Contribution à la définition et à la mise en œuvre de NAUTILE

Armand Hornik

T H E S E

présentée par

Armand HORNIK

pour obtenir le titre de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(arrêté ministériel du 5 juillet 1984)

(**Spécialité** : *Informatique*)

==========

Contribution à la définition et à la mise en oeuvre de **NAUTILE**
(Un nouvel environnement pour la conception et l'assemblage des circuits intégrés).

==========

Date de soutenance : 6 Juin 1989

Composition du jury :

J.	*Mossière*	*Président*
B.	*Courtois*	
A.A.	*Jerraya*	
C.	*Landrault*	*Rapporteur*
C.	*Masson*	*Rapporteur*
J.P.	*Moreau*	

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 Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	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
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
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 GORREC 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
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

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

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

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

E.F.P.
CHARUEL Robert

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
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 **C. Masson**, Responsable du service de C.A.O. de V.L.S.I. de la BULL, qui m'a accueilli il y a quelques années dans son service et m'a fait découvrir le "monde" du routage.

Mr **C. Landrault**, Directeur de recherche au C.N.R.S. pour avoir accepté d'être rapporteur de ce travail.

Mr **J.P. Moreau**, Chef du département aide à la conception de circuits intégrés de SGS-Thomson pour avoir accepté d'être membre de ce jury.

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 les nombreuses discussions et les encouragements qu'il a su me prodiguer.

Le Laboratoire TIM3 dans son ensemble pour son accueil et l'ambiance amicale qui y règne.

Et tous les amis, aussi bien les anciens que les nouveaux, qui ont su montrer qu'en amitié les frontières n'existent pas : **Antonio, Rosana, Bassam, Habiba, Dominique, Isabelle, Philippe, Meryem, Kaïs, François, Didier, Vladimir, Mohammad, Serge, Kholdoun, Jean-Pierre, Gilles, Thien, Bertrand, Eytan, Lydie, Abderrazak, Seif, Yustina,...** et j'en oublie sûrement.

A mes parents,

I. Introduction

Depuis presque 30 ans, la technologie des circuits intégrés a progressé continuellement, ce qui a eu pour principales conséquences [Anc86]:

- une augmentation constante de la complexité des circuits,
- une constante amélioration de leur vitesse opérationnelle,
- une réduction du coût moyen par fonction.

On peut attribuer l'augmentation de la complexité des circuits intégrés à trois facteurs principaux :

☞ à un affinement de la technologie qui permet de placer de plus en plus de transistors sur une même surface :

- en effet des développements dans la technologie permettent une meilleure optimisation des circuits (par exemple l'ajout d'un nouveau niveau d'interconnexion),

☞ à une augmentation de la surface des circuits fabriqués,

☞ à une diminution du nombre de transistors nécessaires pour réaliser une fonction donnée, ceci étant dû à :

- des améliorations dans les techniques de dessin de circuit qui permettent une meilleure utilisation de la technologie,
- des évolutions dans les méthodes de conception.

Le but de la conception des circuits intégrés est de spécifier un ensemble de masques qui seront utilisés dans le processus de fabrication. La conception de ces masques constitue en fait la partie la plus coûteuse de la conception. Il faut concevoir les circuits de la façon la plus intégrée, et la plus fiable possible. Pour cela il est courant d'adopter une approche structurée de la conception, en subdivisant chaque tâche en sous-tâches, jusqu'à obtenir des unités topologiques de plus en plus faciles à réaliser. Ce type de méthodologie permet une bonne organisation du travail et permet une optimisation globale, plus efficace qu'une optimisation locale.

Pour ces raisons il est essentiel de disposer d'outils de conception évolués, permettant de structurer convenablement la conception d'un circuit.

I.1 Présentation générale des systèmes de conception de circuits VLSI.

Alors que la conception de circuits VLSI de la complexité d'un microprocesseur est relativement récente, on conçoit des logiciels d'une complexité égale ou supérieure depuis de nombreuses années. Il serait donc naturel de penser que l'expérience et les outils utilisés pour la conception de logiciel pourraient être utilisés pour la conception de circuits intégrés [Ull84]. On peut recenser un certain nombre de facteurs communs à ces deux disciplines :

- dans les deux cas, pour concevoir un circuit ou un logiciel important, il est nécessaire que plusieurs ingénieurs travaillent en équipe, et la partie la plus ardue consiste à coordonner et assembler les différents travaux,
- dans les deux domaines les concepteurs travailleront "mieux" en utilisant des langages dits de haut niveau.

Cependant on peut citer plusieurs raisons pour lesquelles ces similarités ne peuvent être exploitées complètement :

Exécuter un programme, vérifier qu'il fonctionne, trouver une erreur, la corriger, recompiler le programme, ... tout cela ne prendra en général que quelques minutes, en revanche envoyer un circuit à la fabrication et attendre son retour peut prendre de quelques semaines à quelques mois. De même le prix d'un circuit est proportionnel à sa taille; pour un programme en revanche, il sera parfois plus économique d'utiliser un langage évolué qui générera plus de code mais qui permettra de programmer en moins de temps. Enfin la vérification de la syntaxe pour le "code machine" est plus facile dans le cadre de l'écriture d'un programme que dans la conception d'un circuit. Pour le programme une simple vérification de la syntaxe de chacune des lignes et des adresses données suffira (ce qui correspondra à un temps dépendant linéairement de la taille du programme), alors que pour un circuit il faudra vérifier toutes les possibilités de collision, ceci ne se concevant pas localement mais plutôt globalement.

Pour ces différentes raisons il faut automatiser au maximum les étapes de la conception d'un circuit intégré et détecter au cours de la conception même du circuit les erreurs possibles. Les solutions les plus couramment utilisées pour résoudre ces problèmes sont :

- un compilateur de silicium de haut niveau (voir II.2.2),
- un environnement de modules générateurs.

C'est cette dernière solution qui a été choisie ici.

I.2 Histoire et histoires de NAUTILE

L'équipe d'architecture des ordinateurs a commencé à s'intéresser au problème des systèmes de conception de VLSI à la fin des années 70. A cette époque un outil de dessin de bas niveau a été développé, le système LUCIE (Langage Universitaire de Conception de Circuits Intégrés pour l'Enseignement) [Pai85]. Cet outil offrait essentiellement un éditeur graphique.

Par la suite il a été nécessaire d'envisager un outil plus puissant de conception de circuit V.L.S.I., mais restant simple, et répondant aux besoins des concepteurs de circuits intégrés. L'idée de ce système partait du principe de la hiérarchisation des circuits intégrés sous forme d'arbres, dont les feuilles étaient constituées par des petites figures (des cellules), pouvant être dessinées à la main et prévues pour s'assembler entre elles. Le système LUBRICK (assembleur de BRIQues LUcie) était né [Sch83][Sch85].

Forte de cette expérience, l'équipe d'architecture des ordinateurs s'est associée à un projet ambitieux : SYCOMORE. Il s'agissait de créer un système complet de conception de circuits intégrés. Ce projet mettait en collaboration BULL S.A., THOMSON, l'INRIA et l'INPG (TIM3-Grenoble). L'INPG devait se charger de réaliser un compilateur de silicium (SYCO) [JVJ86], ainsi que participer à la définition d'un système de conception hiérarchique symbolique de dessin de masques (STYX) [JRR85] [Rou87]. Cet éditeur, STYX, (voir chap II.2.1.3) avait pour éléments de base des articulations et des fils (pour les connexions) ainsi que des transistors. Il offrait un système puissant d'assemblage automatique de cellules, utilisant des fils extensibles. Ce système cependant s'est révélé insuffisant pour le traitement de grands exemples, 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).

L'expérience acquise avec STYX, permit, en septembre 1986, d'envisager un nouveau projet, pouvant résoudre les problèmes posés. Ce projet, appelé NAUTILE (New AUtomatic and Technology Independant Layout Environment), devait permettre d'une part, de résoudre le problème posé par la place mémoire, grâce à des "motifs" externes au système (pouvant éventuellement être décrits à l'aide d'autres systèmes de conception de circuits intégrés), d'autre part être indépendant de la technologie, et enfin permettre de décrire des circuits sous différents niveaux d'abstractions tout en conservant les liens entre ces niveaux. Cette thèse est une contribution à la définition du système NAUTILE.

I.3 Présentation de la thèse

Le premier chapitre est constitué de cette introduction. Le deuxième chapitre décrit les différents outils nécessaires à la conception d'un circuit intégré, ainsi que les systèmes de conception existants (de façon non exhaustive). Le troisième chapitre décrit les grands principes qui ont conduit à la création du système NAUTILE. Le quatrième chapitre, décrit la réalisation pratique d'un premier prototype. Enfin le cinquième chapitre donne un exemple d'utilisation de NAUTILE dans le cadre du projet SYCOMORE.

Dans l'annexe I on trouvera un glossaire expliquant les différents termes techniques couremment utilisé. L'annexe II décrit la structure de données NAUTILE. Enfin l'annexe III présente un manuel d'utilisation du système NAUTILE.

II. Environnement des systèmes de CAO de VLSI

Dans ce chapitre nous allons décrire dans une première partie les principales composantes d'un système de C.A.O. de V.L.S.I. Puis nous décrirons quelques uns des différents systèmes existant actuellement ainsi que leurs principales caractéristiques.

II.1 Description d'un système de conception de VLSI

Nous allons décrire un système conventionnel de conception de VLSI. La figure II.1 donne une idée de l'organisation générale de ce type de système [Ull84]. L'ensemble des outils évolue autour d'une description topologique. La représentation topologique d'un circuit est la plus proche de la réalisation physique. Elle est l'un des objectifs finaux de la conception d'un circuit et représente en fait le dessin des masques.

II.1.1 Les outils disponibles

Un système de conception de VLSI doit offrir un certain nombre d'outils qui facilitent le travail de l'utilisateur. Ces outils doivent interagir pour offrir à l'utilisateur une véritable "boîte à outils" lui permettant de choisir l'outil le plus approprié au meilleur moment.

Nous allons décrire ici la fonction de ces outils.

II.1.1.1 Les outils de génération

A. Les générateurs de structures régulières

La conception de circuits intégrés utilise fréquemment des cellules à structures régulières correspondant à des fonctions de logique combinatoires ou séquentielles [Seg85] ou à des mémoires.

Elles possèdent les caractéristiques suivantes :

— une structure logique simple,

— une structure électrique simple,

— une structure topologique simple.

Exemple

Les PLA's sont un exemple typique de ce type de structure.

Un PLA transposera en multi-fonction soit des équations booléennes, soit des fonctions séquentielles. Ces PLA's peuvent être générés automatiquement par des programmes spécialisés permettant de passer directement d'une description logique au dessin des masques.

