la gestion "dynamique" des équipotentielles: dans le cas du parcours des connecteurs pour obtenir une équipotentielle donnée (soit eq1), que va-t-il se passer lorsque l'on arrive sur un connecteur appartenant à deux équipotentielles (par exemple eq1 et eq2). Dans ce cas il va falloir recommencer la recherche pour eq2, et ainsi de suite. Ce problème revient en fait à partitionner un ensemble de points: on examine tous les points et on les classe.

Une autre solution pourrait être de maintenir une liste de *connexions* chaque connexion se référant uniquement à deux connecteurs. Cette solution est présentée dans la figure 5.2:




(a) Représentation sous forme de connexions

(b) L'équipotentielle résultante

Fig. 5.2: Gestion des connexions

Dans l'exemple de la figure 5.2, on souhaite relier les connecteurs a1, a2, b1 et b2. On voit sur cette figure que pour procéder ainsi, on peut se

ramener à trouver les paires de connexions entre les deux cellules: si on souhaite extraire une équipotentielle, il suffit alors de parcourir l'ensemble des connexions ainsi réalisées. L'intérêt d'une telle méthode tient au fait qu'il est trés facile de supprimer une connexion, sans que cela impacte sur les autres connexions, ou les connecteurs non concernés par cette suppression. Cependant, dans le cas de la suppression d'un connecteur, on doit toujours parcourir toute la liste des connexions, afin de supprimer celles où ce connecteur apparait. D'autre part, il est quasiment impossible avec une telle solution de donner des noms aux équipotentielles. Enfin, le problème le plus important avec une telle représentation, est l'énorme quantité d'informations qui doivent être stockées pour décrire toutes les connexions, ce qui la rend de ce fait peu réalisable.

#### 5.2.5.3. La remontée des connecteurs

Un autre problème que l'on rencontre dans la gestion des connecteurs est celui de savoir que faire des connecteurs qui n'ont aucune possibilité de liaison, une fois la cellule instanciée. Ce problème, décrit dans la figure 5.3, est particulièrement bien adressé par [Hornik 89].

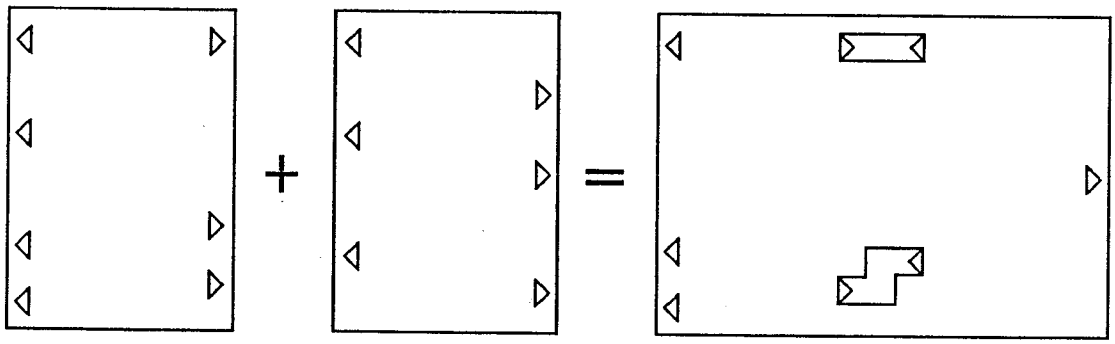


Fig. 5.3: Remontée des connecteurs

On voit sur la figure 5.3 qu'un certain nombre de connecteurs des cellules assemblés sont restés non résolus: c'est-à-dire qu'ils n'appartiennent à aucune équipotentielle de la cellule résultant de l'assemblage. Le problème qui émerge alors est de savoir que faire de ces connecteurs. La solution adoptée dans NAUTILE consiste à faire "remonter" tous les connecteurs non résolus: lors de l'instanciation d'une cellule dans une autre, tous les connecteurs de la cellule appelée qui n'ont pas de connexion avec d'autres connecteurs deviennent des connecteurs de la cellule appelante. Cependant, il faut pouvoir laisser le choix à l'utilisateur de transgresser cette règle quand il le désire (il peut effectivement souhaiter qu'un connecteur remonte, même s'il appartient déja à une équipotentielle).

La stratégie adoptée est alors la suivante:

- si le concepteur ne précise rien, il y a remontée automatique des connecteurs non résolus dans la cellule appelante
- le concepteur peut préciser de façon sélective les connecteurs qui doivent remonter, en utilisant pour cela des fonctions de filtrage permettant de sélectionner des types de connecteurs particuliers.

Un connecteur utilisé par un outil d'assemblage sera donc marqué (c'est la fonction de l'attribut "occupé"), et ne sera plus remonté, sauf sur demande explicite lors de l'instanciation des cellules.

Il convient également de noter que certains connecteurs spéciaux, tels que des connecteurs permettant la polarisation du substrat, ne seront *jamais* remontés, ni ne possèderont d'information d'équipotentielle.
Il conviendrait d'ailleurs afin de simplifier ces traitements, de créer des classes spéciales pour ce genre de connecteurs.

Enfin, on peut remarquer qu'il existe des noms globaux, permettant de remonter l'équipotentielle à laquelle appartient un connecteur: il en est ainsi notamment de la masse et de l'alimentation, dont les noms sont prédéfinis. Ainsi, en cas de remontée d'un connecteur, si le nom de son équipotentielle est global, on remonte également ce nom.

### 5.2.6. Les différentes vues

La structure de données NAUTILE comporte une classe spéciale permettant de regrouper toutes les vues: il s'agit de la classe "vue". Cette classe comporte des sous-classes, qui sont la vue de construction (classe "construction"), la vue environnement (classe "environnement"), et l'ensemble des vues physiques (classe "physique"). Les vues environnement et construction ayant été présentées ci-dessus, il nous reste à aborder les vues physiques.
Les vues physiques sont décomposées en deux catégories, selon qu'elles sont issues d'un système externe, ou qu'elles sont écrites en NIL.

Les vues physiques externes:
Elles comportent un attribut permettant de connaitre le type de la vue (topo, elec, ...), ainsi que des informations permettant de connaitre leur origine: le système dont elles sont issues et le fichier qui les contient. Cette dernière information n'est pas toujours nécessaire: par exemple les

vues décrites en ASTAP ne sont pas contenues dans des fichiers. Il est seulement nécessaire que les motifs décrivant une vue ASTAP soient présents dans la mémoire centrale de l'ordinateur qui aura à les traiter.

Les vues physiques internes:
Elles sont formées d'une liste d'éléments de base qui sont propres à chaque vue. Ainsi la vue électrique comportera des transistors, des capacités, des résistances, et éventuellement des connexions permettant de reconstituer les nœuds multiples, alors que la vue topologique sera formée de rectangles et de polygones.
Chaque élément appartient à une classe particulière. Par exemple, la classe "transistor", qui n'a de sens que dans un contexte électrique, définit un transistor par son type (transistor N, transistor P, transistor déplété, transistor enrichi, ...), ses nœuds (grille, source et drain), et ses paramètres (W et L). De même, chaque rectangle et chaque polygone est caractérisé par ses propriétés géométriques (pour un rectangle ce seront deux points, et pour un polygone une suite de points), et par un niveau technologique (visualisé sur l'écran par une couleur).

Les tableaux suivants montrent les structures des vues physiques:

| **EXTERNE** | |
|---|---|
| *Champs* | *Type* |
| Type_vue | symbol |
| Nom_fichier | string |
| Origine | "type de vue"[1] |

| **INTERNE** | |
|---|---|
| *Champs* | *Type* |
| Type_vue | symbol |
| Liste | (list) |

[1]Cet attribut est typé, c'est-à-dire qu'il existe dans la structure une classe "ELEC", une classe "TOPO", etc, permettant de typer les vues. Ceci permet d'appliquer des méthodes déterminées par le type de vue.

### 5.2.7. La cohérence hiérarchique et l'état

Nous avons vu (paragraphe 4.1.5) que la cellule comportait un certain nombre d'informations permettant de gérer la cohérence hiérarchique, par des méthodes basées sur la chronologie des appels. Chaque cellule comporte notamment des indications telles que la date de sa création et de sa dernière modification, le nom de son propriétaire (ou de l'outil qui l'a créée) et des informations relatives à la cohérence (limitées pour l'instant à un booléen indiquant si la cellule n'est ou n'est pas cohérente). L'état fait également partie de la structure des différentes vues, afin de permettre des vérifications élémentaires sur ces vues (ici le nom du propriétaire peut indiquer le moyen d'obtention de la vue: si la vue est une vue importée, ou au contraire si elle correspond à la mise à plat de la vue de construction dans la vue en question).

Le tableau suivant donne la structure de l'état:

| **ETAT** | |
|---|---|
| *Champs* | *Type* |
| Date_Creat | DATE |
| Date_Modif | DATE |
| Origine | string |
| Coherence | (list) |

## 5.3. Les primitives du langage NIL

Le langage NIL est immergé dans LISP, c'est-à-dire qu'il consiste en un ensemble de primitives, les unes propres au langage LISP, les autres propres à NIL. Nous ne détaillerons pas ici les primitives propres à Lisp, le lecteur pouvant se reporter sur ce sujet à [Chailloux 88]. Nous nous attacherons plutôt à mettre en avant les particularités du langage NIL liées à la conception des circuits.

Avant de détailler les différent types de fonctions et primitives, il convient de remarquer qu'on peut découper celles-ci en deux ensembles distincts: les primitives à sémantiques multiples, et les primitives ne s'appliquant que dans un contexte particulier. La catégorie des primitives à sémantiques multiples recouvre toutes les fonctions qui peuvent s'appliquer dans n'importe quel contexte: elles ont été définies de telle sorte que leur résultat seul dépend du contexte de leur utilisation.

En revanche, il existe des primitives qui ne peuvent s'appliquer que dans un contexte bien précis: il s'agit notamment de primitives ayant trait à la gestion des éléments de base d'un motif.

Outre ce découpage suivant le domaine d'application des fonctions, on peut également diviser celles-ci suivant leur fonction. On distingue ainsi:

- les primitives de base du langage
- les outils
- les générateurs

Nous ne nous étendrons pas sur les deux dernières catégories, le nombre des outils et des générateurs développés en NAUTILE ne permettant guère de généralisation.

Nous allons donc maintenant nous attacher à décrire les primitives de base du système. Dans la suite, les primitives sont représentées avec leurs paramètres d'appel. Le signe "?" signifie que ce qui suit est tapé par l'utilisateur, alors que le signe "=" indique ce que retourne le système. En outre les paramètres optionnels sont en italique.

### 5.3.1. Les primitives de gestion des données.

LeLisp (et son extension CEYX) propose des primitives permettant la gestion des structures de données: création, modification et consultation d'une structure. Aussi n'avons-nous pas eu besoin d'écrire à nouveau ces primitives. En revanche le langage NIL comporte des primitives de manipulation des données qui lui sont propres: il s'agit notamment de primitives permettant le chargement en mémoire d'une hiérarchie, ou bien sa sauvegarde dans un fichier. Ces primitives sont ainsi celles qui permettent les liaisons avec les systèmes externes (elles peuvent en effet sauvegarder une hiérarchie dans un système autre que NAUTILE).

Chargement de cellules depuis un fichier:

?(charge <système> <fichier> <*cellule*>)

=<Nom réel de cellule>

Le nom "réel" de la cellule retourné par cette fonction (ainsi que les suivantes), est en fait un nom hiérarchique: il est constitué de la concaténation des noms de ces ancetres. C'est ce qu'on appelle un "cheminon" ("pathname" en anglais). Il s'oppose au nom relatif de la

cellule, qui est celui donné par l'utilisateur (la plupart des primitives utilisent comme paramètres d'entrée des noms relatifs, et retournent des noms absolus).

Sauvegarde d'une cellule :

?(genere <système> <fichier> <*cellule*>)

=<Nom réel de cellule>

### 5.3.2. Les primitives de construction

Directement dépendantes des fonctions de gestion des données, les primitives de construction permettent de construire de nouvelles données: il existe ainsi des primitives permettant de définir une nouvelle cellule, un nouveau motif, de nouveaux connecteurs, etc...

Définition d'une nouvelle cellule:

?(defcell <nom de cellule>)

=<Nom réel de cellule>

Définition d'un nouveau motif :

?(defmotif <nom de motif>)

=<Nom réel de cellule>

Ajout d'un connecteur dans une cellule

? (addconn <nom de cellule> <*direction*> <*x*> <*y*> <*largeur*> <*niveau*> <*genre*> <*equipotentielle*> <*nom de connecteur*>)

### 5.3.3. Les fonctions de gestion de l'environnement courant

Ils s'agit des fonctions ayant trait à la manipulation de la cellule courante, du mode, des répertoires de bibliothèques, etc ...

Accès à une cellule: pour pouvoir ajouter des éléments dans la cellule il faut accéder à cette cellule, c'est-à-dire que la cellule devienne la cellule courante.

?(acces <nom de cellule>)

=<Nom réel de cellule>

Fin d'accès à une cellule: cette fonction ferme toutes les cellules ouvertes depuis la dernière ouverture de la cellule citée.

?(fin-acces <*nom de cellule*>)

La fonction mode renverra ou permettra de définir (si un paramètre est présent) le mode de travail courant (cf paragraphe 5.2.1):

?(mode <*mode*>)

=mode

Fonctions de manipulation des bibliothèques de cellules:

?(cd <système> <directory>)

?(ls <système>)

?(ll <système>)

?(pwd <système>)

### 5.3.4. Les fonctions de consultation et de modification

Ces fonctions permettent d'accéder à la structure de données, sans avoir à utiliser les fonctions de manipulation de base (qui sont théoriquement réservées aux programmeurs du système).

Fonction de consultation d'une vue (elle retourne soit la vue en question si celle-ci existe, soit nil dans le cas contraire):

? (vue? <*mode*> <*nom de cellule*>)

Le mode indique le type de la vue que l'on souhaite examiner (en fait un nom de système).

Fonction de consultation d'une vue (elle retourne soit la vue en question si celle-ci existe, soit la vue de construction dans le cas contraire):

? (V? <*mode*> <*nom de cellule*>)

Primitive permettant de connaitre la position d'une instance à l'intérieur d'une cellule donnée:

? (posinst <Nom de l'instance> <Nom de la cellule>)

= (x-inst , y-inst)

Fonction d'extraction de connecteurs: cette fonction renvoie la liste des connecteurs dans la direction donnée de la cellule demandée.

? (conn-dir <Direction> <Nom de cellule>)

Fonction de modification et de consultation de la boite englobante

?(boite <nom de cellule> <*x*> <*y*> <*Dx*> <*Dy*>)

### 5.3.5. Les fonctions d'assemblage

Ces fonctions permettent de réaliser des instances de cellules, avec appel ou non à des routeurs.

Fonction de placement absolu d'une cellule dans la cellule courante:

?(instance <Nom de cellule> <*Nom d'instance*> <x> <y> <*Liste de paramètres*>)

=<Nom de l'instance générée>

Dans le cas de cette fonction, l'objet retourné est le nom de l'instance générée: ceci est nécessaire, par exemple dans le cas où le paramètre "Nom d'instance" est omis. En effet, dans ce cas le système génère automatiquement un nom: celui-ci ne sera donc connu de l'utilisateur que si on le lui indique explicitement. Cela permettra en outre d'enchainer des instructions de placement: on pourrait par exemple avoir une instruction du type:

(direct nord (direct nord (instance 'a () 0 0) 'b) 'c)

qui permet de placer au nord de l'instance de "a" une instance de "b", suivi d'une instance de "c".

Quant à la "Liste de paramètres", elle contient dans la version actuelle les transformations qui doivent être appliquées à la cellule que l'on souhaite instancier. Cette liste peut contenir n'importe quelle composition de:

- symétries:

(sym <Axe de symétrie>) ;avec <Axe de symétrie> ∈ {x,y}

- rotations:

(rot <Angle>) ;avec <Angle> ∈ {90,180,270}

- répétitions:

(rep <Direction> <Nombre de fois> <Decalage en x> <Decalage en y>)

Pour des exemples de listes de paramètres, le lecteur pourra se reporter au paragraphe 3.4.2, ainsi qu'aux annexes 5 et 6.