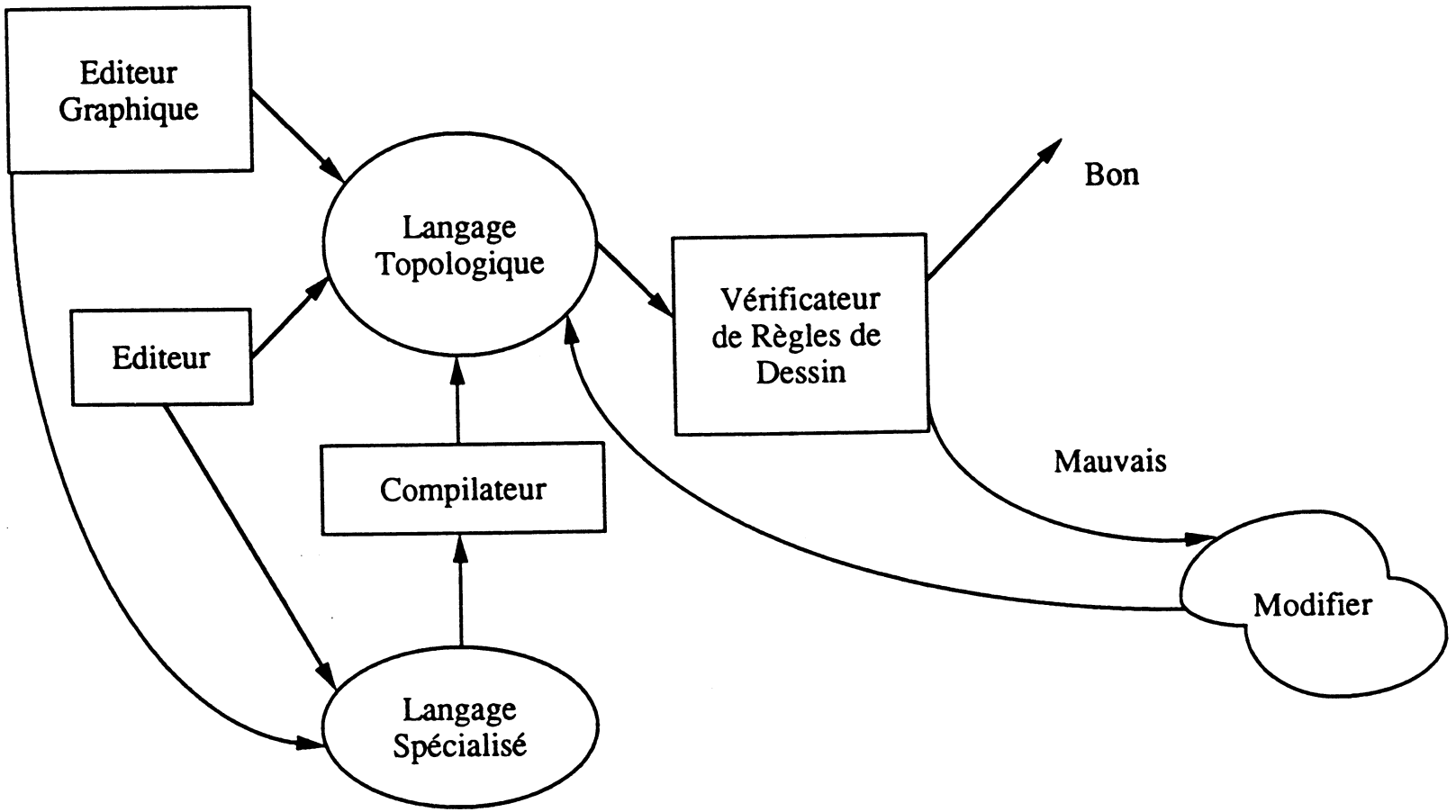


Figure II.1 : Représentation d'un Système Conventionnel

Les générateurs de PLA's peuvent être associés à des optimiseurs de PLA's permettant de les optimiser logiquement et topologiquement.

Les générateurs de PLA's sont simples à réaliser et sont abondamment utilisés dans la conception de circuit intégrés.

De plus, de par la régularité des PLA's, il est facile d'adopter des techniques générales pour générer des structures auto-testables (voir II.1.3).

B. Les outils de routage

Il existe différents types de problèmes de routage [Hor86]. Ces problèmes proviennent de l'assemblage de cellules ou de blocs. En effet pour assembler deux cellules il est nécessaire de relier par des "fils" les connecteurs de ces cellules. Un ensemble de connecteurs de cellules à relier par une même équipotentielle sera appelé réseau à connecter. Pour connecter ces réseaux, on dispose en général d'une surface rectangulaire, appelée canal, comportant un certain nombre de connecteurs sur les bords et un certain nombre de contraintes à respecter. Ces contraintes peuvent être définies par les caractéristiques suivantes :

- nombre de couches d'interconnexion utilisables,
- règles d'interconnexion :
  - — deux réseaux différents ne peuvent se croiser sur la même couche;
  - — un même réseau sur deux couches est connecté à l'aide d'un via;
  - — seuls les déplacements verticaux et horizontaux seront autorisés.

Les conventions définies ici sont générales mais non exhaustives. Cependant la plupart des routeurs classiques s'y soumettent.

Il faut différencier deux grandes familles de routage (Voir Figure II.2) :

i. Le routage Global [Lee61] [Hig83] [Clo84] [Xio86]

Dans l'assemblage des cellules, une phase importante consiste en le placement des cellules, puis l'assignation des canaux par lesquels vont être routées les différentes connexions. Pour cela il existe des routeurs dits "globaux" permettant d'effectuer le partage du plan de masse en divers canaux de routage et de décider du chemin global à adopter pour les diverses connexions.

Le résultat d'un routage global consistera en fait en ordres de routages locaux pour des routeurs bi-couche ou mono-couche.

ii. Le routage Local

Une fois planifiés les différents routages à effectuer, un routeur local prendra le relais du routeur global. Il faut distinguer deux types de routage local qui dépendront des contraintes à observer (nombre de couches disponibles, ordonnancement des connecteurs,...) (voir Figure II.2) :

Routage mono-couche

Il est préférable, dans la mesure du possible, de router sur une seule couche. En effet ceci permet d'utiliser la couche économisée pour d'autres routages ... Ceci n'est possible que lorsque les connexions à effectuer ne nécessitent aucun croisement de fils.

Là encore existent deux types de routages :

— Routage sans obstacle [Hsu83] [Sou83]

Les connecteurs peuvent être placés de chaque côté d'un canal réservé au routage, chacun des connecteurs devant être relié à son correspondant sur la rive opposée, dans ce cas on parle de routage par dévoiement(*River-routing* en anglais) ou bien l'on peut disposer d'une surface fermée, sur les bords de laquelle sont placés les connecteurs, toujours dans un ordre tel qu'on puisse les relier sans croisement de fils.

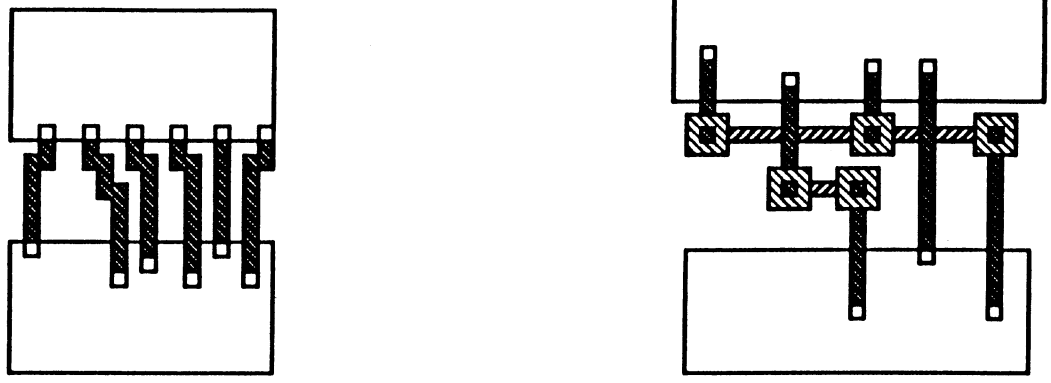
— Routage avec obstacles [Kar83]

On peut introduire au milieu des surfaces de routage des obstacles, qui sont définis comme des zones dans lesquelles il est interdit de faire passer les fils. Ceci peut s'appliquer au dévoiement tout comme à un routage mono-couche plus général.

Le problème généralement rencontré dans ces types de routage est l'ordre dans lequel les fils doivent être routés et de là, la nécessité d'une vision globale.

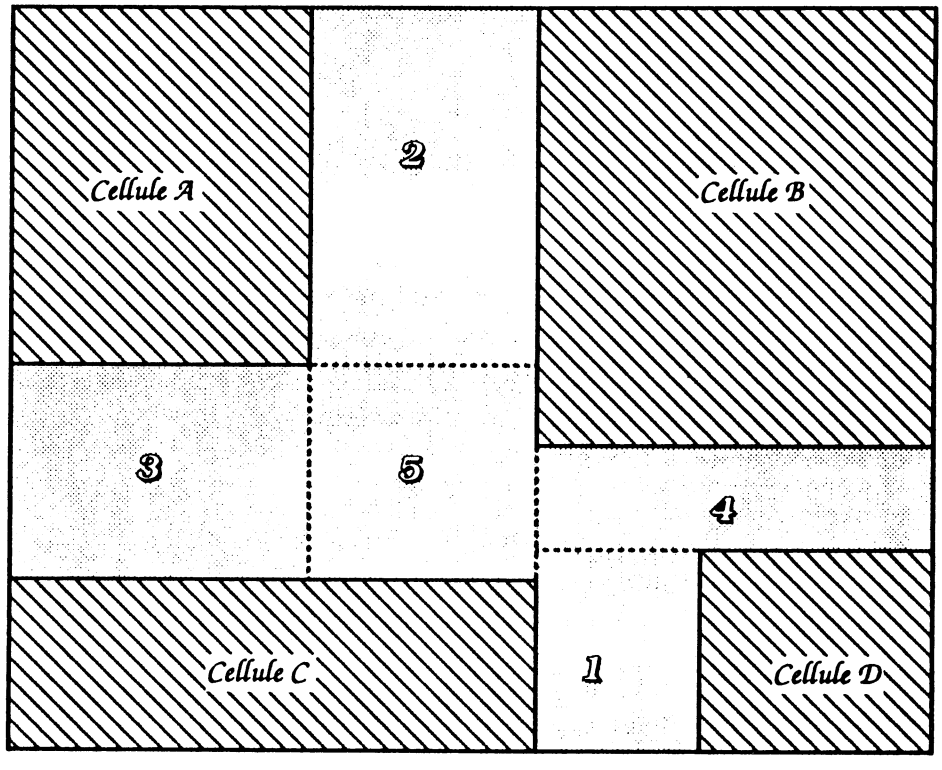
Routage multi-couche

On utilise le routage multi-couche (bi-couche en général) pour effectuer le routage des canaux prédéfinis par le routage global. On a deux types de routage : à deux rives fixes et à quatre rives fixes. En effet lors de la division d'une surface de routage en différents canaux on pourra pour les premiers canaux n'avoir que deux ou trois rives dont les positions des connecteurs sont imposées, mais pour le ou les derniers canaux à router on aura logiquement quatre rives où les



Routage mono-couche **Routage bi-couche**

[a] Routage local



[c] Routage Global

Fig. II.2 : Différents types de routage

positions des connecteurs seront imposées (voir Figure II.2.b).

— Routage dans un canal [RiF82] [Deu76] [Yos84]

Ce type de routage classique est très utilisé dans les systèmes de C.A.O. de V.L.S.I. On retrouve dans ces routeurs des règles communes, comme l'utilisation d'une couche pour les déplacements verticaux et d'une autre couche pour les déplacements horizontaux. Le canal a une longueur L, il y a quatre listes de terminaux (connecteurs) , Nord, Sud, Est et Ouest avec leurs positions ainsi que les équipotentielles aux laquelles ils appartiennent. On a par exemple les positions des terminaux Nord et Sud qui sont imposées alors que les positions Est et Ouest sont libres.

Le résultat obtenu par un routeur sur ces données sera un ensemble de segments verticaux et horizontaux permettant de relier entre eux les terminaux appartenant à la même équipotentielle.

— Routage dans une boîte d'interconnexions [Bur83] [Joo86]

Dans ce routage la seule différence avec le routage précédent réside dans les positions des terminaux Est et Ouest qui sont imposées, ceci augmentant considérablement les contraintes et la complexité du problème posé.

II.1.1.2 Les outils de vérification

D'une façon générale, on distingue différents types de vérification, correspondant à des étapes données de la vie d'un circuit. Avant de réaliser le circuit obtenu, il est nécessaire de vérifier que toutes les règles de dessin ainsi que les règles électriques sont respectées. On a besoin à cet effet de Vérificateur de Règles de Dessin (V.R.D.), de Vérificateur de Règles Electriques (V.R.E.), de simulateurs divers,... . Il existe deux méthodes de vérification de la conception :

— La vérification hiérarchique, qui peut être aussi appliquée en concevant le circuit : si l'on n'assemble que des cellules déjà vérifiées, en utilisant des règles interdisant un assemblage donnant lieu à des erreurs, le circuit final ne pourra être que correct par construction

— La vérification par mise à plat, qui cloture souvent la phase de conception avant de commencer celle de fabrication : Tous les liens hiérarchiques sont supprimés et le circuit est mis *à plat* (les appels de cellules sont remplacés par les corps de ces cellules) puis on vérifie sur l'intégralité du circuit que toutes les règles sont respectées, ceci prenant généralement un temps de calcul considérable.

Après que la conception ait été vérifiée il faut s'assurer que la fabrication du circuit ne présente pas de défaut.

A. Vérification de la conception

1. Les outils de simulation logique et électrique

Dans la conception d'un circuit intégré il est très important de pouvoir vérifier que le résultat obtenu, à savoir le dessin des masques, est bien conforme au circuit souhaité, et que le circuit obtenu est électriquement correct. La seule façon de le vérifier consiste à simuler le fonctionnement logique et électrique du circuit.

Cette simulation consiste à calculer virtuellement les états des noeuds (entrées/sorties, internes, externes, . . .) du circuit afin de vérifier son bon fonctionnement . Il s'agit donc de créer dans le système des objets utilisables par des simulateurs. Il faut citer les quelques simulateurs rencontrés :

— les simulateurs logiques (HILO , EPILOG),

— les simulateurs *switch*, ou interrupteurs (MOSSIM [Bry84]),

— les simulateurs électriques (SPICE).

Ces simulateurs traitent la représentation logique ou électrique d'un circuit et peuvent inclure une simulation temporelle prenant en compte les délais. Ils peuvent également comporter une partie "simulation de pannes" servant de base à la génération de vecteurs de test (voir B).

2. Les différents outils de comparaison et d'extraction

A tous les niveaux de la conception, il est nécessaire de pouvoir vérifier si les différentes descriptions sont cohérentes entre elles. Pour cela on peut utiliser divers outils permettant de comparer des descriptions fonctionnelles et logiques, logiques et électriques, électriques et topologiques. Ceci permet à tout moment de savoir si le circuit en mémoire est identique au circuit désiré. De même il existe des extracteurs, topologique/électrique, électrique/logique, logique/fonctionnel pouvant passer d'une description à une autre. Ces types d'outils sont coûteux en temps de calcul et en place mémoire, aussi le principe des comparateurs utilisés de façon incrémentale est-il préférable (pour chaque cellule créée, on peut vérifier si ses diverses descriptions sont cohérentes en considérant que les sous-cellules appelées sont déjà cohérentes).

B. Vérification de la fabrication - les générateurs de vecteurs de test

Une fois qu'un circuit a été conçu et réalisé, il est probable qu'il y aura des défauts. Ces défaillances sont dues soit à des défauts de fabrication, soit à un vieillissement. Dans ce cas il est nécessaire de prévoir une série de données d'entrée du circuit permettant de mettre en évidence les pannes possibles. On appelle ces séries de données **des vecteurs de test .**

Il faut commencer par recenser les pannes possibles (court-circuit entre deux lignes proches, coupure de lignes, collage d'un transistor,...) puis on cherche les vecteurs de test manifestant ces pannes.

Cette recherche de vecteurs de test peut être automatisée à l'aide de générateurs de vecteurs de test. Les méthodes classiques utilisées, dont le D-algorithme [RBS67], consistent à propager une panne jusqu'aux sorties, puis à remonter jusqu'aux entrées du circuit par une procédure de sensibilisation d'un chemin de données.

II.1.1.3 Les outils de compaction topologique

Un concepteur cherche à réaliser un circuit dans un temps raisonnable, et ne peut pas le dessiner pour qu'il occupe une surface minimale. Il existe en effet un grand nombre de règles liées à la technologie utilisée qui imposent des distances à respecter suivant les différentes couches technologiques utilisées [WoD86]. Le but d'un compacteur sera de réduire la surface utilisée.

C'est pourquoi il est plus facile pour le concepteur de décrire un circuit sans chercher à compacter son dessin des masques, et d'utiliser des outils automatiques à cet usage. Les systèmes classiques de conception étant hautement hiérarchisés, il est normal que l'on retrouve cette hiérarchie dans ces outils. On peut recenser trois types de compaction :

- La compaction de cellules de plus bas niveau hiérarchique.

  Ces cellules sont souvent des cellules extraites de bibliothèques, (ou conçues pour figurer dans une bibliothèque) et sont répétées un nombre considérable de fois. Il est donc essentiel qu'elles soient le plus compactes possible.

- La compaction de cellules que l'on assemble.

  Deux cellules prises séparément peuvent nécessiter une surface supérieure à la surface nécessaire à leur assemblage. En effet il est possible de modifier suivant les cas particuliers, les marges à respecter (superposition de deux fils d'alimentations, etc...)

- La compaction de hiérarchies entières de cellules

De même que précédemment, et pour les mêmes raisons, des compactions différentes peuvent être réalisables.

Pour ces trois niveaux on peut appliquer des compactions suivant différentes directions (suivant l'axe des X, Y, XY, YX, . . .) pour obtenir des résultats différents.

II.1.2 L'environnement des outils

II.1.2.1 Un éditeur graphique évolué

La plupart des concepteurs n'utiliseront un système de C.A.O que si celui-ci offre des possibilités graphiques évoluées. Il doit être possible d'effectuer toutes les opérations courantes graphiquement :

- Création de cellules,
- Modification de cellules,
- Création de macro-fonctions, etc...

II.1.2.2 Langages de descriptions topologique, électrique ou logique.

Un système doit pouvoir traiter les différents aspects de la conception d'un circuit intégré. Pour cela on doit pouvoir décrire à l'aide d'un langage approprié ses aspects topologique, électrique et logique. Dans une description topologique on trouvera en général des rectangles et les différents niveaux où ils se placent. Dans une description électrique on trouvera des transistors, des résistances et des capacités. Enfin dans une description logique, on trouvera des portes *NAND, NOR, INV* (non-et, non-ou, inverseur), ...

On pourra aussi avoir un langage de description fonctionnelle (il se rapproche du langage logique et la description fonctionnelle correspond étroitement à la vision que se fait le concepteur du futur circuit à réaliser), ainsi qu'un langage temporel (introduction des délais), ...

II.1.2.3 Langage de description technologique

Les changements de technologie étant fréquents, un système de conception de VLSI doit pouvoir s'adapter à ces changements. Il ne faut en aucun cas qu'un changement de technologie oblige le concepteur à reprendre son travail à zéro pour un même circuit et qu'il soit obligé de redessiner entièrement son circuit. Pour cela il faut que la technologie soit paramètrable. On doit avoir alors un langage permettant de décrire la technologie, et dans le cas d'un changement de technologie, la seule modification de cette description doit permettre de générer un nouveau circuit.

II.1.2.4 Description procédurale de circuit

On assiste à une évolution du métier de concepteur ; si le concepteur d'*hier* n'avait pas besoin de connaissances en informatique, il n'en est plus tout à fait de même aujourd'hui. Le concepteur dispose d'un langage de description offrant les mêmes possibilités que les langages informatiques de haut niveau. En effet la description d'un circuit nécessite d'exprimer des notions complexes, de façon générale, et il est nécessaire de disposer de facilités algorithmiques (comprenant la boucle simple, l'éxécution conditionnelle, l'appel de procédure, ... voir III.1.2)

II.1.3 La structure de données

II.1.3.1 Conception structurée

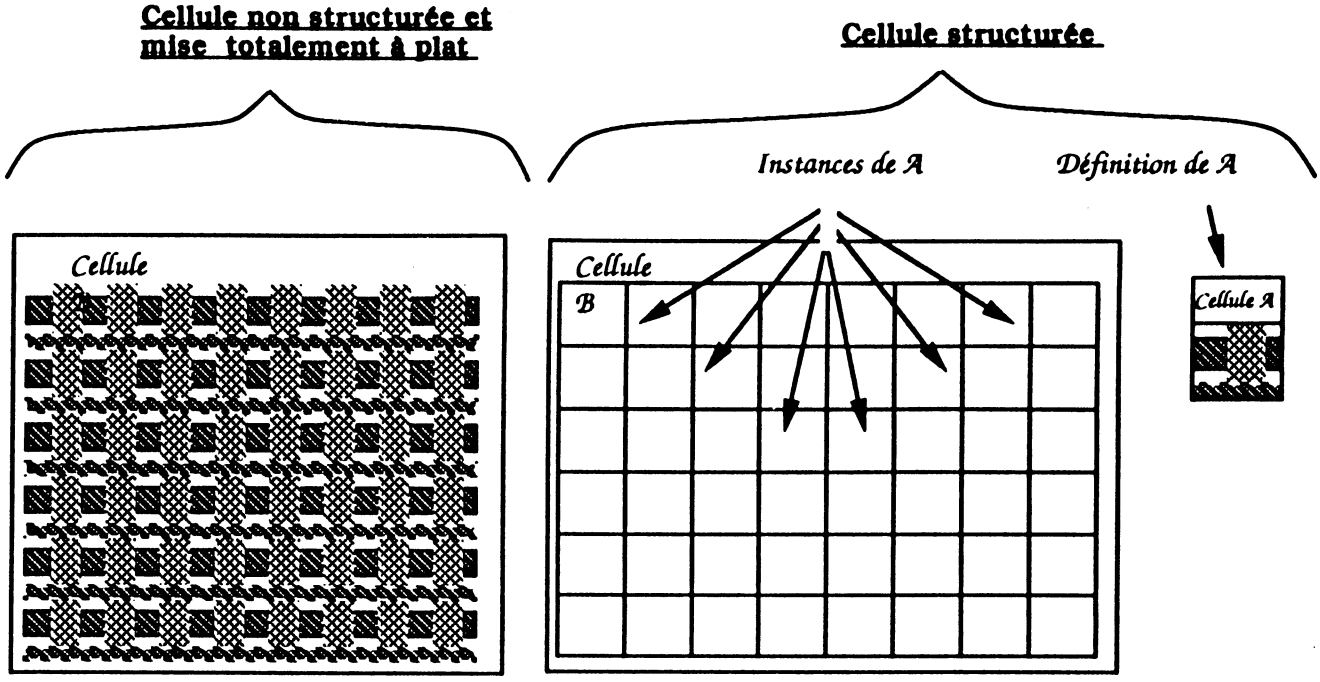
La conception d'un circuit doit être considérée de façon descendante. La première approche consiste à rechercher les fonctionnalités de celui-ci, puis à chercher à le décomposer en sous-blocs en extrayant des sous-fonctionnalités.

Une fois cette approche descendante effectuée, il va falloir réaliser les "micro-fonctionnalités" ou les blocs élémentaires (qui peuvent être de l'ordre du transistor) constituant la base du circuit, puis les assembler pour obtenir un premier étage dans la structure du circuit, etc...

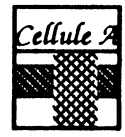
Pour chacune de ces étapes de nombreux outils seront utilisés, soit pour générer les cellules, pour les assembler, pour les vérifier, ou pour vérifier la cohérence du tout au fur et à mesure de la conception.

La nécessité d'une structuration est imposée par le principe même de la conception d'un circuit. Pour concevoir un circuit, un 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 par différentes parties du circuit. C'est-à-dire qu'au lieu de décrire N fois une même cellule, on la décrira une seule fois et on appelera sa description N fois.

De plus la conception d'un circuit implique la manipulation de données en quantités très importantes, aussi il est impossible que celles-ci soient organisées *à plat.* 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, place mémoire). Il faut qu'un maximum de données soit en commun, et que la modification d'une cellule se répercute automatiquement et rapidement dans toute la structure (voir Figure II.3). Pour cela il faut que la structure de données reflète exactement la structure du circuit, et la pensée du concepteur.



Si l'on veut modifier la cellule "à plat" il faudra intervenir dans toute la cellule. En revanche pour la cellule structurée, il suffira de modifier la cellule appelée (A), pour que la modification se répercute sur toute la cellule B.

On modifie A :  devient 

et automatiquement B devient :

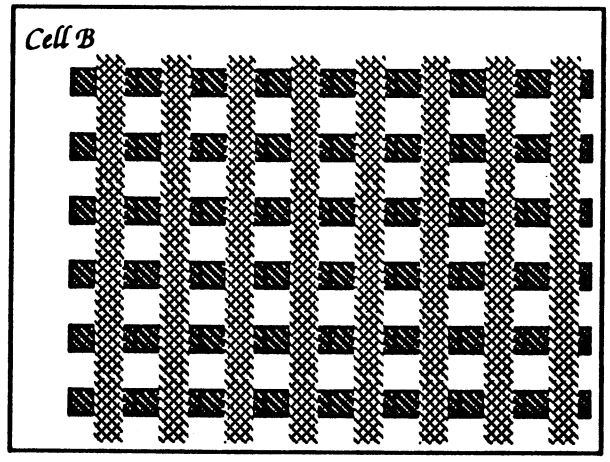


Figure II.3 : Exemple d'application directe de la structuration d'une cellule

II.1.3.2 Liens dynamiques dans une structure de données

La structure de données doit être dynamique. En effet le concepteur pourra à tout moment intervenir dans la structure, modifier celle-ci, rajouter des éléments, avoir deux structures différentes pour deux cellules différentes,... Il effectuera donc toutes sortes d'opérations que n'autoriserait pas une structure figée et ne pouvant pas évoluer.

II.1.4 Gestion de la cohérence

Il existe plusieurs problèmes attachés à la cohérence de la conception d'un circuit :

Un concepteur qui cherche à réaliser un circuit V.L.S.I. est amené à créer des cellules, à les appeler à l'intérieur d'autres cellules, puis à les modifier,... Au bout de quelques opérations, le circuit en mémoire ne correspondra plus au circuit initialement prévu, en effet les appels des cellules et leur placement ne se font pas toujours automatiquement, etc...