Fonction de placement relatif avec routage élémentaire:

?(direct <Direction> <Nom de la reference> <Nom de cellule> <*dx*> <*dy*> <*Liste de paramètres*> <*nom de l'instance genérée*>)

=<Nom de l'instance générée>

Fonction de placement avec routage par dévoiement

?(dev <Direction> <Nom de l'instance> <Nom de cellule> <*dx*> <*dy*>)

=<Nom de l'instance générée>

Fonction de placement avec routage bi-couche

? (rout <Direction> <Nom de l'instance> <Nom de cellule> <*Frontière1*> <*Frontière2*> <*Décalage*> <*Liste de connexions*>)

=<Nom de l'instance générée>

Les différents paramètres qui peuvent être pris en compte pour ces fonctions sont essentiellement des paramètres géométriques: transformations (symétries et rotations) et répétitions. Un exemple d'une liste de tels paramètres est donné ci-dessous:

'((sym x) (rot 90) (rep nord 10 0 5))

une telle liste exprime le fait qu'avant d'être instanciée, la cellule sera répétée 10 fois au nord, avec un décalage de 5 unités verticales, le résultat subissant successivement une rotation de 90° puis une symétrie par rapport à l'axe des X.

### 5.3.6. Fonctions de création des vues physiques internes

Ces fonctions, qui sont spécifiques à une vue donnée, permettent de rajouter des éléments de base dans les motifs NIL.

Fonction d'ajout un rectangle

?(addrect <niveau> <x> <y> <Dx> <Dy>)

Fonction d'ajout d'un polygone

?(addpoly <niveau> <liste de points>)

Fonction d'ajout d'un transistor

?(addtrans <type> <grille> <source> <drain> <x> <y> <w> <l>)

Fonction d'ajout d'un nœud

?(addnode <nom> <capacite>)

Fonction d'ajout d'une capacité

?(addcapa <capacite> <nœud1> <nœud2>)

Fonction d'ajout d'une résistance

?(addres <resistance> <nœud1> <nœud2>)

### 5.3.7. Fonctions d'utilité générale

Il existe un certain nombre de fonctions du type utilitaire, comme des fonctions permettant de mettre à plat une vue hiérarchique, d'effacer des cellules ou des instances, de recalculer la boite enveloppante, etc...
D'autre part il existe également tout une catégorie de fonctions graphiques permettant de manipuler les représentations à l'écran des différents objets. Pour plus de détail sur ces fonctions, le lecteur pourra se reporter au manuel d'utilisation de nautile [Nautile 89].

## 5.4. L'environnement de travail

Les spécifications du système NAUTILE nous ont permis d'arriver à un prototype, qui comporte quelques unes des caractéristiques que nous souhaitions donner initialement au système complet.
Nous allons dans une premier temps décrire le langage ayant servi de base au système, puis nous étudierons les relations de NAUTILE avec le système d'exploitation qui l'accueille, avant d'aborder les relations avec l'extérieur. Enfin, nous terminerons par la présentation des relations entre les parties graphique et textuelle.

### 5.4.1. LeLisp

F. Anceau [Anceau 84] (CF paragraphe 2.9.2) a distingué trois possibilité d'implantation d'un système basé sur la description textuelle procédurale des circuits:

- l'immersion dans un langage algorithmique procédural et hiérarchisé
- la paramétrisation du langage de description
- la génération de descriptions statiques

Parmi toutes ces possibilités, nous avons fait le premier choix. Nous disposions en effet du langage LeLisp avec tout son environnement et ses possibilités pour le développement, ce qui a permis de rapidement obtenir un prototype "viable" du système NAUTILE. La seconde solution aurait imposé la création d'un véritable compilateur ou interpréteur. Enfin la dernière solution offrait moins de possibilités: en effet les langages

statiques sont généralement mal adaptés aux manipulations dynamiques dans une structure de données.

Le système est donc immergé dans LeLisp, dialecte de Lisp développé à l'INRIA, comportant une extension orientée objet. Ce choix a été dicté à la fois par l'environnement de conception dans lequel est apparu NAUTILE (notamment les outils développés pour le projet SYCOMORE), et par les nombreuses possibilités offertes par LeLisp pour l'écriture d'un prototype: description procédurale et paramétrée, extensions graphiques du système permettant notamment l'utilisation d'un bitmap virtuel utilisant X-Windows, fonctionnement en mode soit interprété, soit compilé, gestion hiérarchique des noms évitant des conflits et des confusions, etc...
NAUTILE conserve donc toute la souplesse d'un véritable langage de programmation permettant notamment l'utilisation de paramètres (et donc la définition de générateurs) et de structures de controle. On peut ainsi décrire dynamiquement les objets manipulés, par opposition à des descriptions statiques, telles que celles générées par exemple dans le langage EDIF [Edif 87].
Le langage comporte des boucles itératives (for, while, repeat), des sauts conditionnels (if, when). Les sauts conditionnels permettent d'ajouter des éléments à une cellule sous certaines conditions (comme par exemple l'entrée "mise à zéro" d'un registre). Les boucles itératives permettent notamment de construire des structures régulières (matrices de cellules).

L'espace des noms
Une autre facilité importante introduite par LeLisp, est la possibilité de définir des "packages", ou espaces de noms séparés (à rapprocher des mécanismes basés sur ADA, implantés par exemple dans le langage IMAGES de ATT [Hill 89]).
Une cellule est déclarée comme appartenant à un package, et n'est alors connu que de la hiérarchie des descendant de la cellule dans laquelle elle est définie: ceci facilite la construction de noms complexes, sans redondance ou risque de confusion.
La séparation de l'espace des noms permet de mieux cloisonner des parties d'un projet, afin par exemple que deux concepteurs puissent chacun créer une cellule appelée "nand", sans qu'il y ait de conflit.

L'interactivité

Enfin, l'interactivité de LeLisp est à nos yeux essentielle, comparé à des projets comme CHISEL [Karplus 82] (qui constitue un préprocesseur du langage C), où le concepteur ne voit le résultat de ses efforts que lorsque le programme de génération a été complètement compilé et exécuté. Dans NAUTILE, il est possible de suivre le déroulement des opérations, pour revenir en arrière si un choix effectué s'est révélé être erroné, ou si les objectifs poursuivis n'ont pu être atteints.

Le seul inconvénient à nos yeux du langage LeLisp réside dans l'impossibilité de modifier dynamiquement la mémoire allouée à un programme, et à la difficulté de s'interfacer avec des programmes écrits en langage C (on est en effet obligé de recompiler tout Lisp avec les programmes en question).

### 5.4.2. Le système hote

Outre le langage LeLisp, NAUTILE utilise également les ressources du système d'exploitation sur lequel il réside, notamment pour ce qui concerne l'espace de stockage des données. Pour l'instant ce système d'exploitation est le système UNIX.

Les données sont organisées selon les conventions de nommage des fichiers dans lesquels elles résident: cette solution est classiquement celle qui est adoptée pour des systèmes simples, et notamment pour des besoins de prototypage. Ceci permet entre autre de bénéficier pour la gestion des données, des particularités du système de fichiers proposé par le système hote. Ainsi avec le système UNIX, on pourra utiliser les mécanismes de liens dynamiques entre fichiers ("link") pour permettre à des cellules identiques d'être présentes dans des répertoires différents.

Les données sont ainsi lues dans des fichiers UNIX, amenées en mémoire pour y être traitées, pour être écrites de nouveau dans leurs fichiers, une fois le traitement terminé. Les cellules et les motifs sont donc stockés dans des fichiers. L'organisation de ces fichiers diffère selon que l'on a un motif ou une cellule:

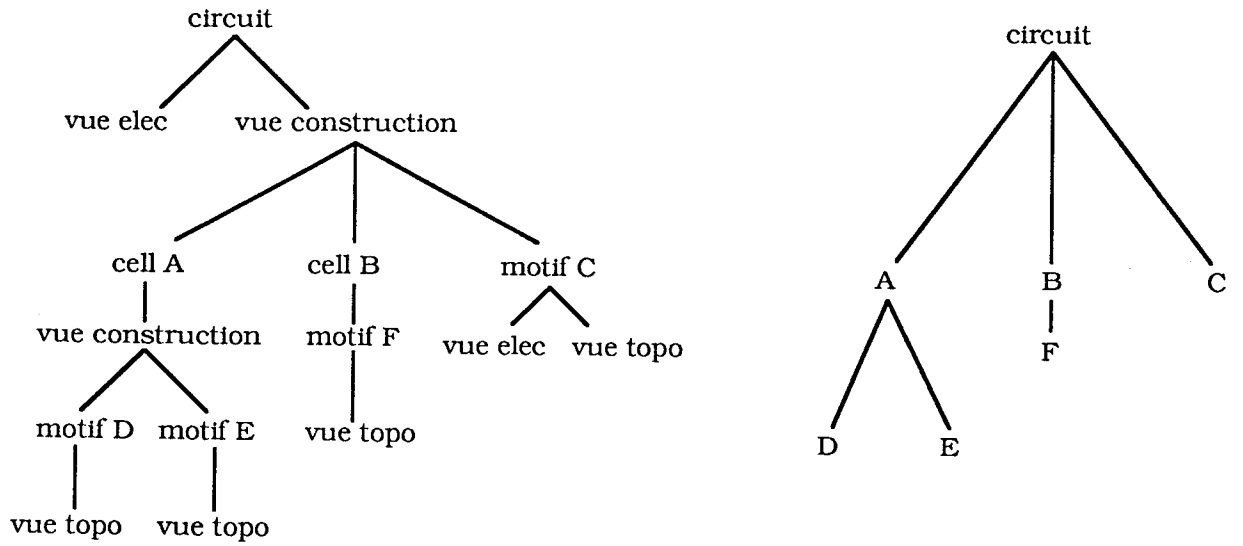
- les motifs: un motif sera décrit par un ensemble de fichiers, un par vue physique, plus un fichier contenant la structure du motif (i.e. sa vue environnement et son corps, permettant de retrouver les différentes vues).

- les cellules: celles-ci ne comportent en général qu'un seul fichier, dans lequel est stocké la structure de la cellule. Les informations sauvegardées sont notamment la vue environnement de la cellule, ainsi que sa vue de construction. Cependant, il se peut que la cellule possède des vues physiques spécifiques: dans ce cas la description de la cellule comportera en plus l'ensemble des fichiers décrivant les diverses vues physiques.

La représentation des vues physiques comporte soit des fichiers compréhensibles par NAUTILE (il s'agit par exemple des fichiers NIL ou GL1), soit des données que NAUTILE ne connait pas (par exemple des données LUCIE). Dans ce dernier cas, NAUTILE agit uniquement comme intermédiaire, en passant les données "brutes" aux outils qui doivent les manipuler.

L'organisation des données

Dans NAUTILE, le stockage des fichiers correspondant aux données est fait de façon globale, par type de vue: ainsi toutes les données correspondant à une vue particulière sont présentes dans un même répertoire, du nom de la vue en question (il y a ainsi un répertoire "LUCIE", "RNL", "GL1",... un répertoire "NIL" comportant les différents générateurs de modules, ainsi que le répertoire "NAUTILE" qui conserve le squelette des motifs et cellules). Cette organisation est d'une grande simplicité, et nous la conservons pour l'instant dans la mesure où nous n'avons que peu de cellules différentes à traiter. Cependant, dans un environnement de production de circuits normalisé, il faudrait probablement revoir cette organisation, et découper les fichiers suivant la hiérarchie des cellules qui y sont stockées.
La figure 5.4 donne une représentation graphique de ce que pourrait donner une telle réalisation:




(a) Représentation de la hiérarchie du circuit

(b) Représentation de la hiérarchie des répertoires

Fig. 5.4: Exemple de découpage hiérarchique des répertoires

Dans la figure 5.4(a), on a représenté la hiérarchie du circuit: celui-ci est découpé selon un arbre d'appel, les feuilles de l'arbre étant les vues des motifs. On peut ainsi associer à chacune des cellules de cet arbre un répertoire UNIX qui contiendra (éventuellement) comme élément les différentes vues de la cellule (figure 5.4(b)).

Remarque:
Dans la version actuelle de NAUTILE, c'est le système LeLisp qui se charge de la lecture et de l'écriture des structures dans un fichier: on utilise pour cela des fonctions d'impression et de lecture spécialisées.

Enfin, il existe un fichier de démarrage appelé ".nautilerc" (selon les conventions de nommage des fichiers de ressource sous UNIX), qui peut contenir des primitives qui seront exécutées lors du lancement du système NAUTILE. Ce fichier de démarrage contient typiquement les ordres de correspondance entre les niveaux de masque, et les couleurs qui seront utilisées pour représenter ces niveaux de masque à l'écran.
L'exemple du fichier .nautilerc utilisé dans la version courante du système au laboratoire IBM est donné dans l'annexe 1.

### 5.4.3. Relation avec les systèmes externes

La version actuelle du système, considérée comme un prototype, comporte une vue topologique et une vue électrique, qui peuvent être traduites respectivement dans les systèmes LUCIE, LUBRICK et GL1 pour le topologique, RNL et ASTAP pour l'électrique.
L'édition de chacun des motifs est faite dans leur système respectifs: ainsi le dessin des motifs nécessaires aux exemples des paragraphes 5.7.1 et 5.7.2 est-il produit dans respectivement les systèmes LUCIE et GL1.
La validation aussi bien des circuits dessinés, que des outils développés dans NAUTILE, est obtenue à l'aide des différents outils de vérification inclus dans les systèmes respectifs.

### 5.4.4. Relation entre parties graphiques et textuelles

Dans NAUTILE, nous avons cherché à soigner particulièrement l'équivalence entre les parties graphique et textuelle.
Cette équivalence est due essentiellement au fait que toutes les primitives graphiques de manipulation (autres que les primitives d'affichage pur) possèdent leur équivalent textuel. En fait les primitives sont des méthodes différentes, mais qui répondent aux mêmes messages. Le résultat dépend donc de leur contexte d'appel.
Pour l'instant, NAUTILE est cependant limité graphiquement à l'affichage des objets, leur édition devant forcément être faite de façon textuelle. Seuls les motifs externes peuvent être définis graphiquement, en utilisant les éditeurs graphiques disponibles sur leur système d'origine.

## 5.5. Méthodologie de conception

Nous ne prétendons pas indiquer ici une méthodologie générale de travail avec un outil tel que NAUTILE, ayant parfois été surpris par l'usage qui en est fait par les concepteurs l'utilisant, mais plutôt un aperçu de l'éventail des possibilités qui sont offertes à l'utilisateur.

L'usage initial de NAUTILE étant la construction de générateurs, il convient de remarquer que pour des générateurs de petite taille, la meilleure méthode consiste à définir graphiquement un exemplaire de la cellule à générer (parce que c'est bien plus simple que d'écrire un programme faisant le même travail), puis on édite le texte produit par

NAUTILE pour cette cellule (NAUTILE permet d'extraire la suite d'instructions aboutissant à la construction d'une cellule donnée, ce qui est utile entre autre, pour opérer des retours en arrière en cas d'erreur de conception ou de jugement), en y incluant les paramètres souhaités.
NAUTILE peut donc servir à écrire des compilateurs de silicium, à réaliser différentes versions d'une même classe de cellules, ou à aider à la validation des choix effectués par le concepteur. Dans ce dernier cas, celui-ci commence par construire son circuit à l'aide de NAUTILE (c'est-à-dire rapidement), il évalue alors aussi bien la taille des différents blocs le composant, que leurs caractéristiques électriques ou temporelles (délais et temps de réaction). Si besoin est, il peut enfin redessiner manuellement une partie peut-être plus critique du circuit, en termes soit de performances, soit d'encombrement.

Méthodologie de conception incrémentale

Les différentes informations contenues dans une cellule peuvent être complétées au fur et à mesure que la conception se précise (soit par l'utilisation d'outils, soit directement, à l'aide de primitives de rajout d'informations).

La construction de cette cellule en NAUTILE est dite incrémentale: on complète la cellule au fur et à mesure que de nouveaux paramètres sont connus. Le circuit est achevé lorsque tous les paramètres sont déterminés: ainsi pour les connecteurs, un certain nombre d'informations telles que la taille, le niveau, la largeur, la position ou la direction ne sont connues qu'en phase finale de conception.
La définition d'une cellule peut donc être décomposée comme suit: on crée la cellule, on y ajoute des éléments, on la ferme pour examiner une autre cellule, on y revient ultérieurement pour la modifier, fixer un certain nombre de paramètres; etc...