De plus, dans le cas d'un système prenant en compte différentes vues (électrique, logique, topologique, ...) d'un circuit, la cohérence doit être assurée entre les différentes vues.

On voit donc bien qu'il est nécessaire que le système prenne en charge la gestion de la cohérence et signale ou interdise les manipulations qui induisent une erreur de conception ou une violation des règles de cohérence.

Pour cette raison il est nécessaire qu'un système de C.A.O. ait des indicateurs de cohérence lui permettant de savoir si une cellule est chronologiquement juste ou pas. Il faut associer à chaque cellule ses dates de création ou de dernière modification, ses dates d'appels, son numéro de version, ...Par exemple lors d'une vérification de cohérence d'une cellule, on vérifiera que les dates d'appel de sous-cellules sont postérieures aux dates de dernière modification des cellules appelées.

II.1.5 Possibilité de paramétrisation

Plus le système permet de tenir compte de différents paramètres, plus il sera possible pour une même conception, de générer plusieurs circuits. Cette paramétrisation peut s'effectuer à différents niveaux:

— **Au niveau technologique :** ceci permet en cas de changement de technologie de n'avoir que quelques paramètres à changer pour regénérer un nouveau circuit.

— **Au niveau de l'assemblage :** il peut ainsi être possible d'avoir une même description pour générer différents types de circuit (Ex : une U.A.L. (Unité Arithmétique Logique) 8 bits et une U.A.L. 16 bits pourront être générées par une même structure, moyennant un changement des paramètres)

Dans les faits cette paramétrisation peut se traduire de plusieurs façons : on peut porter la paramétrisation dans la structure de données et à la demande chercher à évaluer les paramètres manquants ou lorsque le besoin s'en fait sentir, ou bien on peut disposer de programmes admettant des paramètres et générant la structure du circuit désiré. La deuxième méthode semble plus couramment utilisée et plus facile à mettre en oeuvre.

II.1.6 Mécanismes de gestion de la structure de données

Au cours de la conception d'un circuit, un certain nombre d'opérations sur des fichiers est nécessaire. Un circuit peut comporter des centaines de milliers de transistors, des connecteurs encore plus nombreux,... Ceci impose de manipuler des millions d'informations qui ne peuvent résider en totalité en mémoire. Suivant la structure de données, une plus ou moins grande partie de ces informations devra être disponible en mémoire, mais il sera toujours indispensable de manipuler des fichiers.

On peut considérer les fonctionnalités suivantes comme formant une interface minimale entre l'utilisateur et le système de gestion de fichier :

— Sauvegarde de cellules dans un fichier

— Restauration de cellules à partir d'un fichier

— Sauvegarde de hiérarchies

— Restauration de hiérarchies

— Consultation des cellules disponibles

— Accès automatique à des bibliothèques

Pour la gestion de ces fichiers trois choix sont possibles :

a. Refaire totalement un système d'exploitation ou une base de données, avec gestion des autorisations, gestion des versions, création de bibliothèques,...

   Cette solution présente l'avantage d'offrir exactement ce qui est nécessaire à l'utilisateur, d'être relativement indépendante du système hôte, mais elle est relativement complexe à mettre en oeuvre et son coût ne se justifie pas forcément.

b. Se baser totalement sur le système d'exploitation résidant sur la machine hôte.

   Cette solution permet à l'utilisateur de n'être confronté qu'à un système d'exploitation qu'il connait bien et qui est généralement évolué (VMS, UNIX,...). Par contre cette solution pose des problèmes de portabilité du logiciel (fonctionnalité irréalisable d'un système à l'autre,...).

c. Offrir un certain nombre d'utilitaires simples (gestion de bibliothèque : ouverture de fichier, sauvegarde, restauration,...), s'interfaçant avec la plupart des systèmes d'exploitations existant.

Cette solution semble être la meilleure, ne nécessitant pas de connaissances spéciales de l'utilisateur et répondant à la plupart des problèmes posés.

II.1.7 Ouverture du système vers l'extérieur (compatibilité avec les systèmes existants)

Une entreprise achetant un nouveau système de conception de V.L.S.I., aura souvent d'anciens systèmes toujours utilisables, c'est-à-dire un *"passé de conception"*. Ceci se traduira par des bibliothèques de cellules déjà décrites à l'aide d'autres systèmes ainsi que des outils déjà existants. Il est donc indispensable que le concepteur ait toujours à sa disposition ces cellules. Pour cela le nouveau système doit comporter les interfaces nécessaires à :

— La traduction automatique des anciennes cellules dans le nouveau système

— L'utilisation de hiérarchies entières de ces cellules sous forme de squelettes, puis la génération de l'ensemble du circuit (comprenant les nouvelles cellules ainsi que les anciennes) sous forme compréhensible par l'ancien système (pour permettre de conserver opérationnels des utilitaires fonctionnant uniquement dans l'ancien système : simulateurs, extracteurs, générateurs divers, routeurs,...)

II.1.8 Indépendance vis-à-vis de la technologie

Lors d'un changement de technologie il faut minimiser le temps nécessaire à reconstruire un circuit dans la nouvelle technologie. La première opération consistera à décrire la nouvelle technologie, de plus il faudra que le système de conception utilisé permette de concevoir les circuits de façon indépendante de la technologie. Ceci pourra se traduire par plusieurs aspects :

— Un langage de description de la technologie

— Des fichiers technologiques

— Des bibliothèques de cellules décrites dans différentes technologies.

Suivant la ou les solutions choisies, un changement de technologie sera plus ou moins rapide. Dans la solution NAUTILE (voir chapire III) par exemple, à chaque changement de technologie il faut réécrire toutes les cellules de base. Par contre, il n'y a qu'à lancer un programme automatique pour redessiner un ancien circuit dans une nouvelle technologie, une fois la nouvelle technologie décrite.

II.2 Description de quelques systèmes existants

Nous allons ici dresser une liste non exhaustive des différents systèmes de conception existants. Ils sont classés suivant deux types: les systèmes simples et les systèmes dits *intelligents*. On entend par systèmes *intelligents* les compilateurs de silicium. Un compilateur de silicium est un système de conception permettant, à partir d'une description de haut niveau d'un circuit, de générer automatiquement une représentation topologique de ce même circuit (dessin des masques).

II.2.1 Les systèmes simples

II.2.1.1 LUBRICK - LUCIE

LUBRICK (assembleur de BRIQUES bâti au dessus du système LUCIE) [Sch83] [Sch85] [Pai85] est un système d'assemblage écrit et immergé en PASCAL, permettant de compléter LUCIE. En effet le langage LUCIE (Langage Universitaire de Conception de Circuits Intégrés pour l'Enseignement) ne permettait que des applications purement graphiques sans offrir toute la puissance d'un langage de programmation. LUBRICK en lui-même ne permet que de décrire des squelettes de cellules comportant une boîte externe ainsi que des connecteurs, le corps de la cellule, purement topologique, étant décrit en LUCIE. Ce système permet d'assembler des cellules en reliant les connecteurs requis, d'opérer des répétitions, des rotations, des symétries, permettant quelques types de routage ainsi que des frontières variables (les frontières ne sont pas forcément des rectangles mais peuvent avoir des formes polygonales).

La conception d'un circuit à l'aide de LUBRICK, consiste à écrire un programme PASCAL contenant les fonctions LUBRICK prédéfinies, programme qui (après compilation et exécution) va générer des fichiers LUCIE (pour la partie topologique pure : rectangles) et LUBRICK (pour la partie vision externe).

On peut reprocher au système LUBRICK son manque d'interactivité (il faut passer par la phase de compilation, exécution puis appel de l'éditeur LUCIE pour voir un résultat) et son aspect uniquement topologique. On peut cependant noter qu'il existe un mode interprété de LUBRICK, mais n'offrant pas toutes les possibilités de paramétrisation présentes dans le mode compilé, venant de PASCAL.

Cependant il existe des essais d'interface de LUBRICK avec des systèmes autres que topologique (exemple : interface RNL - LUBRICK : elle consiste à traduire sous forme RNL un circuit décrit en LUBRICK-LUCIE [Dri86]). Le système LUCIE est un outil qui a été fortement utilisé dans les universités.

II.2.1.2 VTI

VTI [Tri84] [Lip83] est un système essentiellement graphique permettant de décrire et d'assembler des circuits intégrés. Ce système comprend un ensemble d'outils permettant le placement et la connexion de cellules. Il est axé sur le côté topologique de la conception et travaille de façon symbolique. Il comprend un certain nombre de bibliothèques de cellules permettant à un compilateur de silicium de créer automatiquement de nouveaux circuits fortement paramétrés.

Il y a trois modes généraux de travail sous VTI :

— Editeur de plan de masse

Dans ce cas le système ne gère que des squelettes de cellules comportant leur taille et leurs connecteurs et cherche uniquement à les placer pour le mieux.

— Conception manuelle

Dans ce cas le concepteur utilisera des instances des cellules disponibles en bibliothèque ou de cellules qu'il aura lui-même dessinées, et utilisera VTI pour les placer exactement ou pour les abouter et effectuera les connexions manuellement. Dans ce cas le système est utilisé comme un éditeur de dessin de masques, le concepteur montrant deux points et le système les reliant par un fil, ...

— Conception semi-automatique

Dans ce cas le concepteur prépare ses cellules (en utilisant des cellules de la bibliothèque) sous forme de bâtons (stick) ou sous forme de dessin de masques, puis il réalise un schéma précis de son circuit et utilise les outils de placement routage automatique du système VTI.

La présence d'outils tels que des compacteurs automatiques permet de ne pas se préoccuper de la surface utilisée et donc d'avoir des outils de placement routage beaucoup plus rapides.

On peut reprocher à VTI de ne pas avoir de maintien de la cohérence avec la représentation électrique des cellules et d'être d'une approche complexe si l'on cherche à décrire des cellules dans le langage VIP (langage de base de VTI). En effet VTI est très simple si l'on n'utilise que les éditeurs graphiques mais par contre la syntaxe utilisée pour écrire des générateurs de cellules paramétrées est véritablement complexe.

Ce système est employé dans l'industrie de façon courante.

II.2.1.3 STYX

Le système STYX [JRR85] [Rou87] [FoC88] correspond à une approche générale orientée objet pour décrire des objets graphiques, qui a été utilisée pour la réalisation d'un outil de conception de VLSI. Ce système évolue dans un environnement Lisp. Le système STYX a été réalisé dans le cadre du projet national SYCOMORE de conception de VLSI. Le but de ce projet était de réaliser un système intégré de C.A.O., écrit dans un environnement portable Lisp (Le_Lisp [CDD86]). Ce projet devait contribuer à la réalisation d'une *boite à outils*, comportant :

— un langage de description,

— un simulateur multi-niveau,

— des outils de synthèse,

— un compilateur de silicium (voir SYCO : II.2.2.b),

— un éditeur symbolique appelé STYX.

STYX a été conçu pour s'adapter spécifiquement au dessin des masques. Les concepts graphiques manipulés sont définis à l'aide d'un langage immergé dans Lisp.

Ce système offre 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 leurs comportements, 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 externe 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.

STYX manipule essentiellement des programmes décrivant des cellules et des appels de ces programmes avec les paramètres appropriés.

II.2.1.4 GDT

GDT (GDT : Generator Development Tools) [Bur86] [SDL86] est un système de conception de circuits intégrés, développé par SDL (Silicon Design Lab) basé sur le langage L, lui-même version spéciale du langage C. Ce système comporte toute une série d'outils, d'éditeurs, d'utilitaires qui en font une véritable station de développement pour la conception des VLSI. On y trouve des générateurs de cellules, des routeurs, ... GDT permet de passer par toutes les étapes, de la description fonctionnelle de modèles (dans le langage M, extension de C), jusqu'au dessin des masques, en passant par la génération de vecteurs de test (effectuée grâce au L-simulateur des descriptions de modèles), en offrant un environnement interactif pour l'édition graphique, et permettant de bien structurer les différentes représentations du circuit (voir fig II.4).

On voit déjà apparaître des possibilités de représentations électriques et logiques. Le fait d'avoir C à sa disposition offre toutes les possibilités de paramétrisation pour écrire des programmes générateurs de cellules. Ce langage est particulièrement étudié pour écrire des compilateurs de silicium. La notion de structure est très forte dans GDT. Tout élément de la structure est une cellule, depuis le transistor jusqu'au circuit lui-même. On décrit des cellules que l'on instanciera par la suite pour les assembler, étant entendu que ces instances ne sont pas des copies de la cellule, mais contiennent simplement l'information nécessaire pour l'utiliser. Les notions de connectique sont respectées, c'est-à-dire qu'il est possible de décrire deux connecteurs en contact sans pour autant dessiner les fils correspondants.

De plus GDT offre toutes les possibilités *classiques* des systèmes de conception :

— **Des routeurs:** Les routeurs font intrinsèquement partie du langage L. Cette possibilité offre l'avantage de lier le routage à la structure même des cellules à assembler. Il existe quatre types de routage, deux classiques : un routage mono-couche simple (de type *River-routing*) et un routage bi-couche (de type *Greedy Algorithm* voir II.1.1.1.) avec des entrées sur le côté, plus un routage global décomposant le circuit en différents canaux à router et appelant les deux précédents, et enfin un routage de masse et d'alimentation permettant de connecter les bus de masse et d'alimentation à leurs terminaux avec les bons dimensionnements.

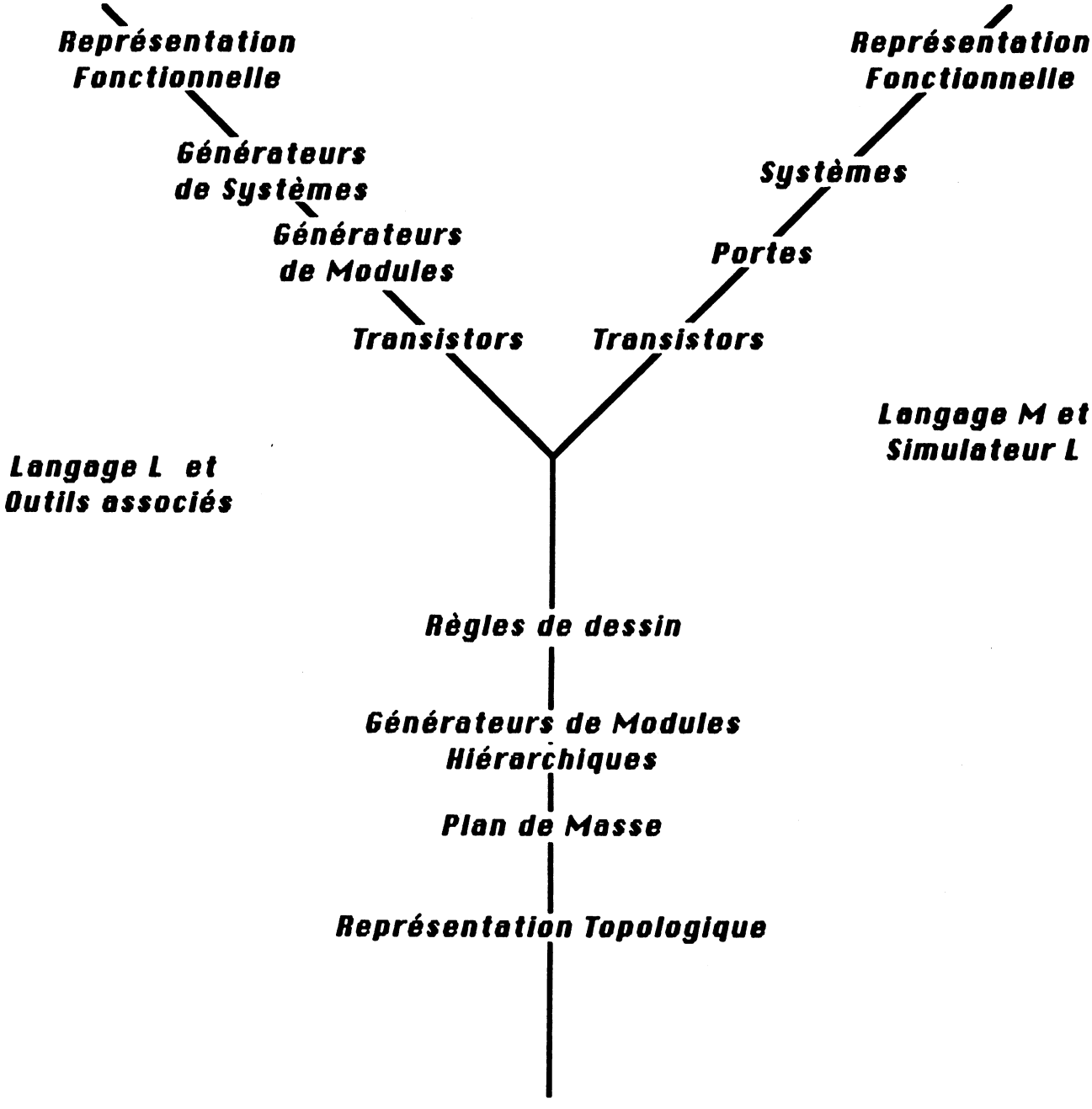


Fig. II.4 : Représentation SDL

— **Des compacteurs :** Des compacteurs sont disponibles permettant de générer des cellules indépendamment de la technologie (le système travaille dans un mode symbolique) et suivant les règles technologiques, il étendra (transformation de segments en rectangles) puis compactera le dessin obtenu. Il existe des compacteurs suivant les axes principaux (X, Y) et leurs bissectrices (XY et YX).

Le système permet de définir des *"Icônes"*. Ces icônes sont des représentations simplifiées de cellules permettant de remplacer l'original, ayant l'apparence de la fonction symbolique de la cellule et le même nombre d'entrées/sorties. Exemple : pour la représentation d'une cellule "inverseur" (INV) on utilisera le dessin symbolique suivant :

Ceci permet de gagner un temps précieux pour le dessin de cellules non utiles à cet instant. GDT est utilisé dans l'industrie et permet d'écrire des compilateurs de silicium.

II.2.1.5 S.D.A. - SKILL

Le système SKILL [LaW86] offre à l'utilisateur des générateurs de modules paramétrés et permet à l'utilisateur de créer ses propres générateurs.

On peut caractériser SKILL en quelques points :

- Un langage interactif

  Le langage SKILL est interactif, en effet il permet à un utilisateur de voir immédiatement sur un écran graphique les effets d'une action, d'une modification.

- Des possibilités d'interface utilisateur

  SKILL offre un grand nombre d'opérations de bases permettant à un utilisateur d'intervenir sur le système. Ainsi au cours de l'exécution d'un programme SKILL l'utilisateur pourra modifier les valeurs des variables d'environnement grâce à des primitives simples telles que *getpoint*, *getresponse* qui demandent à l'utilisateur de *cliquer* un point sur l'écran.

- Toutes les possibilités d'un langage de programmation

  SKILL est une extension du langage IL. Il offre toutes les possibilités d'un langage de programmation, immergé dans LISP (on retrouve dans IL la boucle et la forme conditionnelle classique, ainsi que le *foreach*, permettant d'appliquer à tous les éléments d'une liste une fonction ou bien le *setof*, permettant de

selectionner dans une liste d'éléments ceux qui répondent à une certaine condition).

- Etre lié au système résident.

  SKILL permet d'accéder à toutes les possibilités du système résident de conception. Ainsi on peut demander depuis SKILL l'éxécution d'un extracteur externe, ou bien d'appliquer un programme pour optimiser un ensemble d'équations logiques depuis une procédure SKILL, etc...

- Les accès à une base de données de conception.

  SKILL offre aussi des outils de gestion d'une base de données. Ainsi depuis une procédure SKILL il est possible d'interroger la structure de données sur les relations entre les différents objets.

En résumé on peut dire que SKILL permet d'écrire des générateurs de modules et des cellules paramétrées, permettant l'usage de *Macros*, offrant un langage interactif, permettant d'accéder facilement à des outils déjà existants.

II.2.1.6 SPARCS - OCT - VEM

SPARCS, OCT et VEM sont les trois parties importantes d'un système symbolique de conception développé à Berkeley [New86].

On trouve au centre de ce système la base de données orientée objet OCT. Sur celle-ci vont évoluer une interface graphique, VEM, qui agira sur la base de données et le système symbolique d'assemblage SPARCS (voir Figure II.5 [b]). SPARCS va établir un graphe de contraintes à partir du placement relatif des éléments à assembler et à l'aide d'une table des règles d'assemblage. Le système trouve alors le chemin critique de ce graphe, place les éléments n'appartenant pas à ce chemin, et identifie les contraintes "bloquantes". On peut remarquer que la base de données OCT permet de mémoriser et de gérer des contraintes explicitement données, et de plus toutes les contraintes peuvent être éditées à l'aide de l'interface VEM.

Nous allons décrire ici plus particulièrement OCT qui s'apparente aux mécanismes de gestion des données du système NAUTILE (voir chapitre III).

OCT [Moo86] est une structure de données, accompagnée des outils nécessaires à sa gestion, adaptée à la CAO de VLSI. Ce système offre une structure de données simple pour stocker toutes les informations relatives aux différents aspects de la conception d'un circuit intégré. L'unité de base dans ce système est la cellule. Classiquement une cellule peut être un simple transistor ou le circuit dans son ensemble.

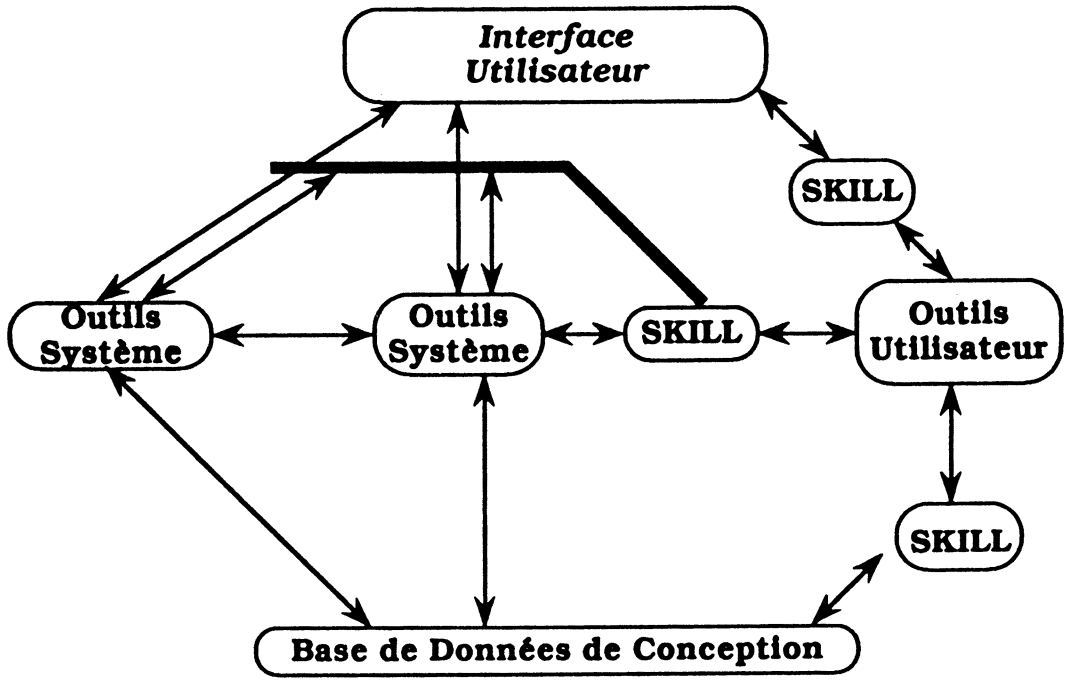


Fig. II.5.a : L'environnement de Skill

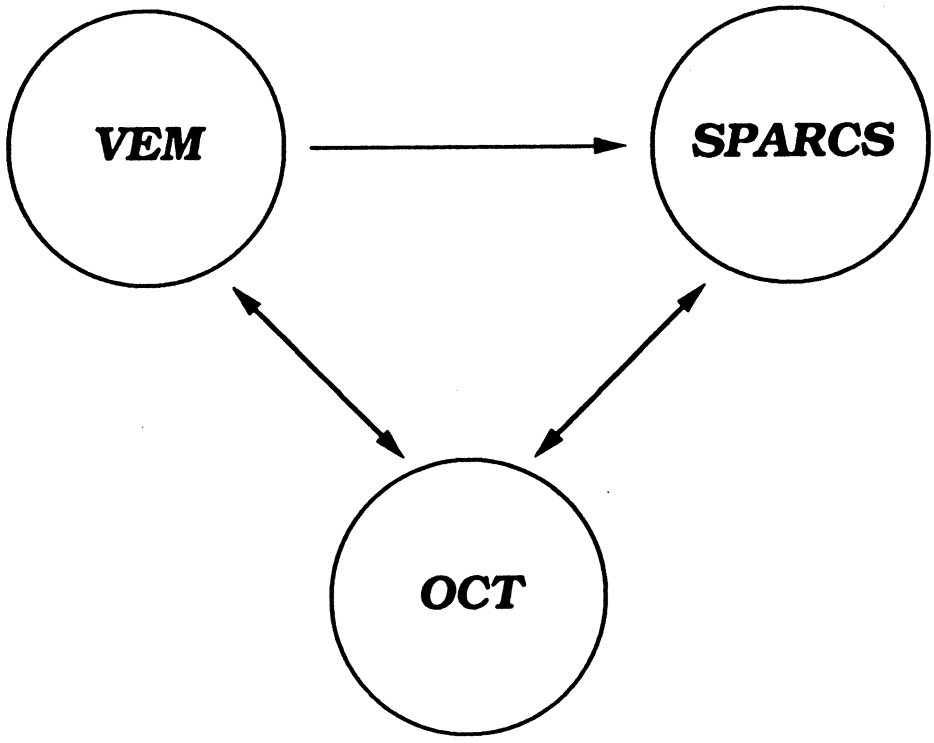


Fig. II.5.b : Le système symbolique de conception : SPARCS

Une cellule est représentée par ses différents aspects (vues), dépendant du type de conception que l'on souhaite faire. La vue peut être :

— symbolique : dans ce cas toutes les informations sur le placement ne sont pas explicites et peuvent être modifiées

— physique :

- la vue topologique classique
- une vue logique ou électrique : prévues pour la simulation.