Méthodologie d'utilisation des motifs externes

L'intérêt majeur de NAUTILE est de permettre la définition des motifs dans d'autres systèmes que le sien. La figure 5.5 décrit la façon dont NAUTILE peut être utilisé dans cette optique:

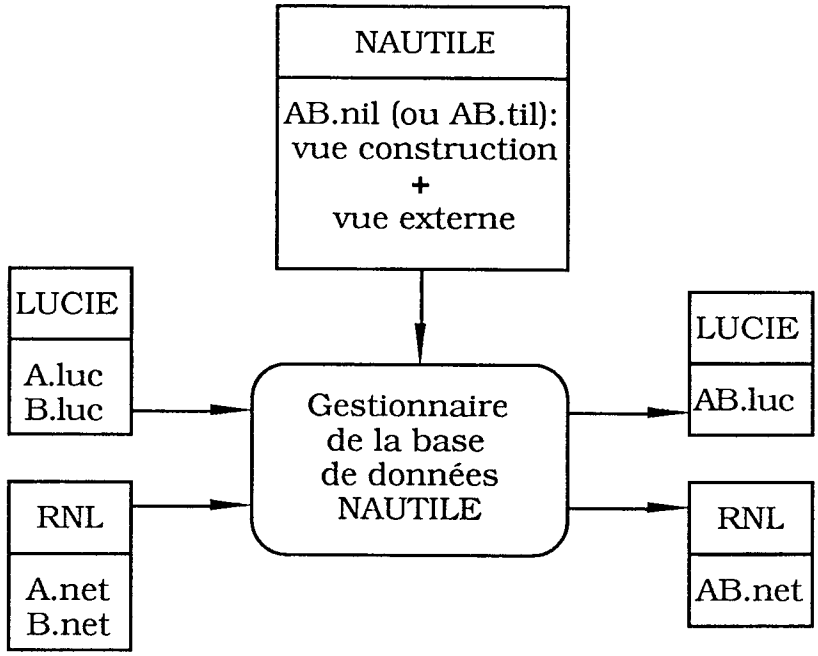



Fig. 5.5: NAUTILE vu du côté utilisateur

Partant des deux motifs "A" et "B" définis dans deux vues physiques externes (fichiers "A.net" et "B.net" pour RNL, "A.luc" et "B.luc" pour LUCIE), le concepteur peut construire une cellule de composition résultant d'un assemblage de ces deux cellules.
Il compose alors la suite d'instructions NIL nécessaires à la formalisation de la cellule souhaitée: il peut le faire soit interactivement, soit écrire la suite d'instruction dans un fichier qu'il exécute par la suite (il s'agit alors d'un générateur de modules). Il faut également qu'il n'oublie pas de préciser un certain nombre d'informations concernant les deux motifs A et B, tel que leur vue environnement (boite enveloppante et connecteurs).
Quoi qu'il en soit, il doit alors sauvegarder la cellule dans un fichier (appelé "AB.til") qui contiendra la structure de la cellule (en fait uniquement la vue environnement, et la vue construction). Le système génère alors automatiquement sur demande les vues physiques de la cellule résultante (dans les fichiers "AB.net" pour RNL, et "AB.luc" pour LUCIE).

Méthodologie générale d'utilisation

La figure 5.6 permet de mieux saisir le processus de conception avec NAUTILE:

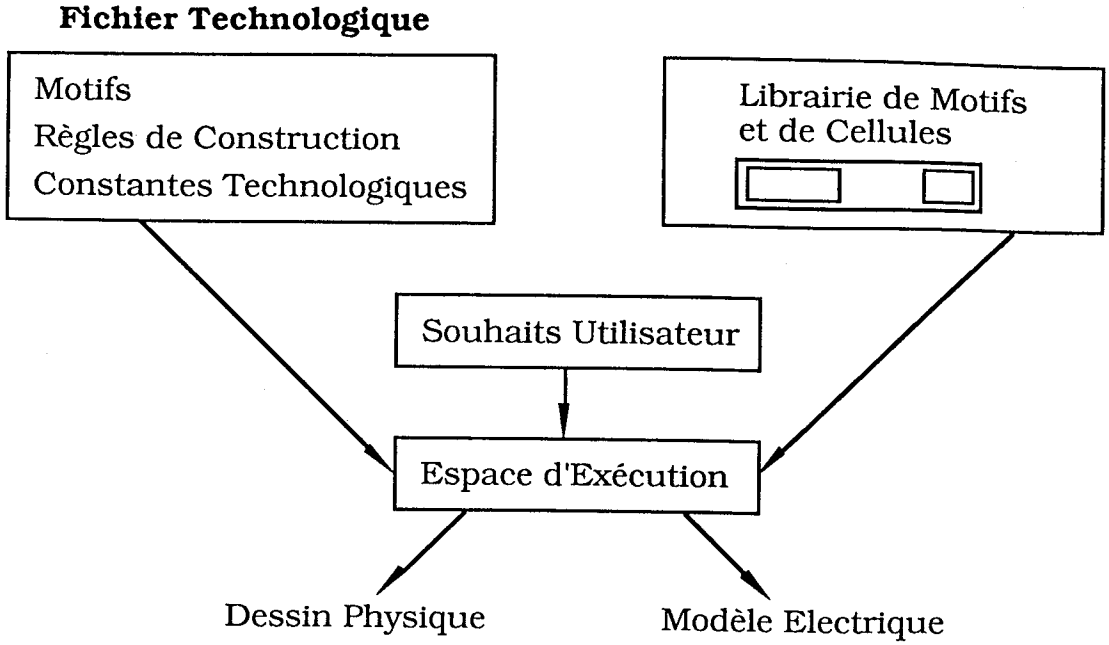



Fig. 5.6: Méthodologie d'utilisation de NAUTILE

Les masques sont obtenus par assemblage de cellules et de motifs définis en bibliothèque. Les cellules et les motifs sont conçus pour s'assembler facilement avec leurs congénères: les problèmes de connexion sont ainsi résolus au niveau de la définition des cellules de base, et non au niveau de l'assemblage. Cela signifie qu'il est nécessaire de réserver des canaux de passage pour les bus à l'intérieur des cellules (cf les fantomes au paragraphe 4.4.4), et qu'il faut prévoir la boite enveloppante en fonction des assemblages que l'on souhaite réaliser. Une cellule doit donc respecter un certain nombre de contraintes:

- avoir une structure de bus prédéfinie
- posséder uniquement des possibilités de connexions directes

Les connexions par bus se font au niveau de l'assemblage global, et non au niveau de chaque cellule, puisqu'un passage est réservé pour chaque bus. Il ne reste au concepteur qu'à gérer les connexions locales entre deux cellules.

L'assemblage est fait automatiquement, par des outils permettant de vérifier les différentes contraintes, à l'aide notamment des règles de construction du fichier technologique.

L'utilisateur peut finalement visualiser et traiter le résultat de l'assemblage dans une des vues disponibles (pour l'instant vues topologique et électrique uniquement).

Des circuits "corrects par construction":

Un des atouts majeurs de NAUTILE est d'offrir une méthodologie de conception sûre. Le concepteur peut développer des circuits corrects par construction,s'il respecte les deux étapes suivantes:

-1- tous les motifs sont définis explicitement, ou sont extraits d'une bibliothèque déja existante. Ces motifs sont supposés corrects par définition, ce qui implique qu'ils ont été vérifiés et simulés au cours d'une phase précédente.

-2- les cellules sont alors formalisées dans une approche soit montante, soit descendante, de façon hiérarchique. Durant toute cette phase, des outils divers tels que routeurs et outils de placement, peuvent être utilisés pour améliorer le dessin final, et bien sur réduire le temps de conception.

Role de NAUTILE dans la formalisation du compilateur SYCO

Bien qu'elle ne mette en œuvre que des algorithmes bien maîtrisés, la génération des dessins de masques constitue à la fois l'épine dorsale et le goulet d'étranglement de tout compilateur de comportement [Jerraya 89]. C'est l'épine dorsale car, sans elle, on ne peut valider les autres outils de compilation; c'est un goulet d'étranglement car elle nécessite des efforts de développement très grands, qui sont anéantis à chaque changement de technologie. Dans le cas du compilateur SYCO, cette situation est aggravée par l'utilisation d'outils de génération de dessins des masques développés spécialement à cette fin. Avec le développement actuel des compilateurs d'architecture et des environnements pour la compilation de silicium, plusieurs solutions sont possibles pour maîtriser la génération des dessins des masques.

Une première solution consiste à utiliser un compilateur d'architecture tel que GENESIL [Cheng 88] comme environnement pour le compilateur de comportement. Un tel environnement fournit des compilateurs spécialisés (compilateur de contrôleurs, compilateur de parties opératives) et une bibliothèque de cellules standards. Dans ce cas, la tâche du compilateur sera réduite à la génération de l'architecture. Cette solution résoud complètement les problèmes de génération des dessins des masques, puisqu'elle permet des changements importants de technologie. Cependant elle rend difficile l'utilisation d'une stratégie de plan de masse: la topologie des cellules standards et celle des blocs

générés par les compilateurs spécialisés sont fixes et ne sont pas nécessairement adaptés à la stratégie choisie pour l'implantation.

Une seconde solution consiste à utiliser un système de dessin symbolique tel que GDT [Burich 87] ou STYX [Rougeaux 87]. Dans ce cas, les éléments de la bibliothèque seront décrits sous forme symbolique, et les compilateurs spécialisés généreront des dessins symboliques. La transposition dans la technologie finale se fera par des outils d'expansion et de compactage de la description symbolique. Cette solution ne couvre que des changements technologiques mineurs, tels que l'évolution d'un procédé de fabrication. En revanche, elle ne permet par exemple pas l'introduction d'un nouveau niveau d'interconnexion. Cette solution est bien adaptée dans le cas d'un processus de conception manuel, mais elle est difficile à mettre en œuvre dans le cadre d'un compilateur de silicium.

Une troisième solution consiste à utiliser des générateurs de cellules et de structures régulières, paramétrés par la technologie. Dans cette solution, on assemble des motifs de base définis en fonction des besoins du compilateur de silicium. Les parties les plus critiques du circuit (et les moins automatisables: NAUTILE n'utilise pas d'outil automatique de synthèse de dessin) seront donc définies comme des motifs, les autres parties résultant d'un assemblage de ces motifs.

La figure 5.7 tirée de [Jerraya 89] indique sur le schéma synoptique du compilateur SYCO le role de NAUTILE:

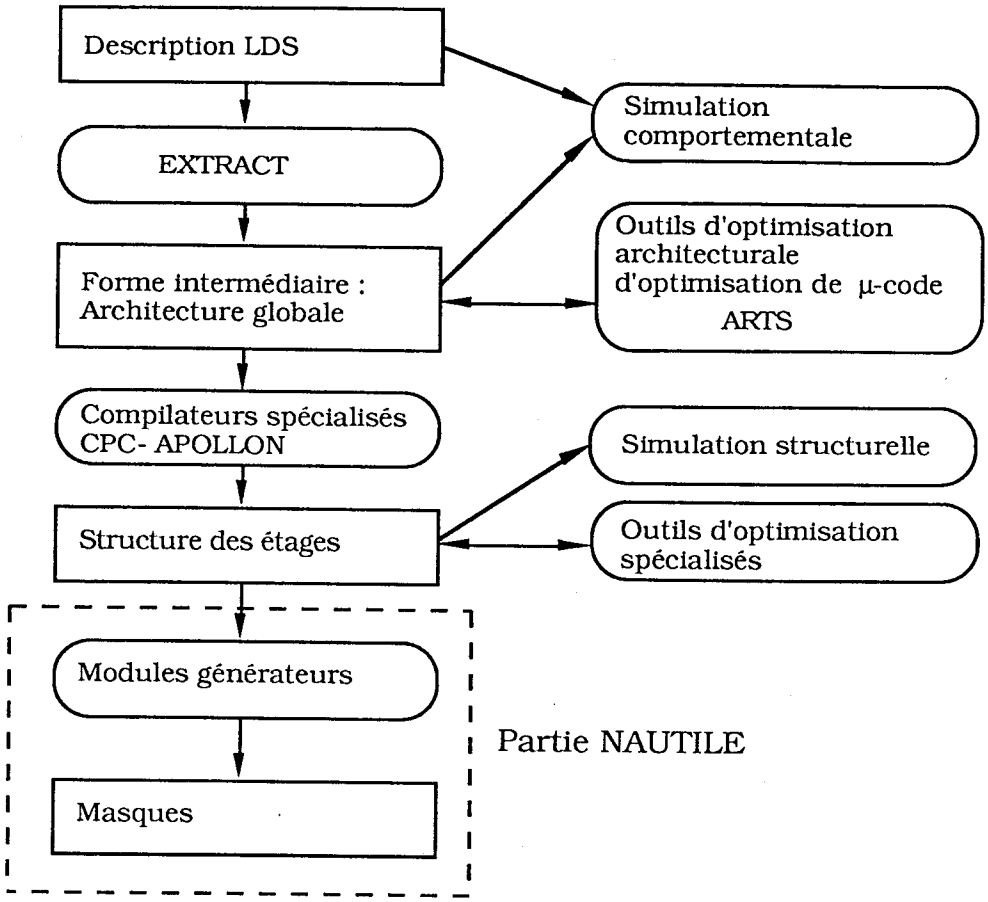



Fig. 5.7: Schéma de fonctionnement du compilateur SYCO

NAUTILE constitue donc la charnière entre les outils de conception à niveau élevé constituant le cœur de SYCO, et la génération du dessin des masques.

## 5.6. Les limitations de NAUTILE

Le lecteur peut penser à la lecture de cette présentation du système NAUTILE, que celui-ci souffre de nombreuses et sévères contraintes.
Nous allons maintenant détailler les contraintes essentielles introduites par l'utilisation d'un système tel que NAUTILE, en nous attachant à montrer en quoi ces contraintes n'en sont en fait pas pour un système intégré d'aide à la compilation comportementale.

1) Hiérarchie unique pour les différentes représentations. Cette limitation peut être trés génante si on travaille avec des représentations des circuits à un niveau de description élevé. En effet, il est par exemple certain que la hiérarchie pour une vue fonctionnelle sera différente de la

hiérarchie pour une vue géométrique: un fil ou un contact n'ont évidemment aucun équivalent fonctionnel. Cependant, si on travaille à des niveaux de description relativement bas, et avec un minimum de discipline, il est relativement aisé d'avoir une seule hiérarchie pour toutes les vues. Ceci nous l'avons vu, est le garant de la cohérence entre les différentes vues.

2) Limitations inhérentes aux générateurs de modules: par expérience, ceux-ci ne se comportent de façon acceptable qu'avec des cellules soit petites (i.e. avec moins de 100 transistors), soit avec des facteurs de répétitions élevés. On n'utilisera donc pas ces générateurs pour dès cellules à optimiser, ou comportant un fort pourcentage de logique anarchique. Dans l'exemple d'une RAM (paragraphe 5.7.2), la partie controle ne sera pas générée par un générateur de modules, mais elle sera importée telle quelle en tant que motif.

3) Pas de possibilité d'ajout d'éléments physiques de base dans une cellule, ou de modification directe d'une vue physique sans modifier en parallèle la vue de construction. Ces contraintes sont trés fortes pour un utilisateur habitué à travailler aux niveaux de représentation les plus bas. Cependant elles sont nécessaires si on souhaite que le système puisse continuer à assurer la cohérence entre les différentes vues. D'autre part, dans l'utilisation d'un outil tel que SYCO, l'utilisateur n'aurait pas vraiment de raisons de travailler à un niveau trés bas. Ce sont uniquement des outils d'un niveau supérieur qui auront à dialoguer avec NAUTILE.

4) Pas de mécanisme de gestion des versions: c'est volontairement que nous avons laissé de côté le problème de la gestion des différentes versions des objets. En effet celui-ci ne nous est pas apparu comme étant fondamental pour la validation des idées introduites par NAUTILE, tout au moins pour une phase de prototypage.
De toute façon, si un concepteur souhaite mémoriser son travail à un moment donné, il lui suffira de dupliquer l'objet qu'il souhaite ainsi conserver, en utilisant une commande de duplication disponible sur le système d'exploitation.
En outre une telle simplification permet d'échapper aux décisions et traitements extrêmement complexes qui doivent être pris lors notamment de la suppression de branches de la hiérarchie.

En effet, les approches qui consistent à mémoriser les différentiels entre versions, ou à associer toutes les versions sur une même structure, alourdissent de façon considérable tous les mécanismes de gestion de la hiérarchie [Rougeaux 87].
L'avantage de disposer d'un système de gestion des versions ne nous a pas paru suffisamment intéressant pour doter NAUTILE de tels mécanismes. En fait le besoin d'avoir des versions différentes ne se fait sentir de façon cruciale qu'en mode de travail interactif, ce qui n'est en général pas le cas avec NAUTILE.

5) Pas d'entrée graphique: l'utilisateur ne peut donc pas effectuer de modifications graphiquement. Les seules entités qu'il puisse modifier de manière graphique sont les motifs, en utilisant un éditeur de dessin extérieur à NAUTILE. Cette limitation en est une évidemment pour les concepteurs, mais si on s'en tient uniquement à l'esprit de NAUTILE, il est certain qu'un éditeur graphique ne servirait pas à grand chose. Sauf peut-être à placer de façon relative les cellules les unes par rapport aux autres. De toutes façons il est aussi rapide pour les actions à effectuer à l'aide de NAUTILE de procéder de façon procédurale plutot que graphiquement. Le mode graphique ne se justifie pleinement que lorsqu'on a de nombreuses figures géométriques à dessiner, auquel cas le mode textuel n'est effectivement que de bien piètre utilité.

En fait il s'avère que l'on peut justifier toutes les limitations de NAUTILE par le fait que c'est avant tout un environnement pour la compilation de silicium. De ce fait plusieurs restrictions du système deviennent peu importantes. Ainsi la vérification de la cohérence combinée à une utilisation stricte des primitives de construction suffisent amplement à assumer tous les besoins du système.

## 5.7. Les réalisations

### 5.7.1. Utilisation pour la partie controle du 6502

Le système NAUTILE a été utilisé dans le cadre du projet Sycomore pour réaliser un Générateur de Partie Controle (GPC) [Geronimi 87].
Ce générateur se décompose en deux parties :

- un générateur de PLA en technologie CMOS

- un programme d'assemblage de parties controles (placement routage) à partir d'une bibliothèque de cellules.

Pour générer une partie controle GPC doit générer une bibliothèque de PLA puis va assembler ces cellules en choisissant une architecture type.
Le générateur de PLA peut soit générer une vue environnement décrivant les frontières des cellules ainsi que les connexions entre elles, soit générer directement le dessin des masques: ceci permet d'assembler le circuit et de le tester, la génération du dessin des masques n'étant effectuée qu'en phase finale de la conception. Les deux représentations sont données respectivement dans les figures 5.8 et 5.9:

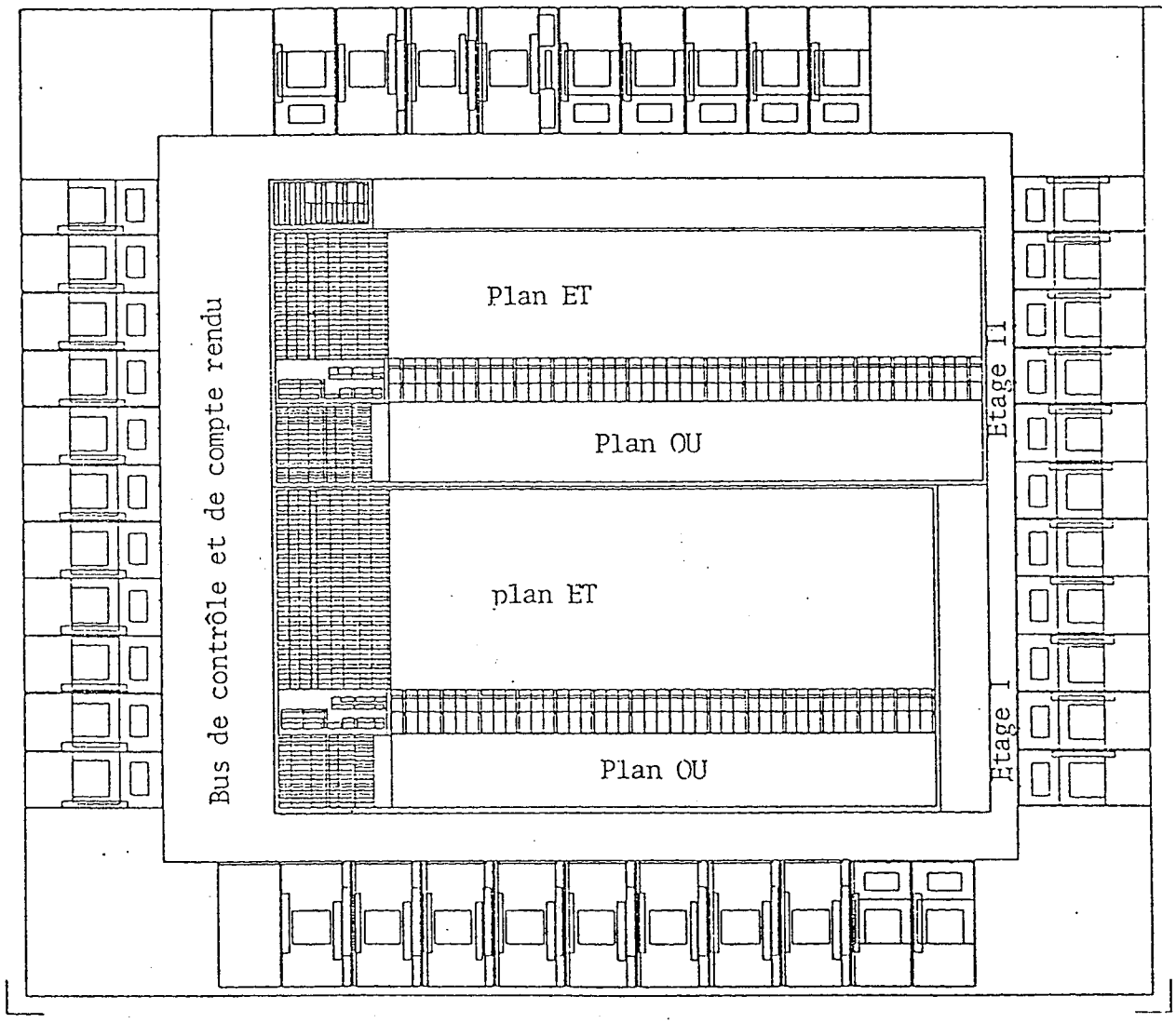



Fig. 5.8: Plan de masse de la partie controle du 6502 [Hornik 89]

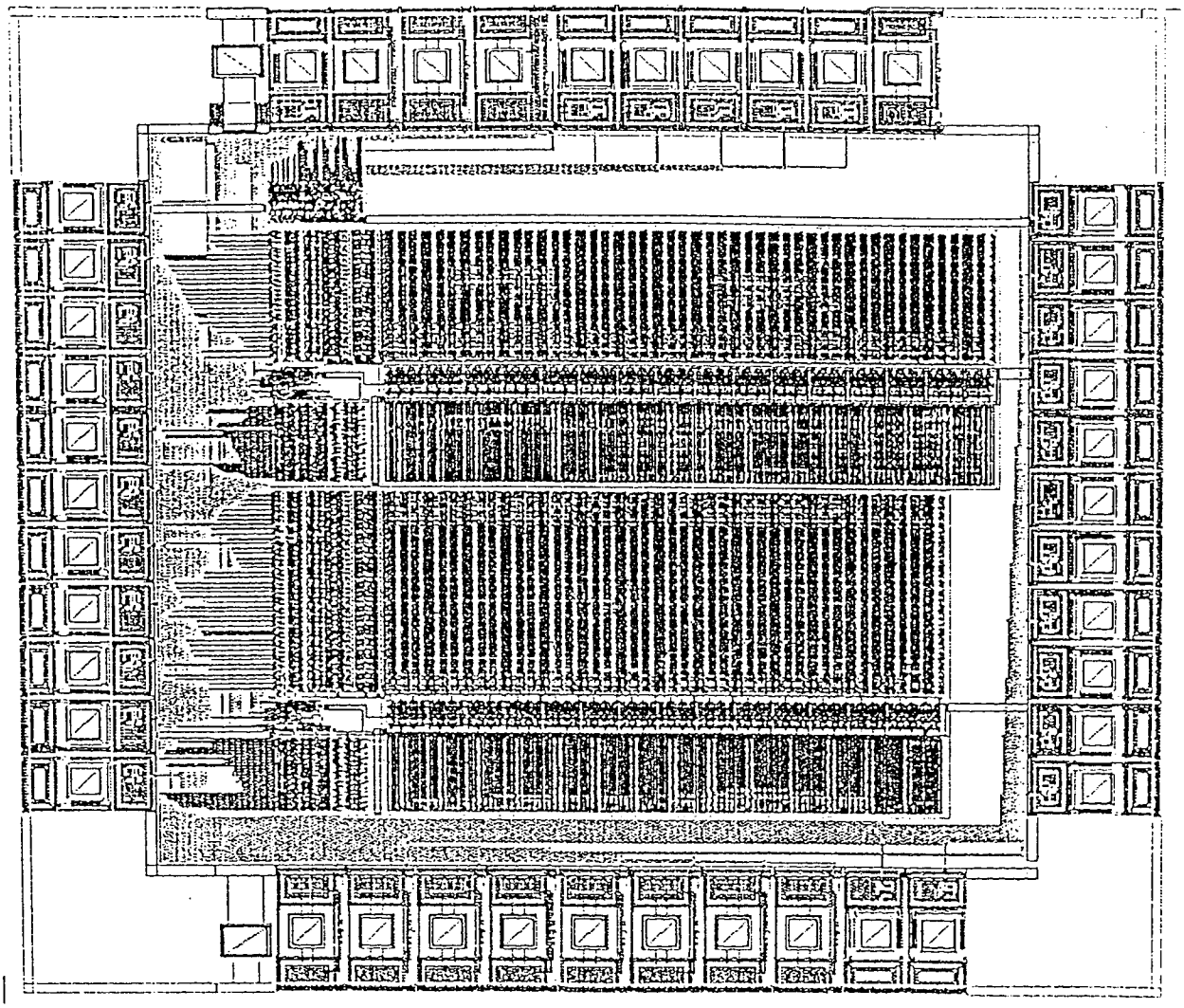


Fig. 5.9: Dessin des masques de la partie controle du 6502 [Geronimi 87]

La partie controle est divisée en plusieurs étages qui seront reliés entre eux par l'intermédiaire d'un bus. Chaque étage est lui même divisé en deux grandes parties, d'une part les entrées/sorties et d'autre part les PLA (matrice OU, Matrice ET).

Le circuit obtenu, est dessiné en technologie CMOS 2µm avec deux niveaux de métal. Il utilise une trentaine de motifs différents (incluant les plots d'entrée/sortie), ainsi que trois sortes de routeurs. Deux de ces routeurs proviennent d'un environnement externe, en l'occurence la société Bull (routeurs de type rivière et canal). Le troisième est un routeur développé à l'aide de NAUTILE (routeur de plots).

La partie contrôle est hiérarchique, chaque tranche (ici au nombre de deux) étant générée par le même générateur. Ce générateur inclut des appels à un routeur, à des fonctions de placement, et à des motifs.

Le texte simplifié du générateur de partie controle est donné dans l'annexe 2.

### 5.7.2. Etude d'un décodeur de RAM

Diverses parties d'une mémoire du type RAM sont actuellement en cours d'étude, dont notamment un décodeur, et des dispositifs d'entrée/sortie.
Ce décodeur est écrit sous la forme d'un générateur paramétré par le nombre de bits d'entrée: il peut donc décrire différentes configurations (le texte du générateur ainsi que différentes représentations du décodeur peuvent être consultés dans les annexes 5 à 9).
Des exemples de décodeur à 3 et à 5 entrées sont donnés ci-dessous, dans les figures 5.10 et 5.11:

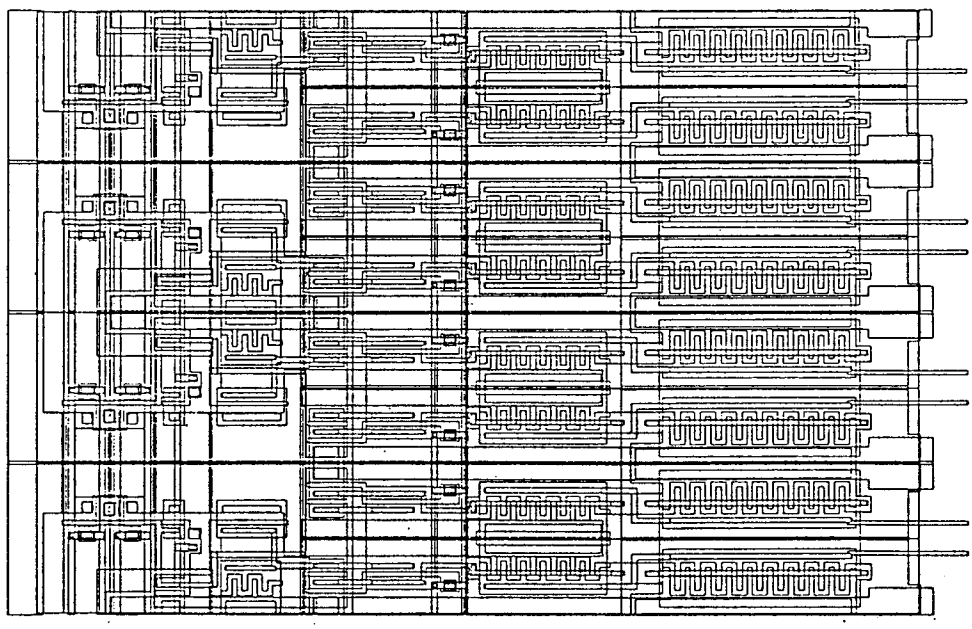


Fig. 5.10: Décodeur à 3 entrées

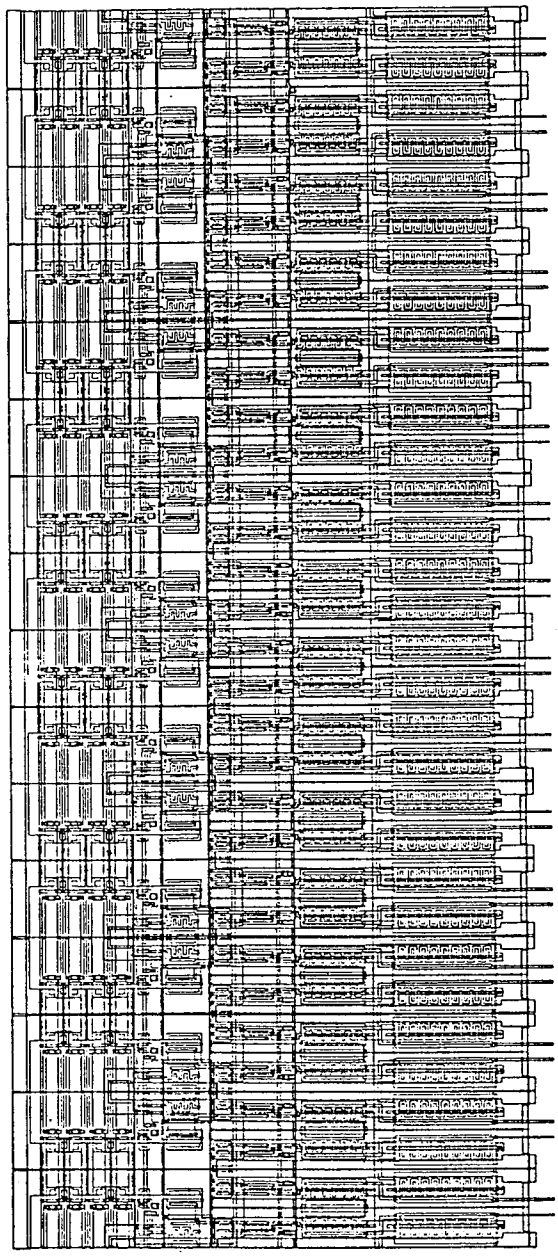


Fig. «#dec5:++fig»: Décodeur à 5 entrées

La forme générale de ce décodeur est donnée dans la figure «#ram»:

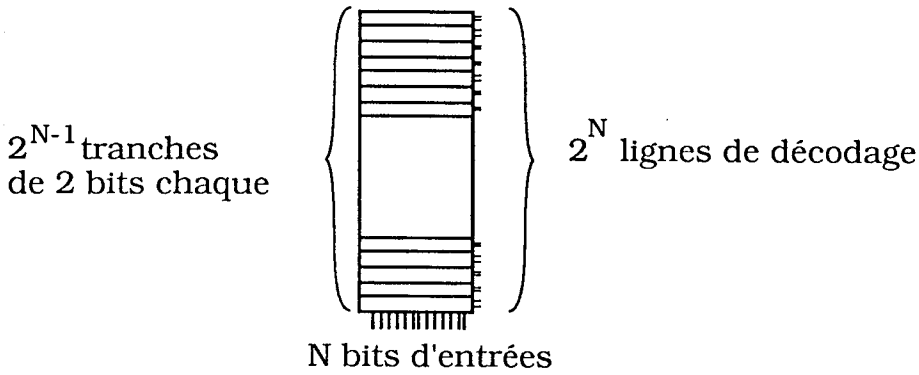



Fig. «#ram:++fig»: Présentation générale d'un décodeur à N entrées

Ce décodeur est composé de 2N-1 étages permettant de décoder les N entrées en $2^{N}$ sorties de sélection. Chaque étage est construit à partir de 7 motifs et permet de décoder 2 sorties de sélection. La figure «#dec» présente l'assemblage des différents motifs produisant un étage du décodeur (il manque sur ce schéma les parties permettant de "personnaliser" chacun des étages du décodeur):

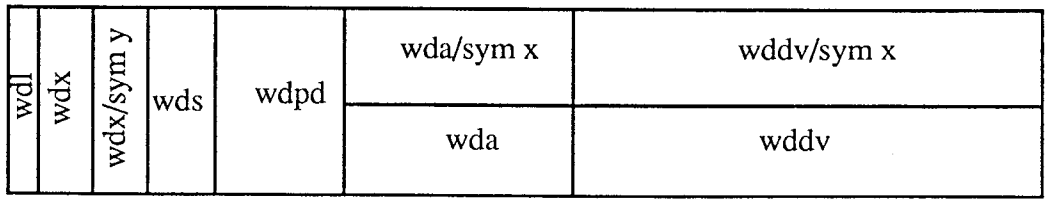



Fig. «#dec:++fig»: Spécification d'une tranche du décodeur

Quelques résultats concernant la réalisation de ce décodeur, ainsi que de la RAM complète sont rassemblés dans les tableaux 5.1 et 5.2:

| Nombre d'Entrées | Temps de Génération | Temps d'Affichage |
|---|---|---|
| 3 | 6s CPU | 9s CPU |
| 5 | 180s CPU | 80s CPU |

Table 5.1: Résultats obtenus pour les décodeurs à 3 et 5 entrées

| | |
|---|---|
| Nombre d'entrées du décodeur: | 7 (soit 128 bits) |
| Nombre de motifs différents: | 12 + la partie controle |
| Nombre total de cellules manipulées: | 8167 |
| Nombre total de rectangles: | 93248 |
| Nombre total de polygones: | 44480 |
| Temps d'affichage avec les 11 niveaux de masque: | 1203s CPU |
| Temps d'affichage avec uniquement les boites: | 125s CPU |

Table 5.2: Résultats obtenus avec la RAM complète

Pour les résultats obtenus avec la RAM complète, il convient de noter que le décodeur n'était plus calculé, mais qu'il avait été généré une fois pour toutes, et qu'on l'avait transformé en motif en utilisant une fonction de mise à plat de la hiérarchie. Ceci nous permet d'éviter des parcours de la hiérarchie par trop pénalisants, et surtout des calculs de positions des différentes instances.