Cette cellule va contenir des instances d'autres cellules qui contiendront d'autres instances, etc... Pour un certain nombre d'applications, il est souhaitable de pouvoir couper la hiérarchie en un point et de remplacer celle-ci par une vue simplifiée et spécifique du processeur qu'on cherche à appliquer à la cellule (exemple : pour un éditeur graphique ce sera une boîte englobante ou une icône (voir GDT); pour un D.R.C il s'agira des rectangles bordant la cellule et pouvant amener des conflits; pour un routeur, les connecteurs nécessaires au routage,...).

OCT utilise ici le terme de *facettes*. La facette est un élément fondamental de OCT. On peut ouvrir, éditer une facette particulière de OCT indépendamment des autres facettes d'une vue d'une cellule, ou des autres vues. Ces facettes sont en fait un ensemble d'objets attachés les uns aux autres. Ces objets peuvent être attachés à plus d'un objet à la fois.

Exemple : On pourra construire une facette de routage, en attachant à un même réseau différents connecteurs.

Un certain nombre d'objets existent dans OCT :

— les *facettes* : elles contiennent le nom de la cellule à laquelle elles se rattachent, la vue et même la facette;

— les *boîtes* : elles comprennent deux points;

— les *cercles* : un centre et des angles de départ et d'arrivée;

— les *chemins* : une suite de points et une largeur;

— les *polygones* : un ensemble d'arêtes;

— les *coins* : un point de départ et d'arrivée;

— les *points* : deux coordonnées;

— les *labels* : permettent d'associer un label à un objet quelconque;

— les *réseaux* : permettent d'associer des terminaux à relier pour un routage;

— les *terminaux* : une instance et un connecteur;

— les *sacs* : permettent de manipuler des ensembles d'objets. On pourra les utiliser par exemple dans un éditeur, pour manipuler les objets sélectionnés;

— Les *niveaux de masques* : ceci correpond aux niveaux physiques (aluminium, polysilicium,...);

— Les *propriétés* : ceci décrit les propriétés attachées à des objets;

— Les *instances* : son nom, son origine, ses transformations, ...

Cette structure de données est très intéressante par les nouveaux concepts introduits, et permet une grande liberté d'utilisation. On la retrouve dans deux systèmes : SPARCS et VEM

II.2.1.7 DPL

DPL (Design Procedure Language) [BaH80] a été développé par le MIT (Massachussets Institute of Technology) comme un langage de conception procédurale pour les circuits intégrés. Beaucoup de concepts de la conception procédurale ont été exprimés pour la première fois grâce à DPL.

DPL comprend une base de données orientée objet et 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 manipulatons de la base de données.

Le processus de conception en DPL va passer par les étapes suivantes [New86] :

- 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.

On retrouve dans DPL un certain nombre de notions de base :

- Le type (ou générateur)

  Le type 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)

Le prototype est une structure construite à partir d'un type

- 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 II.6 [a], 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, B contiendra tous les paramètres de C plus éventuellement d'autres paramètres (voir Fig II.6 [b])

- Les instances placées

  Les instances 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.

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

— Evaluer les paramètres donnés

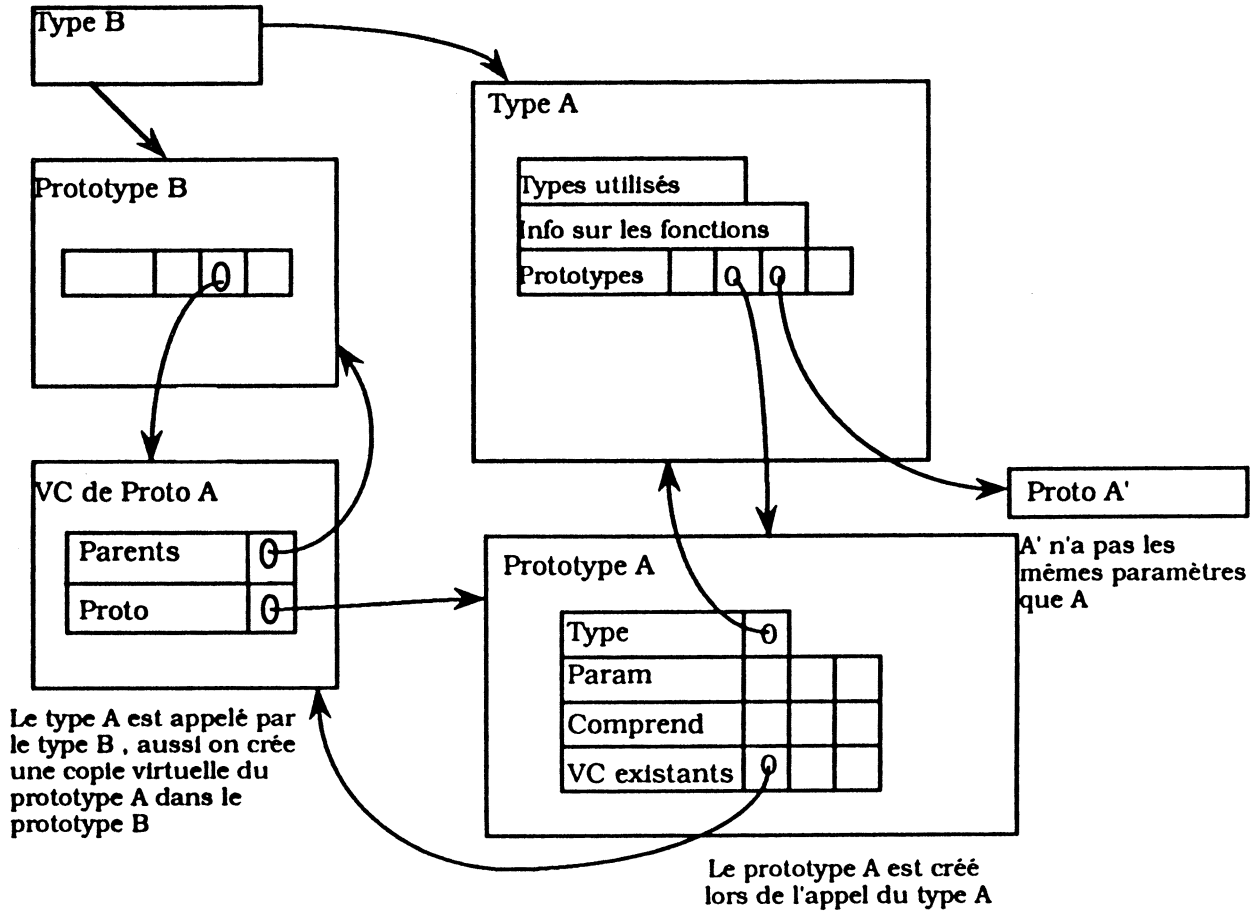
— Résoudre les contraintes, attribuer des valeurs aux variables. Si des variables restent sans valeur attribuée, leur donner la valeur par défaut.

— 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.

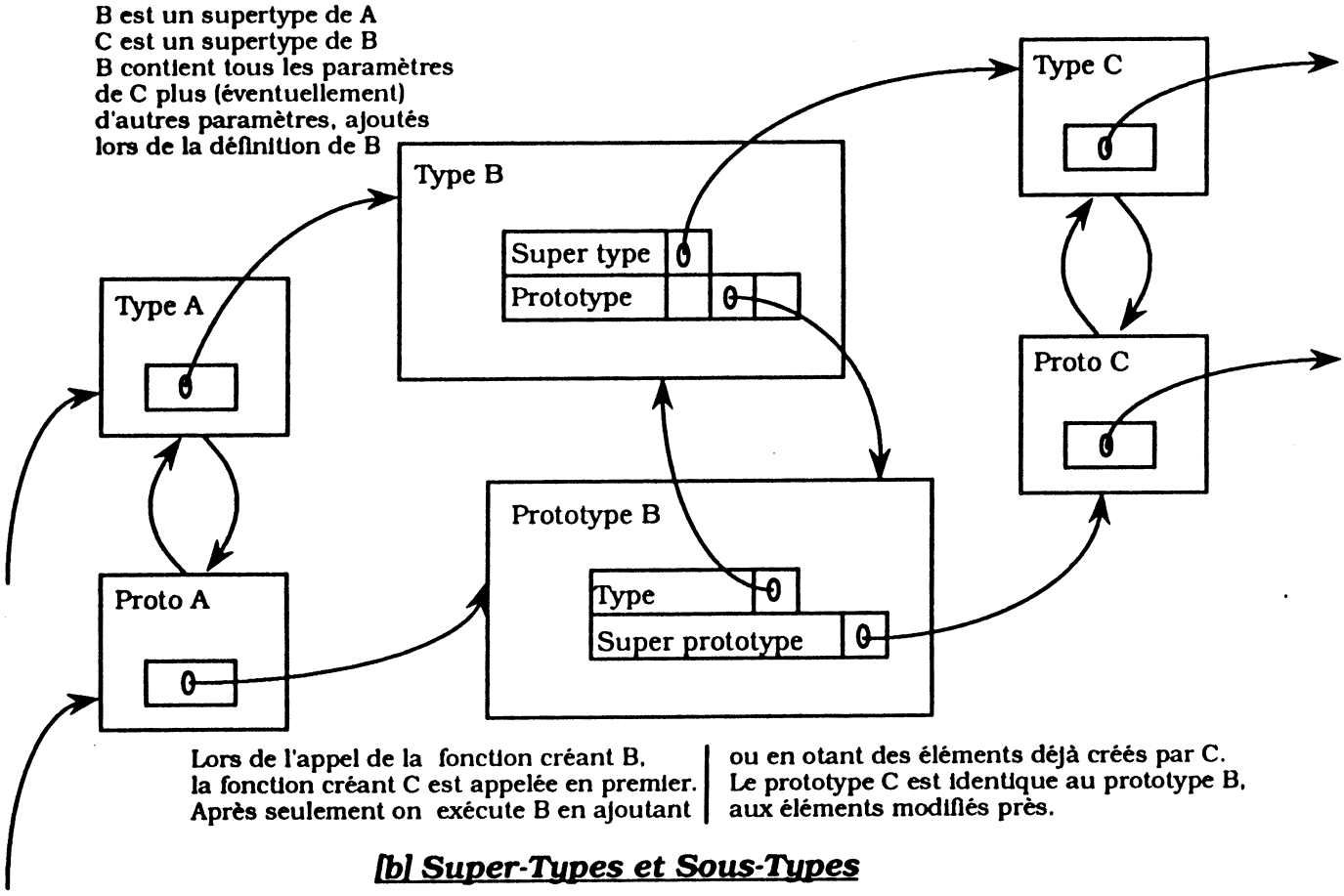
— Donner une instance du prototype créé avec les transformations appropriées.