### 5.7.3. Le système NAUTILE en chiffres

Le prototype actuel du système NAUTILE nous a permis d'obtenir un certain nombre de chiffres résumés dans le tableau 5.3:

| | |
|---|---|
| Nombre de fichiers source | 150 |
| Taille totale du code source | ≅ 500K octets |
| Nombre total de lignes de code | 13000 |
| Place mémoire | 1 core ≅ 1M octets |
| Temps de réponse | cf ci-dessus (tableau 5.2) |
| Ergonomie | 2 fenêtres disponibles:<br>• 1 fenêtre textuelle<br>• 1 fenêtre graphique avec menus |

Table 5.3: Quelques précisions concernant NAUTILE

Le core concerne uniquement la place mémoire occupée par le système compilé. Suivant la taille des circuits cette place mémoire peut avoir à augmenter de façon trés importante.

# 6. CONCLUSIONS - PERSPECTIVES

Historiquement, les spécifications du système NAUTILE ont été définies pour répondre aux besoins du compilateur SYCO, afin de permettre de décrire des circuits indépendants de la technologie.
Une approche basée sur une formalisation symbolique non métrique devait être initialement adoptée, afin de combler les lacunes de STYX [Rougeaux 87]. Cependant, au fur et à mesure que le projet a évolué, la nécessité de décrire les circuits suivant différents types de représentation a pris le pas sur toutes les autres considérations. Ceci a donc impliqué, du fait de la diversité des outils manipulant ces représentations, de concentrer nos efforts sur l'obtention d'un système intégré, permettant l'utilisation d'outils et de cellules extérieurs à NAUTILE.

Cette recherche d'intégration s'est accompagnée de l'obligation de maintenir la cohérence entre les données issues de ces différents systèmes, afin de ne pas se trouver dans l'obligation de procéder à des simulations à chaque fois que de nouveaux éléments étaient introduits dans un circuit. A ce jour, aucun système n'a poussé aussi loin que NAUTILE l'étude de la cohérence des représentations, même si certaines des solutions proposées peuvent sembler limitatives. La plupart des systèmes traitant de la cohérence entre les vues se bornent à vérifier a posteriori la correspondance existant entre ces vues, au prix de vérifications trés couteuses et contraignantes.
NAUTILE en revanche, propose une gestion simple et efficace des problèmes liés au maintien de la cohérence, basée essentiellement sur l'utilisation systématique d'une vue de construction, en conjonction avec un ensemble de règles de construction.

Nous avons également vu que NAUTILE permettait de construire des circuits indépendamment de la technologie: dans NAUTILE, les cellules de composition ne dépendent pas de la technologie. Seules les règles de construction et les cellules de base sont liées à la technologie. Le dessin est correct par construction, la vérification est facilitée, car adaptée au type de cellule traitée.

Enfin, on peut se demander comment un outil tel que NAUTILE pourrait évoluer. En effet, de nombreux systèmes similaires existent déja sur le marché, dont des réalisations industrielles qui n'ont rien à envier aux plus brillantes réussites universitaires. Il est certain que NAUTILE ne peut se comparer à un système tel que GDT. Cependant, il est trés bien

adapté à la fonction qui lui a été dévolue, à savoir l'aide à la compilation de silicium. D'ailleurs, une version de NAUTILE écrite en SKILL est actuellement en gestation, qui devrait permettre de concilier à la fois les avantages d'un système industriel confirmé, avec un environnement spécialisé pour la compilation de silicium.
Quant à l'avenir, il s'annonce relativement rose, de nombreux thèmes de recherches pouvant être développés à partir de NAUTILE.

Ainsi y aurait-il beaucoup à faire pour améliorer notamment l'interface avec l'utilisateur. En effet, pour puissante qu'elle soit, l'approche procédurale n'en demande pas moins une connaissance des méthodes de programmation, que bien peu de concepteurs de circuits possèdent. Une approche de définition graphique des générateurs de modules serait nettement plus appropriée et populaire: l'utilisateur ne risquerait plus d'être perturbé par un changement de ses habitudes de travail, son terrain de prédilection restant sans conteste l'environnement graphique. Des recherches dans ce sens sont actuellement en cours en Suède, avec notamment des approches telles que celle proposée par MOVIE [Andersson 89].

Une autre direction de recherche pourrait consister à définir des vues à un niveau plus élevé que ce qui peut actuellement être proposé par NAUTILE. En effet, si on part d'un niveau de description élevé, le schéma de décomposition en une hiérarchie unique n'est plus valable. Il faudrait alors choisir des représentations telles que celles adoptées dans ADAM [Jain 89], ou CADWELL [Daniell 89]. Une suite logique du système actuel pourrait donc être un sur-ensemble de NAUTILE autorisant la description à des niveaux de représentation plus complexes.

Les approches liées à l'intelligence artificielle semblent également trés prometteuses, et nous pensons les appliquer à l'assemblage. Nos recherches actuelles portent sur une formalisation des règles de construction permettant de les traiter par un moteur d'inférence d'ordre zéro [Bondono 90].

Une autre partie de NAUTILE qui gagnerait à être développée concerne l'obtention de la vue environnement. En effet, dans la version actuelle de NAUTILE, les caractéristiques principales de la vue environnement des motifs externes doivent nécessairement être fournies par le concepteur. Il serait intéressant d'écrire des programmes d'extraction automatique de cette vue dans NAUTILE. Cependant, s'il est relativement aisé d'obtenir l'emplacement des objets, leur type et leurs caractéristiques essentielles, il est en revanche beaucoup plus ardu

d'extraire des données concernant leurs performances (délais, consommation, etc...). On pourrait pour cela procéder à des simulations permettant de fournir ces valeurs au système.

# Références bibliographiques

[Anceau 84] F. Anceau: "Génération versus Immersion et Paramétrisation", Note Interne, IMAG/TIM3, mai 1984

[Andersson 89] P. Andersson, L. Philipson: "MOVIE-An Interactive Environment for Silicon Compilation", IEEE Transactions on CAD, Vol. 8, N° 6, juin 1989

[Astap 84] ASTAP Program Reference Manual, Pub. n° SH 20-1118-0, IBM Corp. Data Proc. Div., White Plains, NY 10604.

[Baer 88] J. Baer & Al.: "A Notation for Describing Multiple Views of VLSI Circuits", Proceedings of the 25th Design Automation Conference, 1988

[Baker 80] C. Baker, C. Terman: "Tools for Verifying Integrated Circuits Designs", Lambda, 4th quarter 1980, pp. 22-30

[Bales 82] M.W. Bales: "Layout Rule Spacing of Symbolic Integrated Circuit Artwork", ERL Memo n° UCB/ERL M82/72, University of California, Berkeley, may 1982

[Batali 80] J. Batali & A. Hartheimer: "The Design Procedure Language Manual" Technical Report 598, Massachussets Institute of Technology (sept. 1980)

[Batali 81] J. Batali, N. Bayle, H. Shrobe, G. Sussman & D. Weise: "The DPL/DAEDALUS Design Environment", in VLSI 81: Very Large Scale Integration, J. Gray Ed., New York Academic, pp 183-192, 1981

[Berge 86] J.M. Berge, L. Donzelle, V. Olive, J. Rouillard & D. Rouquier: "Development and Use of Flexible Block Libraries", ESSCIRC 86, pp 28-30

[Bondono 90] P. Bondono, A. Jerraya, A. Hornik, B. Courtois et D. Bonifas: "NAUTILE: a safe environment for silicon compilation", à paraitre EDAC90

[Brayton 88] R. Brayton, R. Camposano, G. De Micheli, R. Otten and J. Eindhoven: "the Yorktown Silicon Compiler", in "Silicon Compilation", D. Gajski ed., Addison-Wesley, 1988

[Brown 83] H. Brown, C. Tong & G. Forster: "Palladio: An Exploratory Environment for Circuit design", Computer pp 41-56, décembre 1983

[Buchanan 80] I. Buchanan: "Modelling And Verification In Structured integrated Circuit Design", PhD Thesis, 1980, Department of Computer Science,University of Edimburgh

[Burich 87] M. Burich: "Design of Module Generators and Silicon Compilers", in "Design Systems for VLSI Circuits" édité par G. De Micheli, Martinus Nijhof Publishers 1987

[Burns 86] J.L. Burns, A.R. Newton: "Sparcs: A New Constraint-Based IC Symbolic Layout spacer", IEEE Custom Integrated Circuits Conference, 1986

[Bushnell 86] M. Bushnell and S. Director: "VLSI CAD Tools Integration using the ULYSSES Environment", Proceedings of the 23d Design Automation Conference, pp. 55-61, 1986

[Bushnell 89] M. Bushnell and S. Director: "Automated Design Tool Execution in the ULYSSES Design Environment", IEEE Transactions on Computer Aided Design, Vol. 8, n° 3, Mars 89, pp. 279-287

[calma] "Calma GDS2", Calma Company, 2901 Tasman drive, Santa Clara, California 95050

[Chailloux 83] J. Chailloux, J.M. Hullot, J.J. Levy, J. Vuillemin: "Le système Lucifer d'aide à la conception de circuits intégrés", Rapport de Recherche INRIA n° 196, mars 1983

[Chailloux 88] J. Chailloux, M. Devin, F. Dupont, J.M. Hullot, B. Serpette, J. Vuillemin: "LeLisp Version 15.22 Le Manuel de Référence", INRIA, Rocquencourt, 1988

[Cheng 88] E.K. Cheng, S. Mazor: "The Genesil Silicon Compiler", in "Silicon Compilation", édité par D. Gajski, Addison-Wesley, pp 361-405, 1988

[Cherry 85] J. Cherry, H. Shrobe & Al: "NS: An Integrated Symbolic Design System", VLSI 1985, Elsevier Science Publishers B.V., IFIP 1986

[Cho 77] Y.E. Cho, A.J. Korenjack, D.E. Stockton: "FLOSS: an approach to automated layout for high-volume designs", Proceedings of the 14th Design Automation Conference, pp. 138-141, 1977

[Chou 86] H. Chou and W. Kim: "A Unifying Framework for Version Control in a CAD Environment", Proceedings of the 12th Conference on Very Large Data Base, Kyoto, aout 1986

[Chou 88] H. Chou and W. Kim: "Versions and Change Notification in an Object Oriented Data Base System", Proceedings of the 25th Design Automation Conference, pp. 275-281, 1988

[Chu 86] A. Chu, J. Fishburn, P. Honeyman, Y. Lien: "A Database Driven VLSI Design System", IEEE Transactions on CAD, Vol. CAD 5, n°1, 1986

[Chuquillanqui 82] J. Chuquillanqui Bernaola and T. Perez Segovia: "PAOLA, a Tool for Topological Optimization of Large PLAs", Proceedings of the 19th Design Automation Conference, 1982

[Clary 80] D. Clary, R. Kirk, S. Sapiro: "SIDS: A Symbolic Interactive Design System", Proceedings of the 17th Design Automation Conference, pp. 292-295, 1980

[Corbin 88] V. Corbin: "The Concorde Silicon Compiler", in "Silicon Compilation", édité par D. Gajski, Addison-Wesley, 1988

[Daniell 89] J. Daniell and S. Director: "An Object Oriented Approach to CAD Control within a Design Framework", Proceedings of the 26th Design Automation Conference, pp. 197-202, 1989

[De Michelli 87] G. De Michelli, A. Sangiovanni-Vincentelli, P. Antognetti: "Design systems for VLSI Circuits", Martinus Nijhoff Publishers, 1987

[Director 81] S.W. Director, A. Parker, D.P. Sieworek, D.E. Thomas: "A Design Methodology and Computer Aids for Digital VLSI Systems", IEEE Transactions on circuits & systems, vol. CAS-28, n° 7, juillet 1981

[Doasch 87] R. Doasch & R. Fowler: "RNL Advanced Topics", University of Washington, Northwest LIS Release 3.1, 1987

[Edif 87] Electronic Design Interface Format, Version 2.0.0, Edif Steering Committee, may 1987

[Gajski 86]: D. Gajski & Al: "Silicon Compilation (Tutorial)", IEEE Custom Integrated Circuit Conference, 1986, pp 102-110

[Gajski 88] D.J. Gajski editor: "Silicon Compilation", Addison Wesley Publishing Company, 1988

[Geronimi 87] J.P. Geronimi et A.A. Jerraya: "Architecture des Parties Controles Générées par SYCO: GENTOPO", Rapport Interne, IMAG/TIM3, 1987

[Gibson 76] D. Gibson & S. Nance: "SLIC- Symbolic Layout of Integrated Circuits", Proceedings of the 13th Design Automation Conference, pp. 434-440, 1976

[Goldberg 83] A. Goldberg et D. Robson: "SMALLTALK 80, The Language and its Implementation", Addison Wesley, Palo-Alto, 1983

[Harrison 86] D. Harrison, P. Moore, R. Spickelmier, AR. Newton: "Data Management and Graphics Editing in the Berkeley Design Environment", International Conference on Computer Aided design, pp 24-27, 1986

[Hill 89] D. Hill, D. Shugard, J. Fishburn, K. Keutzer: "Algorithms and Techniques for VLSI Layout Synthesis", Kluwer Academic Publishers, 1989

[Hornik 89] A. Hornik: "Contribution à la définition de NAUTILE", Thèse de Docteur de l'INPG, 1989

[Hsueh 79] M.Y. Hsueh & D.O. Pederson: "Computer-Aided Layout of LSI Circuit Building-Blocks", Proceedings of the 1979 IEEE International Symposium on Circuits and Systems, pp. 474-477

[Hullot 84] J.M. Hullot: "Programmer en CEYX", INRIA, Rocquencourt, 1984

[Jain 89] R. Jain, K. Küçükçakar, M.L. Mlinar et A. Parker: "Experience with the ADAM Synthesis System", Proceedings of the 26th Design Automation Conference, pp. 56-61,1989

[Jamier 85] R. Jamier, A.A. Jerraya: "A Datapath Compiler", Proceedings of the International Conference on Circuits Design, 1985

[Jerraya 85] Jerraya, Rosier, Rougeaux, Courtois: "A Hierarchical Symbolic Layout Tool: Styx", VLSI 1985, Elsevier Science Publishers B.V., IFIP 1986

[Jerraya 86] A. Jerraya, P. Varinot, R. Jamier, B. Courtois: "Principles of The Syco Compiler", Proceedings of the 23rd Design Automation Conference,1986