Les concepts de *type, prototype, supertype, copie virtuelle et instance,* relatifs au langage DPL se retrouvent aussi dans beaucoup de langages orientés objet avec quelquefois des noms différents (SMALLTALK [GoR83], FLAVORS (MIT)). On peut noter d'ailleurs que DPL fut un précurseur dans le domaine de la conception de circuits intégrés "programmés". Beaucoup de systèmes se sont par la suite inspirés de DPL .

On verra dans le chapitre III la façon dont ces concepts ont été utilisés pour le système NAUTILE.



[a] Types et Prototypes



[b] Super-Types et Sous-Types

Figure II.6 : Exemple de représentation de la hiérarchie DPL ([BaH81])

II.2.2 Les systèmes intelligents (les compilateurs de silicium)

Le terme de compilateur de silicium est employé pour désigner un programme produisant le dessin des masques d'un circuit intégré, à partir d'une description de haut niveau. En général un compilateur de silicium permettra d'automatiser certaines des étapes (voire toutes) de la conception d'un circuit.

II.2.2.1 Niveaux d'abstraction

On peut différencier les descriptions de circuit suivant trois principaux domaines :

— Domaine comportemental

— Domaine structurel

— Domaine géométrique

Pour chacun de ces domaines, plusieurs niveaux d'abstraction sont possibles, par exemple pour le domaine géométrique :

— Génération du plan de masse

— Placement de cellule

— Routage

— Dessin de masque

Concevoir un circuit ne nécessite pas forcément de spécifier ces différents domaines ou niveaux, mais chaque méthodologie de conception se divisera en diverses étapes et chacune de ces étapes utilisera un ou deux niveaux de description.

II.2.2.2 Les compilateurs de silicium

On pourra classer dans cette catégorie trois principaux types de système [GeJ87]:

— Les environnements pour des générateurs de modules (appelés aussi compilateurs de compilateurs)

— Les générateurs de plan de masse, appelés aussi compilateurs de silicium structurels.

— Les outils traduisant une description algorithmique en dessin des masques sans intervention du concepteur, appelés aussi compilateurs de silicium comportementaux.

La figure II.7 représente la classification de ces différents compilateurs. On peut détailler ces différents types de compilateurs :

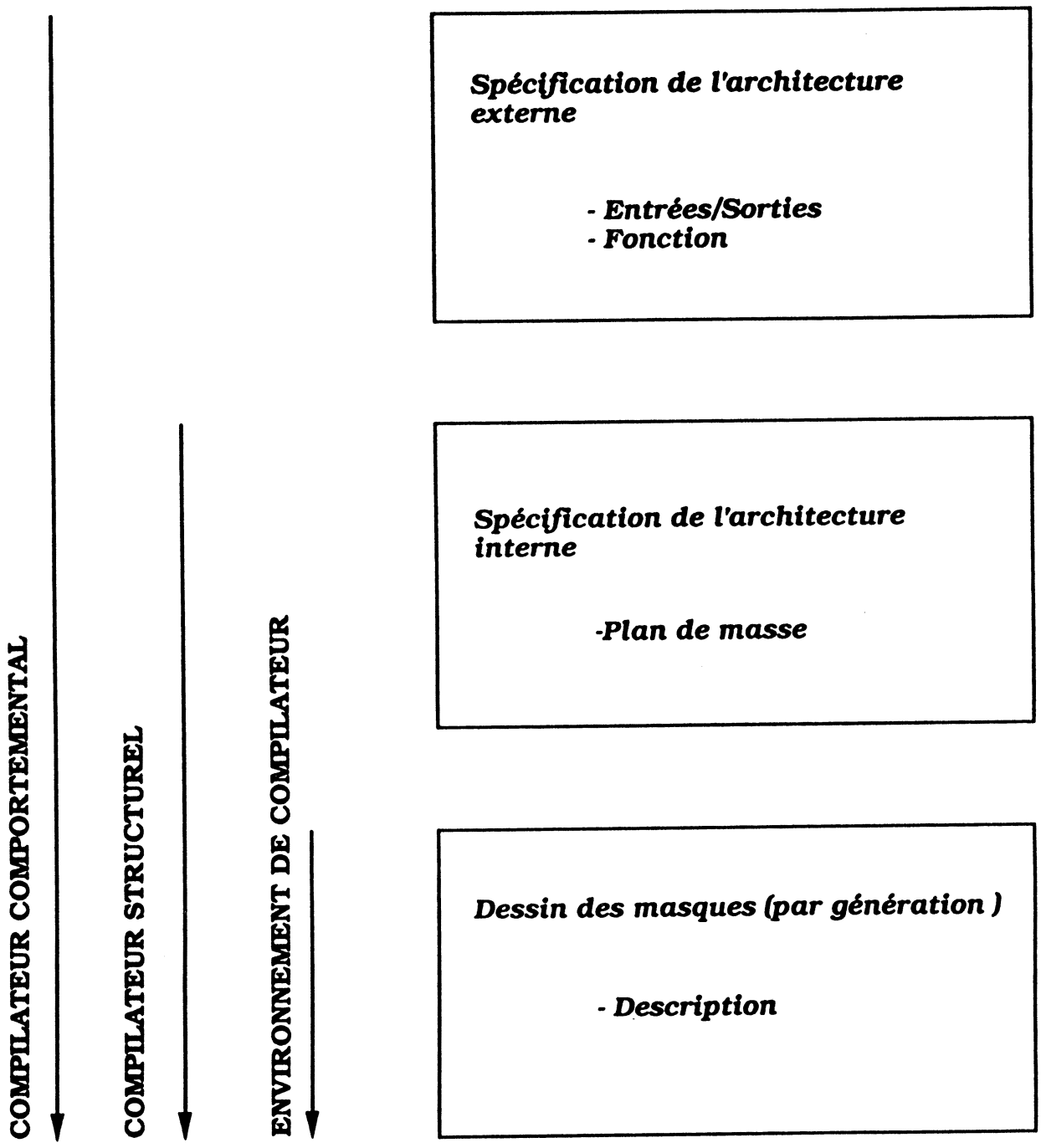


Fig II.7 ; Compilateurs de silicium et processus de conception

- Les compilateurs de compilateurs

Ces types d'outils sont en fait des environnements de programmation pour la conception d'un circuit intégré. Ils sont en général composés de :

- un langage procédural pour le dessin des masques,
- une bibliothèque de cellules,
- une bibliothèque de générateurs de cellules.

Le concepteur va devoir manipuler des générateurs de modules (GDT, SKILL et DPL entrent dans cette catégorie, voir II.2.1)

- Les compilateurs structurels

Pour ce type de compilateurs le concepteur va devoir spécifier la structure interne du circuit. Des outils spécialisés sont utilisés pour générer le dessin des masques et l'assemblage des cellules est effectué automatiquement.

Un tel système permet de réaliser le dessin des masques et de faciliter les autres étapes de la conception (il offre un environnement intégré pour la conception et la vérification d'un circuit). Ce type de système permet de garder un bon contrôle de l'organisation et de la structure du circuit à concevoir. En revanche il nécessite que le concepteur soit un "expert" en matière de conception. Par exemple le concepteur doit founir une stratégie globale pour minimiser les surfaces nécessaires aux diverses connexions, utilisées par le routeur global. On peut citer pour ce type de compilateurs SCI (Silicon Compiler Inc) et SST (Seattle Silicon Technology)

- Les compilateurs comportementaux

Ce type de compilateur part d'une description de haut niveau. Cette description peut être indépendante de l'organisation interne du circuit à générer. Cependant la plupart de ces compilateurs sont spécialisés dans une classe de circuits particulière.

On retrouve dans [GeJ87] une description des divers compilateurs de silicium existants, et on peut citer MacPitts, Cathedral et First comme compilateurs comportementaux. MacPitts part d'une description de type Lisp d'un circuit et génère le dessin des masques correspondant. Il produit des circuits de type microprocesseurs et utilise une architecture cible. Cathedral et First produisent surtout des circuits spécialisés dans le traitement numérique du signal.

II.2.2.3 SYCO

Nous allons décrire ici quelques points essentiels du compilateur SYCO [JVJ86] qui doit utiliser le système NAUTILE pour la génération du layout (voir chapitre V). SYCO appartient à la classe des compilateurs comportementaux.

— 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 entre elles et 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.

Le chapitre suivant décrit le système NAUTILE qui de façon comparable à GDT,

SKILL ou DPL offre un environnement pour l'écriture de compilateurs de silicium. Il faut préciser que le système NAUTILE a permis d'écrire les modules générateurs utilisés par le compilateur comportemental SYCO.

III. NAUTILE : Les principes fondamentaux

Introduction

Ce chapitre présente un nouveau système pour la conception de circuit intégrés NAUTILE (New AUtomatic and Technology Independant Layout Environment [BCH87][JBG88]). Dans ce chapitre sont exposées les notions de base du système, ses fondements qui ont conduit au système proprement dit, ainsi que ses fonctionnalités essentielles.

III.1 Les objectifs du projet NAUTILE

III.1.1 Les fonctionnalités

Le but du système NAUTILE est la réalisation d'un environnement complet de conception de circuits intégrés. Il doit à cette fin apporter une structure de données contenant les différents objets composant les diverses représentations d'un circuit, et les primitives de manipulation de cette structure. Le système en lui-même doit offrir :

— un outil de conception graphique,

— un outil de conception textuelle,

— la définition d'un langage textuel permettant de réaliser :

- un assistant de conception,
- des générateurs divers,
- des compilateurs.

La figure III.1 donne une représentation générale du système NAUTILE et de son organisation.

On peut citer quelques unes des idées directrices qui ont guidé la réalisation du système :

— avoir un environnement complet de conception;

— permettre au système de s'interfacer avec d'autres systèmes existants;

— être indépendant de la technologie;

— avoir une équivalence entre les descriptions graphique et textuelle d'un circuit;

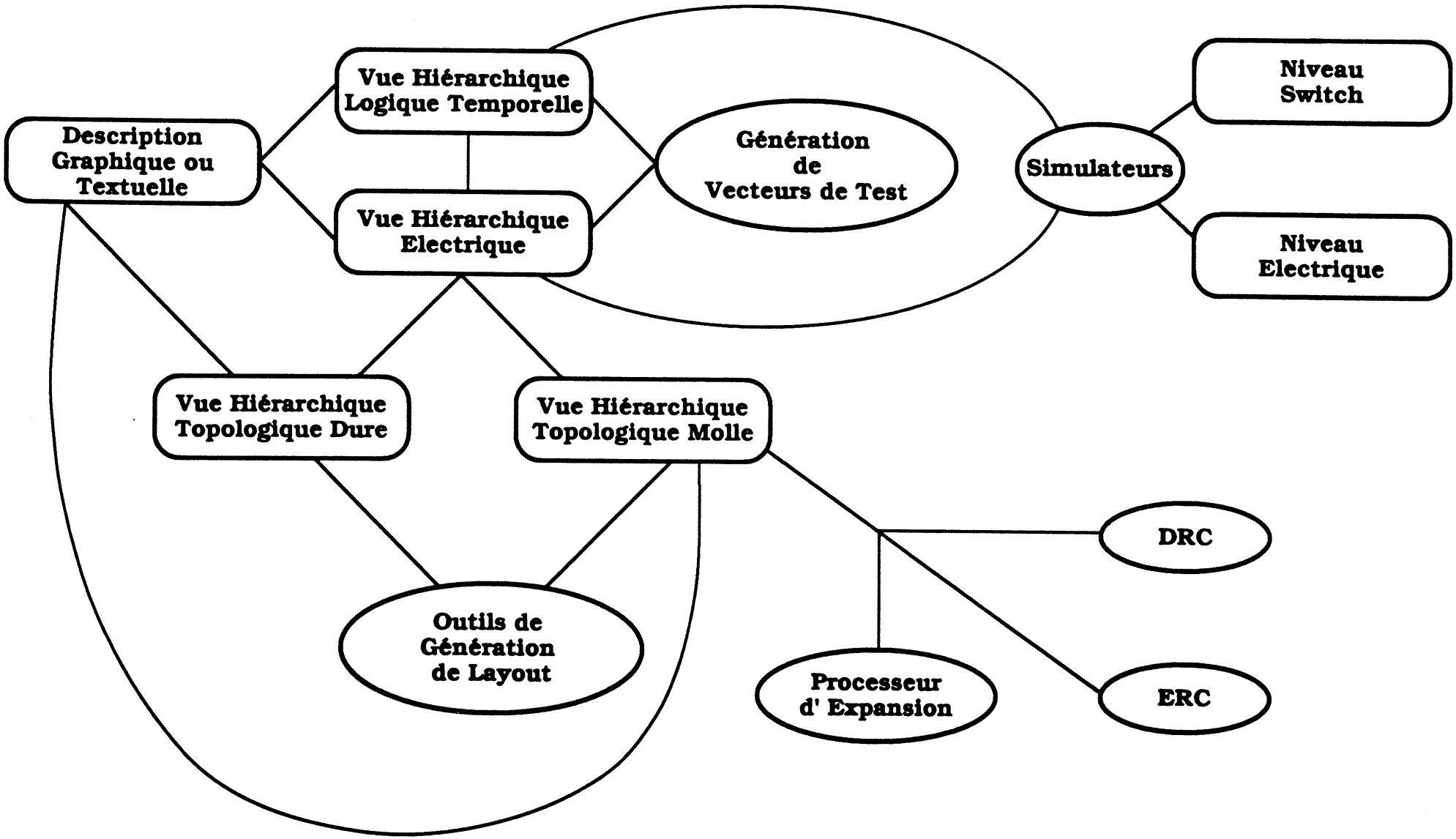


Figure III.1. : Représentation générale du système NAUTILE

— assurer la gestion de différentes vues d'un circuit (électrique, topologique,...), permettre un traitement unifié des différentes vues et gérer la cohérence des vues entre elles;

— offrir un système symbolique hiérarchique.