[Jerraya 89] A. Jerraya: "Participation à la Compilation de Silicium et au Compilateur SYCO", thèse d'état, IMAG/TIM3, 1989

[Johannsen 79] D. Johannsen: "Bristle Blocks: a Silicon Compiler", Caltech Conference on VLSI, janvier 1979

[Johnson 82] S. Johnson: "Hierarchical Design Validation Based on Rectangles", Proceedings of the Conference on Advanced Research in VLSI, MIT, pp. 97-100, janvier 1982

[Jullien 86] C. Jullien, A. Leblond, J. Lecourvoisier: "A Database Interface for an Integrated CAD System" Proceedings of the 23d Design Automation Conference, pp. 760-767, 1986

[Karplus 82] K. Karplus: "CHISEL, an extension to the programming language C for VLSI layout", Ph.D Thesis, Dept. of Computer Science, Stanford University, Stanford, CA., 1982

[Katz 83] R. Katz: "Managing the Chip Design Data Base", Computer, Dec. 83, pp. 26-35

[Katz 86] R.H. Katz, M. Anwarrudin, E. Chang: "A Version Server for Computer Aided Design Data", Proceedings of the 23d Design Automation Conference, 1986

[Keller 82] K.H. Keller, A.R. Newton, S. Ellis: "A Symbolic Design System for Integrated Circuits", Proceedings of the 19th Design Automation Conference, pp. 460-466, 1982

[Knapp 83] D. Knapp, J. Granacki, A. Parker: "An Expert Synthesis System", Proceedings of the International Conference on Computer Aided Design, pp. 164-165, 1983

[Knapp 85] D. Knapp and A. Parker: "A Unified Representation for Design Information", Proceedings of the CHDL 85, Elsevier, 1985

[Lambert 81] D.R. Lambert: "Graphic Language One, IBM Corporate Wide Physical Design Data Format", Proceedings of the 18th Design Automation Conference, pp. 713-719, 1981

[Larsen 78] R.P. Larsen: "Versatile Mask Generation Techniques for Custom Microelectronic Devices", Proceedings of the 15th Design Automation Conference, pp. 193-198, 1978

[Law 87] H.S. Law, G. Wood: "A Mixed-Media Approach to Module Generator Design", in "Design Systems for VLSI Circuits" édité par G. De Micheli, Martinus Nijhof Publishers 1987

[Lipset 89] R. Lipset, C. Schaefer, C. Ussery: "VHDL: Hardware Description Design", Kluwer Academic Publishers, 1989

[Lipton 82] R.J. Lipton, S. Sedgewick, R. Valdes, S. North, G. Vijayan: "ALI: a Procedural Language to describe VLSI Layouts", Proceedings of the 19th Design Automation Conference, pp. 467-474, 1982

[Mead 80] C. Mead, L. Conway: "Introduction to VLSI systems", Addison-Wesley Publishing Company, 1980.

[Meyer 87] E.L. meyer: "On Module Generation", VLSI Systems Design, pp. 48-63, mars 1987

[Moore 86] P.P. Moore: "Oct: Database Programmer's Manual", University of California Berkeley, janvier 1986

[Nautile 89] Manuel d'utilisation du système Nautile, version 3.0, IMAG/TIM3 et Laboratoire IBM de Corbeil-Essonnes, 1989

[Newton 87] A.R. Newton: "Symbolic Layout and Procedural Design", in "Design Systems for VLSI Circuits" édité par G. De Micheli, Martinus Nijhof Publishers 1987

[Notrott 87] R.W. Notrott & H. Koeman: "Netlist & RNL tutorial for beginners", University of Washington, Northwest LIS Release 3.1, 1987

[Ousterhout 81] J. Ousterhout: "Caesar: an Interactive Editor for VLSI Layouts", VLSI Design II-4, pp 34-38, 1981

[Ousterhout 84] J. Ousterhout & Al: "Magic: A VLSI Layout System", Proceedings of the 21st Design Automation Conference, pp. 152-159, 1984

[Paillotin 85] J.F. Paillotin: "Le Système Lucie", version du 1er juillet 1985, TIM3-INPG laboratoire IMAG

[Rabaey 88]: J. Rabaey, H. De Man, J. Van Hoof, G. Goossens, and F. Catthoor: "CATHEDRAL II a Synthesis System for Multiprocessor DSP", in "Silicon Compilation", D. Gajski ed., Addison-Wesley, pp 311-360, 1988

[Rougeaux 87] F.R. Rougeaux: "Outils de CAO et Conception Structurée de Systèmes Intégrés sur Silicium", Thèse de Docteur de l'INPG, 1987

[Rubin 87] S.M. Rubin: "Computer Aids for VLSI Design", Addison-Wesley Publishing Company, 1987

[Savaria 88] Y. Savaria: "Conception et vérification des circuits VLSI", éditions de l'Ecole Polytechnique de Montréal, 1988

[Schoellkopf 83] J.P. Schoellkopf: "Lubrick, a Silicon Assembler and its Application to Data-path Design for FISC", in VLSI 83, ESPBV ed., 1983 pp 435-445

[Sequin 83] C.H. Sequin: "Managing VLSI Complexity: an Outlook", Proc. IEEE, vol. 71, N° 10, Janvier 1983.

[Silva 89] M. Silva, D. Gedye, R. Katz, R. Newton: "Protection and Versioning for OCT", Proceedings of the 26th Design Automation Conference, pp. 264-269, 1989

[Southard 88] J.R. Southard: "Algorithmic System Compilation: Silicon Compilation for System Designers", in "Silicon Compilation", édité par D. Gajski, Addison-Wesley, 1988

[Sproull 80] R.F. Sproull, R.F. Lyon: "The Caltech Intermediate Form For LSI Layout Description", in C. Mead & L. Conway, Introduction to VLSI Systems, Addison Wesley, 1980, pp 115-127

[Trimberger 84] S. Trimberger: "VTI compose, a Powerful Graphical Chip assembly Tool", Proceedings of the 21st Design Automation Conference, pp. 697-698, 1984

[Vhdl 87] VHDL Tutorial for IEEE Standard 1076 VHDL, may 1987

[Weste 81] N. Weste: "Virtual Grid Symbolic Layout", Proceedings of the 18th Design Automation Conference, pp. 225-233, 1981

[Williams 77] J.D. Williams: "STICKS: A New Approach to LSI Design", MIT MSEE Thesis, 1977

[Winslett 89] M. Winslett, D. Knapp, K. Hall, G. Wiedehold: "Use of Change Coordination in an Information-Rich Design Environment", Proceedings of the 26th Design Automation Conference, pp. 252-257, 1989

[Wolf 87] W. Wolf: "Symbolic Layout and Compaction for Silicon Compilation", in "Design Systems for VLSI Circuits" édité par G. De Micheli, Martinus Nijhof Publishers 1987

## ANNEXE 1: exemple de .nautilerc

```
;pas de nom de cellules jusqu'au niveau 6:
(interdit 0 1 2 3 4 5 6)
;définition des différents niveaux de masque, avec leurs couleurs associées
(ajoute-niv 'M1 "red")
(ajoute-niv 'M2 "navyblue")
(ajoute-niv 'P1 "blue")
(ajoute-niv 'PV "green")
(ajoute-niv 'NV "orchid")
(ajoute-niv 'RX "khaki")
(ajoute-niv 'BP "darkorchid")
(ajoute-niv 'BN "orange")
(ajoute-niv 'BF "cyan")
(ajoute-niv 'BC "plum")
(ajoute-niv 'TN "aquamarine")
(ajoute-niv 'NW "wheat")
(ajoute-niv 'CA "gold")
(ajoute-niv 'OUTLIN1 "black")
;définition de la couleur noir (pour les boites et le texte)
(defvar noir (make-named-color "black"))
```

## ANNEXE 2: Générateur de Partie Controle de SYCO

Il s'agit ici de la version simplifiée d'un générateur de module d'un étage quelconque de la partie controle appelé GPC, tiré de [Geronimi 87].

```
(de Etage-Pc (nom Numero-etage nom-cell)

 (defcell nom-cell)
 ;Définition de la cellule contenant l'étage courant

 (acces nom-cell)
 ; Ouverture de la cellule

 (gen-entree nom Numero-etage 'ent)
 ; Génération de la cellule contenant les entrées de l'étage

 (gen-sortie nom Numero-etage 'sort)
 ; Génération de la cellule contenant les sorties de l'étage

 (gen-ampli nom Numero-etage 'ampli)
 ; Génération de la cellule contenant les amplis de l'étage

 (gen-precharge nom Numero-etage 'pre)
 ; Génération de la cellule contenant les précharges de l'étage

 (gen-interface nom Numero-etage 'inte)
 ; Génération de la cellule contenant les interfaces de l'étage

 (gen-pla nom Numero-etage 'plaet 'plaou)
 ; Génération des PLA ET et OU de l'étage et sauvegarde en
 ;bibliothèque

 (char-lib LUBRICK 'sync 'synch)
 ; Chargement d'une cellule de synchronisation depuis la bibliothèque
 ; LUBRICK

  ;********** Assemblage du PLA *******

 (defcell 'pla)
```

```
(acces 'pla)
(instance-C 'plaou 'plaou1 '(0 . 0) ())
;Placement de "plaou" en x=0, y=0

(direct-C 'Nord
        (direct 'Nord 'plaou1 'int ())
       'plaet)

;On place "int" et "plaet" au dessus de "plaou"
;Connexions par aboutement
;placement des amplis et précharges ...
...
(fin-acces 'pla)

 ;********** Assemblage des blocs d'entrées/sorties ********

(defcell 'ent/sort)
(acces 'sort)
(instance-C 'sort 'sort1 '(0 . 0) ())
; placement de sort en x=0, y=0
;sort1 est le nom de l'instance créée de sort
(direct-C 'Nord
        (direct 'Nord 'sort1 'sync ())
       'ent)
;On place "sync" et "ent" au dessus de "plaou"

(fin-acces 'ent/sort)

 ;********** Assemblage de l'étage **********

(instance-C 'ent/sort 'e-s1 '(0 . 0) ())
(dev 'Est 'e-s1 'pla)
;placement et routage mono-couche entre les entrées sorties et le PLA
(fin-acces))
```

Pour la génération de la partie controle dans son intégralité il faudra assembler les différents étages entre eux par l'intermédiaire d'un bus.

# ANNEXE 3: La structure de données Nautile

```
*************************************************************************
*                        STRUCTURE NAUTILE                              *
*************************************************************************

(deftclass :STRUCT   Ldir          ;~(List string)
                     Lhier         ;liste des hierarchies en memoire
                     Cell-cour     ;nom de la cellule courante
                     Mode          ;mode de travail
                     Dessin)       ;ensemble des variables de dessin

(defmake {STRUCT} :mkstruct (Ldir Lhier Cell-cour Mode Dessin))

(deftclass :Ens-Var  Coefx         ;echelle en x
                     Coefy         ;echelle en y
                     Affichable    ;A-liste des niveaux avec leur couleur
                     Profondeur    ;profondeur d'affichage de la hierarchie
                     Interdit)     ;niveau textuel non affichable

(defmake {Ens-Var} :mkvar (Coefx Coefy Profondeur Affichable Interdit))


(deftclass :CELL     Nom           ;~symbol
                     Etat          ;~ETAT
                     Env           ;~ENVIRONNEMENT
                     Corps)        ;~(List VUE))

(defmake {CELL} :mkcell (Nom Etat Env Corps))

(deftclass {CELL}:CELLULE  Filles      ;~(List symbol)
                           Const)      ;~CONSTRUCTION

(deftclass {CELL}:MOTIF)

(defmake {CELLULE} :mkcellule (Nom Etat Env Corps Filles Const))

(defmake {MOTIF} :mkmotif (Nom Etat Env Corps))
```

```
(deftclass :ETAT      Date_Creat          ;~DATE
                      Date_Modif    ;~DATE
                      Origine             ;~string
                      Coherence)          ;~List)

(defmake {ETAT}    :mketat (Date_Creat Date_Modif Origine Coherence))

(deftclass :VUE       Etat)   ;~ETAT)

(defmake {VUE} :mkvue (Etat))

(deftclass {VUE}:CONSTRUCTION   Lappel)       ;~(List APPEL))

(defmake {CONSTRUCTION} :mkconst (Etat Lappel))

(deftclass {VUE}:ENVIRONNEMENT        Connects      ;~(List CONNECT)
                                      Lparam        ; ~(List)
                                      Boite         ;~Rect
                                      Icone )       ;~ICONE)

(defmake {ENVIRONNEMENT} :mkext (Etat Connects Lparam Boite Icone))

(deftclass {VUE}:PHYSIQUE  Type_vue); de type 'elec, 'topo, etc...

(defmake {PHYSIQUE} :mkphys (Etat Type_vue))

(deftclass {PHYSIQUE}:EXTERNE   Nom_fichier ;~string
                                Origine)      ;~string)

(defmake {EXTERNE} :mkphysext (Etat Type_vue Nom_fichier Origine))

(deftclass {PHYSIQUE}:INTERNE   Liste)  ;~(list))

(defmake {INTERNE} :mkphysint (Etat Type_vue Liste))

(deftclass :APPEL )

(deftclass {APPEL}:AP_CELL        Position      ;~Coord
```

```
                    Nom-Def     ;~symbol
                    Nom-Inst    ;~string
                    Transfo     ;entier entre 0 et 7
                    Repet)      ;du type repetition

(defmake {AP_CELL} :mkap_cell (Position Nom-Def Nom-Inst Transfo Repet))

(deftclass :REPETITION    Direction
                          Quantite;     nombre de repetitions
                          DecalageX;
                          DecalageY);

(defmake {REPETITION} :mkrepet (Direction Quantite DecalageX DecalageY))

(deftclass {APPEL}:AP_FANTOME  Position    ;~Coord
                               Nom-Def     ;~symbol
                               Nom-Inst    ;~string
                               Ltransf)    ;~(List))

(deftclass {APPEL}:AP_OUTIL    Nom         ;~symbol
                               Lparam)     ; ~(List))

(defmake {AP_OUTIL} :mkoutil (Nom Lparam))

(deftclass {AP_OUTIL}:ROUTEUR      Dir         ;~string
                                   Point       ;~Coord
                                   Appel       ;~APPEL
                                   Decal       ;~fix
                                   Lconn       ;~(List CONNECT)
                                   Lobst )     ;~(List OBST))

(defmake {ROUTEUR} :mkrouteur (Nom Lparam Dir Point Appel Decal Lconn Lobst))

(deftclass :CONNECT    Nom         ;~string
                       Occupe      ;~cons
                       Equi        ;~string
                       Type)       ;~string
```

```
;definition des connecteurs Normaux:
(deftclass {CONNECT}:CONN-N     Dir    ;      ~string
                                Niveau         ;~string
                                Position       ;~Coord
                                Taille)        ;~fix

;definition des connecteurs Remontes:
(deftclass {CONNECT}:CONN-R     N-inst         ;~string
                                N-connect      ;~string
                                Num)           ;numero de l'instance (repet.)

(defmake {CONNECT} :mkconnect (Nom Occupe Equi Type))

(defmake {CONN-N} :mkconn-n (Nom Occupe Equi Type Dir Niveau Position Taille))

(defmake {CONN-R} :mkconn-r (Nom Occupe Equi Type N-inst N-connect Num))

;****************** LES VUES PHYSIQUES ************************

(deftclass :POLYGONE        poly
                            niv)

(defmake {POLYGONE}   :mkpoly (poly niv))

(deftclass :RECTANGLE               rect ;~Rect
                                    niv  );~string

(defmake {RECTANGLE}   :mkrect (rect niv))

(deftclass :TRANSISTOR              type
                                    grille
                                    source
                                    drain
                                    position
                                    w
                                    l)

(defmake {TRANSISTOR}   :mktrans (type grille source drain position w l))
```

```
(deftclass :NOEUD     nom
                      capacite)

(defmake {NOEUD}      :mknode (nom capacite))

(deftclass :CAPACITE          capacite
                              noeud1
                              noeud2)

(defmake  {CAPACITE}          :mkcapa (capacite noeud1 noeud2))

(deftclass :RESISTANCE        resistance
                              noeud1
                              noeud2)

(defmake  {RESISTANCE}        :mkres (resistance noeud1 noeud2))

;***************** LES CONNECTEURS SIMPLIFIES *****************

(deftclass :TERM      coord-term    ;~Coord
                      equ-term      ;~string
                      Taille        ;~fix
                      Niveau        ;~string
                      Type          );~string)

(defmake {TERM} :mkterm (coord-term equ-term Taille Niveau Type))

(synonymq  {TERM}:Position  {TERM}:coord-term)

(synonymq  {TERM}:Equi  {TERM}:equ-term)

;*************** LES DIFFERENTES  VUES ********************

(deftclass :ENV)
(defmake {ENV} ENV ())

(deftclass :ASSEMB)
(defmake {ASSEMB} ASSEMB ())
```

```
(deftclass :TOPO)
(defmake {TOPO} TOPO ())

(deftclass :ELECT)
(defmake {ELECT} ELECT ())

(deftclass :LOGIC)
(defmake {LOGIC} LOGIC ())

(deftclass :TEMPO)
(defmake {TEMPO} TEMPO ())

;*************** LES DIFFERENTS  SYSTEMES ********************

(deftclass :NAUTILE)
(defmake {NAUTILE} NAUTILE ())

(deftclass :LUBRICK)
(defmake {LUBRICK} LUBRICK ())

(deftclass :LUCIE)
(defmake {LUCIE} LUCIE ())

(deftclass :STYX)
(defmake {STYX} STYX ())

(deftclass :RNL)
(defmake {RNL} RNL ())

(deftclass :ASTAP)
(defmake {ASTAP} ASTAP ())

(deftclass :GDSII)
(defmake {GDSII} GDSII ())

(deftclass :GL1)
(defmake {GL1} GL1 ())

(deftclass :HADES)
```

```
(defmake {HADES} HADES ())

(deftclass :SPICE)
(defmake {SPICE} SPICE ())
```

## ANNEXE 4: Les règles de construction

Il y a deux types de règles à prendre en compte: les règles fonctionnelles (qui si elles sont violées conduisent à ne plus verifier la fonctionnalité), et des règles dues aux délais à respecter: ceux-ci pouvant se dégrader au fur et à mesure. D'autre part, certaines règles sont indépendantes de la technologie utilisée, alors que d'autres sont étroitement liées avec celle-ci. On peut donc distinguer quatre cas:

1) règles fonctionnelles liées à la technologie:

- ne pas relier les sorties de différentes portes CMOS ensemble. Ceci peut etre autorisé dans certains cas en bipolaire, mais on limitera alors le nombre de sorties reliées les unes aux autres.

- vérifications du "bon usage" des transistors: ainsi en CMOS les transistors de type P conduisent à la masse, alors que les transistors de type N conduisent à l'alimentation. En NMOS un transistor déplété dont le drain n'est pas relié à l'alimentation, ou dont la grille n'est pas connectée à la source constitue certainement une erreur. En revanche il est trés rare de trouver des transistors dont la grille est connectée à la source ou au drain autrement qu'en NMOS. Enfin des capacités obtenues en reliant la source et le drain d'un transistor sont rarement volontaires.

2) règles de délais et de dégradation des signaux liées à la technologie:

- typiquement les règles liées à l'entrance et à la sortance: en bipolaire l'entrance est limitée strictement, en CMOS ce n'est pas forcément le cas mais une sortance élevée peut entrainer des degradations au niveau des temps de reponse (alors qu'en bipolaire ce sont les signaux qui se dégradent)

- également le cas des portes de transmission ("transfert gates") dont la quantité que l'on peut mettre en série est limitée

- en MOS, si on relie la sortie d'un interrupteur à la grille d'un autre interrupteur, on assistera à des dégradations du signal.

3) règles fonctionnelles non liées à la technologie:

- aucun chemin direct ne doit mener de l'alimentation à la masse

- si un seul transistor sépare la masse de l'alimentation, il y aura un court-circuit dés que celui-ci sera passant: une telle configuration est donc interdite

- un transistor doit posséder au moins une possibilite de connexion (ceci n'est pas forcément vrai si on souhaite utiliser des transistors d'isolation).

- un transistor dont la sortie ne conduit nulle part et qui n'est pas déclaré comme sortie du circuit est certainement erroné.

- l'entrée d'un transistor d'échantillonnage ne peut etre reliée qu'à la sortie d'une horloge, ou d'un élément ayant validé une horloge

- la sortie d'un transistor d'échantillonnage ne peut etre reliée qu'à l'entrée d'une porte logique à restauration.

- la sortie d'un transistor d'échantillonnage ne peut pas être reliée à la sortie d'un autre transistor d'échantillonnage (même en bipolaire).

- chaque noeud du circuit doit avoir des chemins qui le conduisent à la masse et à l'alimentation, ou être relié à une entrée du circuit.

- la sortie d'un circuit séquentiel peut attaquer un autre circuit séquentiel si l'horloge qui régle le passage d'une porte à l'autre n'est pas une horloge du premier circuit.

- la sortie d'un circuit séquentiel peut servir à générer l'horloge d'un autre circuit séquentiel si le resultat obtenu ne sert pas d'horloge au premier circuit.

4) règles concernant les dégradations non dépendantes de la technologie

- on ne peut associer à l'entrée d'une même porte logique les deux phases d'une même horloge, sous peine d'occasioner des problèmes de charges.

- si la capacité de sortie d'un petit transistor (i.e. de grande résistance) est importante, les délais risquent d'être élevés.

Remarques:

- en bipolaire l'entrance (aussi bien que la sortance) d'une cellule dépend de ce à quoi elle est connectée: ainsi suivant les éléments en entrée, la cellule pourra avoir plus ou moins d'entrance.

- on associe en bipolaire à chaque porte dont la sortie est reliée à la sortie d'una autre porte un générateur de courant, afin d'améliorer l'immunité au bruit.

- Les règles de construction sont des règles internes au circuit: on ne traite pas des règles externes qui apparemment sont assez difficiles à modéliser.

- les horloges doivent etre non recouvrantes, afin d'éviter les conflits possibles.

## ANNEXE 5: Un générateur de décodeurs n bits

```
(de decodeur (n . nom)
 (when (< 1 n)
       (defcell (if nom (car nom) (catenate "decodeur" n)))
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwdl.til t)
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwdx.til t)
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwdpd.til t)
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwda.til t)
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwddv.til t)
       (loadfile '/u/nautile/bondono/nautile/BIBLI/NAUTILE/nwds.til t)
       (lets ((b (boite 'wdx))
              j nom liste-nom
              (larg (sendq width b))
              (haut (sendq height b)))
             (defmotif 'contact)
             (addrect 'PV 0 0 8 14)
             (fin-acces)
             (defcell 'contact-wd)
             (instance 'contact () 19 222)
             (boite 'contact-wd (sendq xorg b) (sendq yorg b) larg haut)
             (fin-acces)
;****************** définition du multiplexeur en A *****************
             (defcell 'double-contact)
             (let ((b (boite 'double-contact 0 0
                                (+ (sendq width (boite 'wds))
                                   (sendq width (boite 'wdpd))
                                   (sendq width (boite 'wda)))
                                (sendq height (boite 'wds)))))
                  (instance 'contact () (- (sendq xext b) 7)
                                        (- (sendq yext b) 89))
                  (instance 'contact () (- (sendq xext b) 93) 75))
             (fin-acces)
;****************** définition de la partie droite ******************
             (defcell 'partie-droite)
             (direct 'sud
                     (direct 'est
                             (direct 'nord
                                     (direct-multiple
```

```
                                'est (instance 'wds ()
                                                0
                                                 (sendq height
                                                        (sendq boite 'wds))
                                                 '((sym x)))
                                '(wdpd ((sym x)))
                                '(wda ((sym x))))
                               'wda)
                        'wddv)
                'wddv '((sym x)))
        (fin-acces)
        (defcell 'temp)
        (fin-acces)
;******************** specification du decodeur **********************
        (instance 'temp (setq nom (string (gensym))) 0 0)
        (for (i 0 1 (1- (2** (1- n))))
                (setq j i)
                (if (evenp i)
                  (prog2
                      (apply 'direct-multiple
                          'est
                           (setq nom (direct 'nord nom 'wdl))
                          (progn
                              (setq liste-nom '(double-contact))
                              (for (h 0 1 (- n 2))
                                  (setq j (quomod j 2))
                                  (setq liste-nom
                                          (cons
                                           (if (equal #:ex:mod 0)
                                                'contact-wd
                                                (list 'contact-wd '((sym y))))
                                           liste-nom)))
                              liste-nom))
                      (apply 'direct-multiple
                          'est
                          nom
                          (progn
                              (setq liste-nom '((partie-droite ((sym x)))))
                              (for (h 0 1 (- n 2))
```

```
                    (setq liste-nom
                          (cons
                           (if (oddp h)
                               (list 'wdx '((sym x)))
                               (list 'wdx '((rot 180))))
                           liste-nom)))
                liste-nom)))
          (apply 'direct-multiple
              'est
               (setq nom (direct 'nord nom 'wdl '((sym x))))
              (progn
                  (setq liste-nom '(double-contact))
                  (for (h 0 1 (- n 2))
                      (setq j (quomod j 2))
                      (setq liste-nom
                            (cons
                             (if (equal #:ex:mod 0)
                                 (list 'contact-wd '((sym x)))
                                 (list 'contact-wd '((rot 180))))
                             liste-nom)))
                  liste-nom))
        (apply 'direct-multiple
              'est
              nom
              (progn
               (setq liste-nom '(partie-droite))
               (for (h 0 1 (- n 2))
                    (setq liste-nom
                          (cons
                           (if (oddp h)
                             'wdx
                              (list 'wdx '((sym y))))
                           liste-nom)))
               liste-nom)))))
(fin-acces)))
```

## ANNEXE 6: Description NIL d'un décodeur à 3 entrées

(defcell 'decodeur3)
(acces 'decodeur3)
(defmotif 'wdl)
(acces 'wdl)
(addpoly 'M1 19 294 84 294 84 447 -81 447 -81 -5 19 -5 19 294)
(fin-acces)
(defmotif 'wdx)
(acces 'wdx)
(addrect 'BC 44 198 44 30)
(addrect 'RX 47 201 38 24)
(addpoly 'M1 114 116 137 116 137 183 -5 183 -5 169 114 169 114 116)
(addpoly 'M1 -5 -5 -5 148 83 148 83 95 137 95 137 -5 -5 -5)
(addrect 'M1 16 203 100 20)
(addrect 'CA 51 206 30 14)
(addpoly 'P1 40 187 92 187 92 95 105 95 105 200 92 200 92 231 40 231 40 187)
(addrect 'CA 117 119 19 36)
(addrect 'CA 52 115 28 30)
(addrect 'RX 32 100 105 70)
(addrect 'M2 101 -5 18 452)
(addrect 'M2 13 -5 18 452)
(fin-acces)
(defmotif 'wdpd)
(acces 'wdpd)
(addpoly 'M1 -5 294 -5 447 274 447 274 169 214 169 214 294 -5 294)
(addrect 'M1 -5 -5 313 53)
(addpoly 'M1 22 182 4 182 4 90 -5 90 -5 76 22 76 22 182)
(addpoly 'M1 308 147 308 131 192 131 192 126 42 126 42 140 178 140 178 236 32 236 32 250 192 250 192 147 308 147)
(addpoly 'P1 30 66 50 66 50 115 93 115 93 66 108 66 108 115 151 115 151 66 166 66 166 96 209 96 209 82 180 82 180 52 137 52 137 101 122 101 122 52 79 52 79 101 64 101 64 52 -5 52 -5 274 225 274 225 261 20 261 20 222 30 222 30 66)
(addrect 'CA 35 297 170 20)
(addrect 'NW -5 -5 277 185)
(addrect 'TN -5 -5 277 185)
(addrect 'BN -5 16 292 148)
(addrect 'CA 7 79 12 100)
(addrect 'CA 35 237 156 12)

(addrect 'RX 20 222 200 110)
(addrect 'CA 45 127 144 12)
(addrect 'CA 46 -5 143 41)
(addrect 'RX 30 -5 174 160)
(addrect 'BP -5 16 292 148)
(addrect 'BF -5 -5 277 185)
(fin-acces)
(defmotif 'wda)
(acces 'wda)
(addrect 'M2 464 -5 18 231)
(addrect 'M2 376 -5 18 231)
(addrect 'M1 31 74 294 16)
(addrect 'CA 201 76 121 12)
(addrect 'CA 34 76 70 12)
(addrect 'P1 14 99 115 14)
(addrect 'BF -6 -5 150 231)
(addrect 'TN -6 -5 150 231)
(addrect 'CA 34 124 70 12)
(addrect 'M1 -5 -5 485 57)
(addrect 'M1 31 173 449 53)
(addpoly 'P1 186 113 186 162 14 162 14 148 173 148 173 100 455 100 455 157 403 157 403 113 186 113)
(addpoly 'M1 31 137 31 123 345 123 345 74 480 74 480 88 359 88 359 137 31 137)
(addrect 'NW -6 -5 150 231)
(addrect 'BN 10 -5 118 210)
(addrect 'BP 10 -5 118 210)
(addrect 'CA 34 174 70 52)
(addrect 'RX 19 61 100 165)
(addrect 'BC 411 123 36 32)
(addrect 'RX 414 126 30 26)
(addrect 'M1 379 129 100 20)
(addrect 'CA 414 134 30 12)
(addrect 'CA 201 124 155 12)
(addrect 'RX 186 -5 200 156)
(fin-acces)
(defmotif 'wddv)
(acces 'wddv)
(addpoly 'M1 -5 173 -5 226 1355 226 1355 145 1169 145 1169 173 -5 173)
(addrect 'M1 -5 -5 397 57)

(addrect 'BN 30 -5 383 25)
(addrect 'BP 30 -5 383 25)
(addpoly 'M1 493 61 493 149 53 149 53 133 477 133 477 45 1105 45 1105 38 1455 38 1455 50 1119 50 1119 61 493 61)
(addpoly 'M1 455 109 455 95 30 95 30 74 -5 74 -5 88 12 88 12 109 455 109)
(addrect 'CA 415 96 37 12)
(addpoly 'P1 342 122 342 67 327 67 327 122 286 122 286 67 271 67 271 122 228 122 228 67 213 67 213 122 172 122 172 67 157 67 157 122 116 122 116 67 101 67 101 122 60 122 60 71 40 71 40 120 3 120 3 58 73 58 73 109 88 109 88 54 129 54 129 109 144 109 144 54 185 54 185 109 200 109 200 54 243 54 243 109 258 109 258 54 299 54 299 109 314 109 314 54 355 54 355 109 370 109 370 72 413 72 413 85 383 85 383 122 342 122)
(addrect 'CA 1144 97 20 12)
(addrect 'CA 518 96 37 14)
(addrect 'CA 588 46 528 14)
(addrect 'M1 515 95 652 16)
(addpoly 'P1 602 72 602 152 618 152 618 72 662 72 662 152 678 152 678 72 722 72 722 152 738 152 738 72 782 72 782 152 798 152 798 72 842 72 842 152 858 152 858 72 902 72 902 152 918 152 918 72 962 72 962 152 978 152 978 72 1022 72 1022 152 1038 152 1038 72 1082 72 1082 152 1098 152 1098 76 1176 76 1176 122 1132 122 1132 90 1112 90 1112 166 1068 166 1068 86 1052 86 1052 166 1008 166 1008 86 992 86 992 166 948 166 948 86 932 86 932 166 888 166 888 86 872 86 872 166 828 166 828 86 812 86 812 166 768 166 768 86 752 86 752 166 708 166 708 86 692 86 692 166 648 166 648 86 632 86 632 166 588 166 588 86 568 86 568 132 403 132 403 72 602 72)
(addrect 'CA 56 -4 331 46)
(addpoly 'NW 446 -5 446 226 1318 226 1318 137 1283 137 1283 -5 446 -5)
(addpoly 'BF 446 -5 446 226 1318 226 1318 137 1283 137 1283 -5 446 -5)
(addpoly 'TN 446 -5 446 226 1318 226 1318 137 1283 137 1283 -5 446 -5)
(addrect 'BP 558 -5 584 206)
(addrect 'BN 558 -5 584 206)
(addrect 'CA 15 77 12 31)
(addrect 'CA 488 178 629 47)
(addpoly 'RX 1132 226 1132 30 568 30 568 162 473 162 473 226 1132 226)
(addrect 'RX 40 -5 363 169)
(addrect 'CA 56 134 331 14)
(fin-acces)
(defmotif 'wds)
(acces 'wds)
(addrect 'M2 57 -5 18 452)
(addrect 'M1 -130 -5 297 53)

```
(addrect 'M1 -5 294 172 153)
(addrect 'CA -127 -5 99 52)
(addrect 'RX -143 -5 130 68)
(addrect 'M1 60 183 64 20)
(addpoly 'M1 19 90 167 90 167 76 5 76 5 259 -5 259 -5 273 19 273 19 250 74 250 74 236 19 236
19 90)
(addrect 'BN -3 16 170 97)
(addrect 'BF -168 -5 335 134)
(addrect 'TN -168 -5 335 134)
(addrect 'CA 21 -5 50 41)
(addrect 'CA 21 77 50 12)
(addrect 'RX 6 -5 80 109)
(addrect 'BC 97 218 36 31)
(addrect 'RX 100 221 30 25)
(addrect 'NW -168 -5 335 134)
(addrect 'BP -3 16 170 97)
(addrect 'RX 26 221 60 111)
(addrect 'CA 41 237 30 12)
(addrect 'CA 41 297 30 20)
(addpoly 'P1 135 251 101 251 101 274 21 274 21 261 86 261 86 66 1 66 1 52 101 52 101 175 135
175 135 251)
(addrect 'CA 98 187 25 12)
(fin-acces)
(defmotif 'contact)
(acces 'contact)
(addrect 'PV 0 0 8 14)
(fin-acces)
(defcell 'contact-wd)
(acces 'contact-wd)
(boite 'contact-wd 0 0 132 442)
(instance 'contact 'contact102 19 222)
(fin-acces)
(defcell 'double-contact)
(acces 'double-contact)
(boite 'double-contact 0 0 905 442)
(instance 'contact 'contact103 897 353)
(instance 'contact 'contact104 811 75)
(fin-acces)
(defcell 'partie-droite)
```

```
(acces 'partie-droite)
(boite 'partie-droite 0 0 2217 442)
(instance 'wds 'wds105 0 442 '((sym x)))
(direct 'est 'wds105 'wdpd 'wdpd106 '((sym x)))
(direct 'est 'wdpd106 'wda 'wda107 '((sym x)))
(direct 'nord 'wda107 'wda 'wda108)
(direct 'est 'wda108 'wddv 'wddv109)
(direct 'sud 'wddv109 'wddv 'wddv110 '((sym x)))
(fin-acces)
(defcell 'temp)
(acces 'temp)
(boite 'temp 0 0 0 0)
(fin-acces)
(instance 'temp 'g111 0 0)
(direct 'nord 'g111 'wdl 'wdl112)
(direct 'est 'wdl112 'contact-wd 'contact-wd113)
(direct 'est 'contact-wd113 'contact-wd 'contact-wd114)
(direct 'est 'contact-wd114 'double-contact 'double-contact115)
(direct 'est 'wdl112 'wdx 'wdx116 '((sym x)))
(direct 'est 'wdx116 'wdx 'wdx117 '((rot 180)))
(direct 'est 'wdx117 'partie-droite 'partie-droite118 '((sym x)))
(direct 'nord 'wdl112 'wdl 'wdl119 '((sym x)))
(direct 'est 'wdl119 'contact-wd 'contact-wd120 '((sym x)))
(direct 'est 'contact-wd120 'contact-wd 'contact-wd121 '((rot 180)))
(direct 'est 'contact-wd121 'double-contact 'double-contact122)
(direct 'est 'wdl119 'wdx 'wdx123)
(direct 'est 'wdx123 'wdx 'wdx124 '((sym y)))
(direct 'est 'wdx124 'partie-droite 'partie-droite125)
(direct 'nord 'wdl119 'wdl 'wdl126)
(direct 'est 'wdl126 'contact-wd 'contact-wd127 '((sym y)))
(direct 'est 'contact-wd127 'contact-wd 'contact-wd128)
(direct 'est 'contact-wd128 'double-contact 'double-contact129)
(direct 'est 'wdl126 'wdx 'wdx130 '((sym x)))
(direct 'est 'wdx130 'wdx 'wdx131 '((rot 180)))
(direct 'est 'wdx131 'partie-droite 'partie-droite132 '((sym x)))
(direct 'nord 'wdl126 'wdl 'wdl133 '((sym x)))
(direct 'est 'wdl133 'contact-wd 'contact-wd134 '((rot 180)))
(direct 'est 'contact-wd134 'contact-wd 'contact-wd135 '((rot 180)))
(direct 'est 'contact-wd135 'double-contact 'double-contact136)
```

```
(direct 'est 'wdl133 'wdx 'wdx137)
(direct 'est 'wdx137 'wdx 'wdx138 '((sym y)))
(direct 'est 'wdx138 'partie-droite 'partie-droite139)
(fin-acces)
```

## ANNEXE 7: Structure brute de la cellule "decodeur3"

#:tclass:Nautile:CELL:CELLULE:#[#:tclass:Nautile:environnement:cell:decodeur3 #:tclass:Nautile:ETAT:#[#:date:#[1989 10 3 16 43 40 () 2] #:date:#[1989 10 3 16 43 50 () 2] NAUTILE t] #:tclass:Nautile:VUE:ENVIRONNEMENT:#[#:tclass: Nautile:ETAT:#[#:date:#[1989 10 3 16 43 40 () 2] #:date:#[1989 10 3 16 43 40 ( ) 2] NAUTILE t] () () #:tclass:Rect:#[#:tclass:Coord:#[0 0] 0 0 #:tclass: Coord:#[2560 1768] 2560 1768] ()] () (#:tclass:Nautile:environnement:cell: decodeur3:wdl #:tclass:Nautile:environnement:cell:decodeur3:wdx #:tclass: Nautile:environnement:cell:decodeur3:wdpd #:tclass:Nautile:environnement:cell: decodeur3:wda #:tclass:Nautile:environnement:cell:decodeur3:wddv #:tclass: Nautile:environnement:cell:decodeur3:wds #:tclass:Nautile:environnement:cell: decodeur3:contact #:tclass:Nautile:environnement:cell:decodeur3:contact-wd #: tclass:Nautile:environnement:cell:decodeur3:double-contact #:tclass:Nautile: environnement:cell:decodeur3:partie-droite #:tclass:Nautile:environnement: cell:decodeur3:temp) #:tclass:Nautile:VUE:CONSTRUCTION:#[#:tclass:Nautile: ETAT:#[#:date:#[1989 10 3 16 43 43 () 2] #:date:#[1989 10 3 16 43 51 () 2] NAUTILE t] (#:tclass:Nautile:APPEL:AP_CELL:INSTANCE:#[#:tclass:Coord:#[0 0] temp g111 0 #:tclass:Nautile:REPETITION:#[() () () ()]] #:tclass:Nautile: APPEL:AP_CELL:DIRECT:#[() wdl wdl112 0 #:tclass:Nautile:REPETITION:#[() () () ()] nord g111 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd113 0 #:tclass:Nautile:REPETITION:#[() () () ()] est wdl112 ()] #: tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd114 0 #:tclass: Nautile:REPETITION:#[() () () ()] est contact-wd113 ()] #:tclass:Nautile: APPEL:AP_CELL:DIRECT:#[() double-contact double-contact115 0 #:tclass:Nautile: REPETITION:#[() () () ()] est contact-wd114 ()] #:tclass:Nautile:APPEL: AP_CELL:DIRECT:#[() wdx wdx116 6 #:tclass:Nautile:REPETITION:#[() () () ()] est wdl112 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() wdx wdx117 2 #: tclass:Nautile:REPETITION:#[() () () ()] est wdx116 ()] #:tclass:Nautile: APPEL:AP_CELL:DIRECT:#[() partie-droite partie-droite118 6 #:tclass:Nautile: REPETITION:#[() () () ()] est wdx117 ()] #:tclass:Nautile:APPEL:AP_CELL: DIRECT:#[() wdl wdl119 6 #:tclass:Nautile:REPETITION:#[() () () ()] nord wdl112 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd120 6 #:tclass:Nautile:REPETITION:#[() () () ()] est wdl119 ()] #:tclass:Nautile: APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd121 2 #:tclass:Nautile: REPETITION:#[() () () ()] est contact-wd120 ()] #:tclass:Nautile:APPEL: AP_CELL:DIRECT:#[() double-contact double-contact122 0 #:tclass:Nautile: REPETITION:#[() () () ()] est contact-wd121 ()] #:tclass:Nautile:APPEL: AP_CELL:DIRECT:#[() wdx wdx123 0 #:tclass:Nautile:REPETITION:#[() () () ()]

est wdl119 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() wdx wdx124 4 #:
tclass:Nautile:REPETITION:#[() () () ()] est wdx123 ()] #:tclass:Nautile:
APPEL:AP_CELL:DIRECT:#[() partie-droite partie-droite125 0 #:tclass:Nautile:
REPETITION:#[() () () ()] est wdx124 ()] #:tclass:Nautile:APPEL:AP_CELL:
DIRECT:#[() wdl wdl126 0 #:tclass:Nautile:REPETITION:#[() () () ()] nord
wdl119 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd127
4 #:tclass:Nautile:REPETITION:#[() () () ()] est wdl126 ()] #:tclass:Nautile:
APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd128 0 #:tclass:Nautile:
REPETITION:#[() () () ()] est contact-wd127 ()] #:tclass:Nautile:APPEL:
AP_CELL:DIRECT:#[() double-contact double-contact129 0 #:tclass:Nautile:
REPETITION:#[() () () ()] est contact-wd128 ()] #:tclass:Nautile:APPEL:
AP_CELL:DIRECT:#[() wdx wdx130 6 #:tclass:Nautile:REPETITION:#[() () () ()]
est wdl126 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() wdx wdx131 2 #:
tclass:Nautile:REPETITION:#[() () () ()] est wdx130 ()] #:tclass:Nautile:
APPEL:AP_CELL:DIRECT:#[() partie-droite partie-droite132 6 #:tclass:Nautile:
REPETITION:#[() () () ()] est wdx131 ()] #:tclass:Nautile:APPEL:AP_CELL:
DIRECT:#[() wdl wdl133 6 #:tclass:Nautile:REPETITION:#[() () () ()] nord
wdl126 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd134
2 #:tclass:Nautile:REPETITION:#[() () () ()] est wdl133 ()] #:tclass:Nautile:
APPEL:AP_CELL:DIRECT:#[() contact-wd contact-wd135 2 #:tclass:Nautile:
REPETITION:#[() () () ()] est contact-wd134 ()] #:tclass:Nautile:APPEL:
AP_CELL:DIRECT:#[() double-contact double-contact136 0 #:tclass:Nautile:
REPETITION:#[() () () ()] est contact-wd135 ()] #:tclass:Nautile:APPEL:
AP_CELL:DIRECT:#[() wdx wdx137 0 #:tclass:Nautile:REPETITION:#[() () () ()]
est wdl133 ()] #:tclass:Nautile:APPEL:AP_CELL:DIRECT:#[() wdx wdx138 4 #:
tclass:Nautile:REPETITION:#[() () () ()] est wdx137 ()] #:tclass:Nautile:
APPEL:AP_CELL:DIRECT:#[() partie-droite partie-droite139 0 #:tclass:Nautile:
REPETITION:#[() () () ()] est wdx138 ()])]]

## ANNEXE 8: la cellule telle qu'elle est imprimée

Nom : decodeur3

Filles : wdl - wdx - wdpd - wda - wddv - wds - contact - contact-wd - double-contact - partie-droite - temp -

Etat :

Date de creation : mardi 3 octobre 1989 16h 43mn 40s 000ms

Date de Modific : mardi 3 octobre 1989 16h 43mn 50s 000ms

Origine : NAUTILE

Coherence : t

vue environnement

Etat :

Date de creation : mardi 3 octobre 1989 16h 43mn 40s 000ms

Date de Modific : mardi 3 octobre 1989 16h 43mn 40s 000ms

Origine : NAUTILE

Coherence : t

Connects :

Boite : ((0,0)(2560,1768))

Icone : ()

vue construction

Etat :

Date de creation : mardi 3 octobre 1989 16h 43mn 43s 000ms

Date de Modific : mardi 3 octobre 1989 16h 43mn 51s 000ms

Origine : NAUTILE

Coherence : t

Liste des appels :

Nom : temp - Inst : g111 - Pos : (0,0)

Nom : wdl - Inst : wdl112 - Dir : nord - Ref : g111

Nom : contact-wd - Inst : contact-wd113 - Dir : est - Ref : wdl112

Nom : contact-wd - Inst : contact-wd114 - Dir : est - Ref : contact-wd113

Nom : double-contact - Inst : double-contact115 - Dir : est - Ref : contact-wd114

Nom : wdx - Inst : wdx116 - Dir : est - Ref : wdl112 - Transfo : 6

Nom : wdx - Inst : wdx117 - Dir : est - Ref : wdx116 - Transfo : 2

Nom : partie-droite - Inst : partie-droite118 - Dir : est - Ref : wdx117 - Transfo : 6

Nom : wdl - Inst : wdl119 - Dir : nord - Ref : wdl112 - Transfo : 6

Nom : contact-wd - Inst : contact-wd120 - Dir : est - Ref : wdl119 - Transfo : 6

Nom : contact-wd - Inst : contact-wd121 - Dir : est - Ref : contact-wd120 - Transfo : 2

Nom : double-contact - Inst : double-contact122 - Dir : est - Ref : contact-wd121

Nom : wdx - Inst : wdx123 - Dir : est - Ref : wdl119

Nom : wdx - Inst : wdx124 - Dir : est - Ref : wdx123 - Transfo : 4

Nom : partie-droite - Inst : partie-droite125 - Dir : est - Ref : wdx124

Nom : wdl - Inst : wdl126 - Dir : nord - Ref : wdl119

Nom : contact-wd - Inst : contact-wd127 - Dir : est - Ref : wdl126 - Transfo : 4

Nom : contact-wd - Inst : contact-wd128 - Dir : est - Ref : contact-wd127

Nom : double-contact - Inst : double-contact129 - Dir : est - Ref : contact-wd128

Nom : wdx - Inst : wdx130 - Dir : est - Ref : wdl126 - Transfo : 6

Nom : wdx - Inst : wdx131 - Dir : est - Ref : wdx130 - Transfo : 2

Nom : partie-droite - Inst : partie-droite132 - Dir : est - Ref : wdx131 - Transfo : 6

Nom : wdl - Inst : wdl133 - Dir : nord - Ref : wdl126 - Transfo : 6

Nom : contact-wd - Inst : contact-wd134 - Dir : est - Ref : wdl133 - Transfo : 2

Nom : contact-wd - Inst : contact-wd135 - Dir : est - Ref : contact-wd134 - Transfo : 2

Nom : double-contact - Inst : double-contact136 - Dir : est - Ref : contact-wd135

Nom : wdx - Inst : wdx137 - Dir : est - Ref : wdl133

Nom : wdx - Inst : wdx138 - Dir : est - Ref : wdx137 - Transfo : 4

Nom : partie-droite - Inst : partie-droite139 - Dir : est - Ref : wdx138

## ANNEXE 9: Le fichier LUCIE généré

```
NIV MD,MB,MC,MP,MM,MS,ME,MV,MH,MG
FIG decodeur3
FIG contact-wd
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lcontact.luc (19,222)
FFIG
FIG double-contact
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lcontact.luc (897,353)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lcontact.luc (811,75)
FFIG
FIG partie-droite
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwds.luc (0,442)
FSYM
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdpd.luc (162,442)
FSYM
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwda.luc (429,221)
FSYM
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwda.luc (429,221)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwddv.luc (904,221)
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwddv.luc (904,221)
FSYM
FFIG
FIG temp
FFIG
FIG temp (0,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdl.luc (0,0)
FIG contact-wd (79,0)
FIG contact-wd (211,0)
FIG double-contact (343,0)
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (79,442)
FSYM
ROT(+,0)
ROT(+,0)
```

FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (343,442)
FROT
FROT
SYM(Y,0)
FIG partie-droite (343,442)
FSYM
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdl.luc (0,884)
FSYM
SYM(Y,0)
FIG contact-wd (79,884)
FSYM
ROT(+,0)
ROT(+,0)
FIG contact-wd (343,884)
FROT
FROT
FIG double-contact (343,442)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (79,442)
SYM(X,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (343,442)
FSYM
FIG partie-droite (343,442)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdl.luc (0,884)
SYM(X,0)
FIG contact-wd (211,884)
FSYM
FIG contact-wd (211,884)
FIG double-contact (343,884)
SYM(Y,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (79,1326)
FSYM
ROT(+,0)
ROT(+,0)
FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (343,1326)
FROT
FROT
SYM(Y,0)
FIG partie-droite (343,1326)

FSYM

SYM(Y,0)

FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdl.luc (0,1768)

FSYM

ROT(+,0)

ROT(+,0)

FIG contact-wd (211,1768)

FROT

FROT

ROT(+,0)

ROT(+,0)

FIG contact-wd (343,1768)

FROT

FROT

FIG double-contact (343,1326)

FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (79,1326)

SYM(X,0)

FIGEXT /u/nautile/bondono/nautile/BIBLI/LUCIE/lwdx.luc (343,1326)

FSYM

FIG partie-droite (343,1326)

FFIG