Pour cela NAUTILE comporte des outils d'aide à la conception, des outils de vérification et d'optimisation ainsi qu'une base de données comportant une structure de données, l'ensemble des primitives de description et de manipulation des objets et un ensemble d'objets prédéfinis permettant de masquer la technologie (bibliothèque de motifs voir III.1.3).

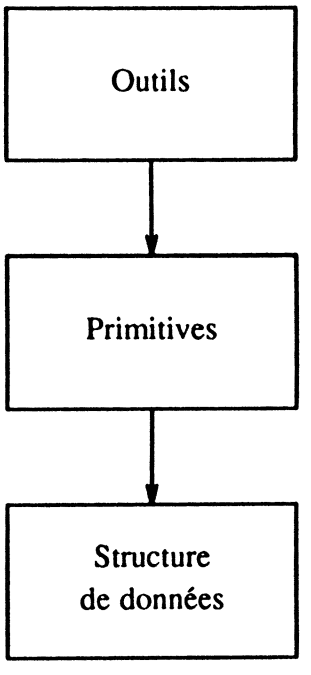


Figure III.1.bis : Les différents niveaux d'utilisation de NAUTILE

De fait le système NAUTILE doit permettre de gérer une base de données orientée objet [JBG88], et disposer des primitives nécessaires à cet effet, ainsi que d'un ensemble minimum d'outils évoluant au-dessus de ces primitives (voir Figure III.1.bis). Le système offre des outils graphiques et textuels, des outils de génération, de visualisation et d'édition (modification vérification,...). Il faut citer aussi les simulateurs logique, interrupteur ou électrique, les générateurs de vecteurs de test, des vérificateurs divers (topologique,électrique, logique, ...) des processeurs d'expansion permettant de générer différents formats de description (GDS II, CIF, GL1,...). La plupart de ces outils ne seront pas développés dans le cadre du projet NAUTILE. Le plus souvent, il s'agira d'interfacer des outils existants avec NAUTILE.

Dans la structure de données, l'entité de base manipulée est la cellule, chaque cellule réalise une fonction particulière. La représentation adoptée est hiérarchique, chaque cellule pouvant en utiliser d'autres pour sa propre définition. Chaque cellule correspond à une fonction et il est possible de décomposer une fonction que l'on cherche à réaliser en différentes sous-fonctions réalisées par des cellules *Filles*. Au niveau le plus bas de la hiérarchie (les *feuilles de l'arborescence*), les cellules s'appellent des motifs. Ces motifs seront soit dessinés à la main soit prédéfinis dans le système (bibliothèque de motifs, motif généré par un outil,...)

III.1.2 Choix d'un langage

Un des principes moteurs du projet NAUTILE fut la nécessité de disposer d'un système permettant de concevoir des circuits indépendamment de la technologie et utilisant un langage unique pour traiter les différents aspects de la conception. Ceci a conduit à envisager un système ayant plusieurs représentations, fortement hiérarchisé et offrant le maximum de possibilités de paramétrisation.

On peut recenser trois types généraux de langage [Anc84] [Rou87]:

— Des langages *composites* (immersion de primitives dans un langage donné)

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 Le_Lisp [CDD86] ou SmallTalk [GoR83]) et on peut programmer tous les outils dans un langage unique (celui choisi). On peut relever cependant un inconvénient provenant de la complexité de programmation de tels langages: le concepteur doit être lui-même un programmeur.

— Des structures dynamiques et paramétrées (exemple L [Bur86] et SKILL [LaW86]).

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 dans la puissance de paramétrisation.

— Des descriptions statiques obtenues par génération de programmes écrits dans un langage quelconque (exemple Lucie [Pai85] ou CIF [SpL80]).

Cette méthode permet une simplification des problèmes rencontrés, cela permet d'utiliser un langage quelconque pour la génération du code de description. Cependant ceci implique un volume de code important.

Parmi toutes ces possibilités, nous avons fait le premier choix. En effet on disposait du langage Le_Lisp avec tout son environnement et ses possibilités pour le développement. Ceci 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.

III.1.3 Les motifs et les cellules

Une des principales composantes du système NAUTILE est d'offrir une structure de données permettant de s'adapter à tous les types d'outils existants. Cette structure de données consiste en une organisation hiérarchique de cellules. Une cellule pourra aussi bien être un simple transistor que le circuit dans son ensemble (voir Figure III.2 et III.3).

- Les motifs : atomes de la structure

  L'élément de base de la structure de données est le motif. Le motif est une cellule se situant au bas de la hiérarchie qui peut être définie soit manuellement soit par le système (prise dans des bibliothèques, créée par un outil quelconque,...). Ces motifs peuvent être définis sous plusieurs aspects différents (électrique, logique, topologique,...). Un circuit est un assemblage de motifs. Le système NAUTILE se caractérise par un traitement unifié avec un langage unique des différentes vues de ces motifs.

  Les motifs devront comporter toutes les vues sur lesquelles le concepteur a l'intention de travailler. Les outils travaillant de façon hiérarchique, ceci suffira pour qu'ils fonctionnent sur toute la hiérarchie.

- Les cellules

  L'élément structurel *cellule* se retrouve à tous les étages de la hiérarchie. Une cellule est composée d'un assemblage de motifs, d'appels à des générateurs de motifs (routeurs, générateurs de PLA's,...) et d'appels à d'autres cellules.

III.1.4 La forme générale d'une cellule

Une cellule sera décomposée en deux grandes parties :

- La partie externe : le squelette

  La partie externe constitue une interface entre la cellule et les autres cellules ainsi qu'avec le *monde extérieur*. Elle permet de connaître l'encombrement d'une cellule, ses possibilités de connexion avec d'autres cellules, ses dernières modifications,... Ceci permet de manipuler la cellule sans tenir compte de ce

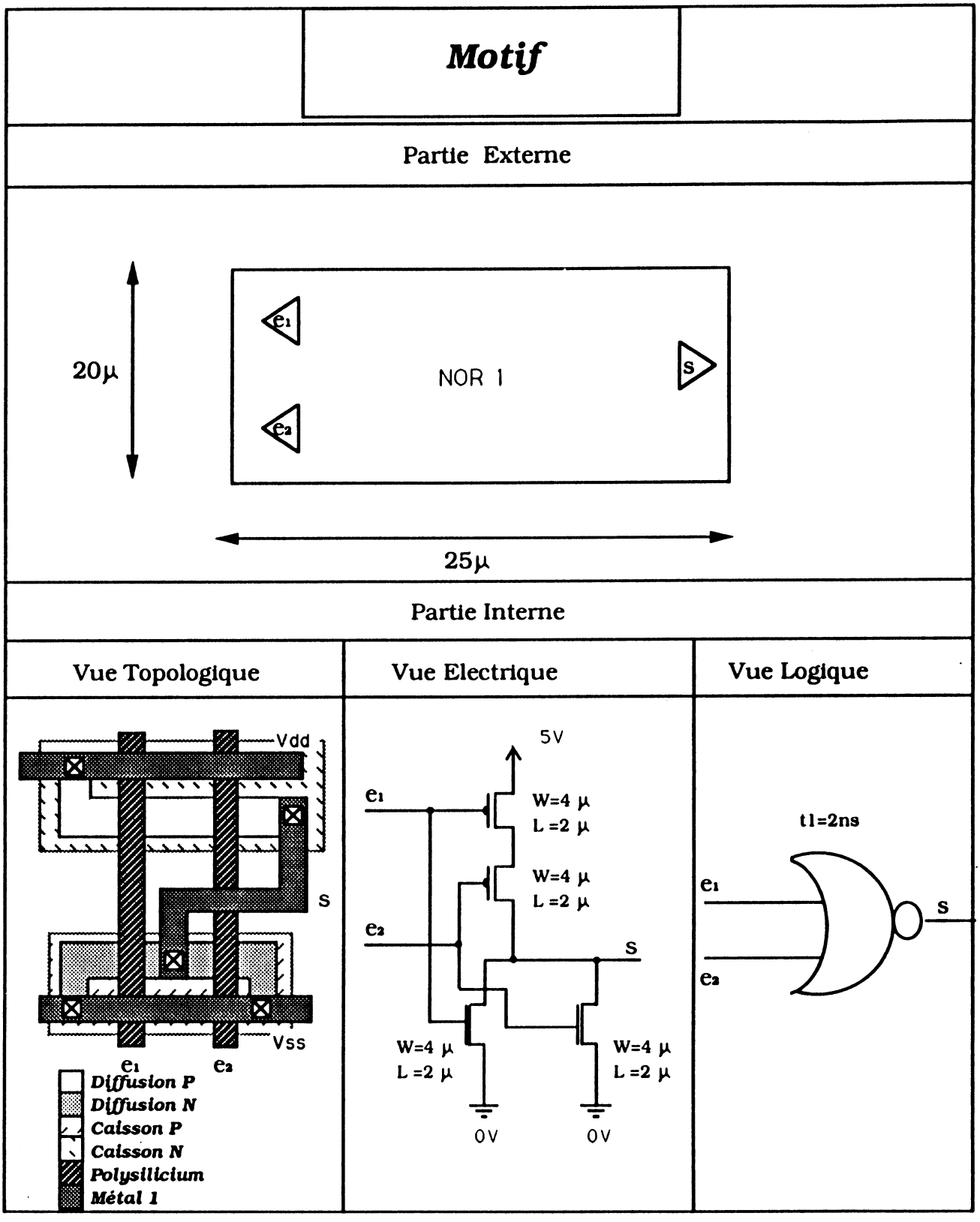


Figure III.2 : Exemple de représentation structurelle d'un Motif

qu'elle contient.

- La partie interne :

  Cette partie n'est pas toujours présente en mémoire, et concerne les détails internes des différentes représentations physiques de la cellule. S'il s'agit d'un motif on retrouvera ici les différentes vues physiques du motif (topologique, logique,...) sinon on y trouvera la vue de construction. On détaillera ces différentes vues dans la section III.2.

III.1.5 Communication avec les systèmes externes

Une des grandes originalités du système NAUTILE réside dans sa possibilité de gérer facilement des vues venant de systèmes externes. On utilise par exemple couramment des vues topologiques de motifs dessinées à l'aide de l'éditeur LUCIE [Pai85], utilisées sans traduction en tant que squelette (on ne traduit que la vue externe de la cellule écrite en LUBRICK [Sch83]), qui comporte les connecteurs et la boîte de la cellule, et ayant pour vue électrique une représentation R.N.L. [Dri86]

III.2 Les différentes vues

Ainsi qu'on l'a vu précédemment (III.1.4), le système NAUTILE est basé sur une notion d'objets multi-vues, c'est-à-dire que dans une cellule ou dans un motif on peut retrouver différentes vues logique, électrique,... L'utilité de ces vues est décrit dans les paragraphes suivants.

III.2.1 Représentation logique

- Pourquoi ?

  La représentation logique sert à manipuler des notions logiques dans la conception du circuit. Une vue logique est souvent associée à une vue temporelle permettant d'avoir une simulation chronologique du fonctionnement du circuit et ainsi d'avoir une idée des performances et de la vitesse de ce circuit. La représentation logique permet d'avoir un langage et un format de données pour des simulateurs logiques.

- Comment ?

  La représentation logique d'un motif, consistera à associer à celui-ci une fonction logique. Ceci peut se traduire par une table de vérité associée comprenant des valeurs d'entrées et de sorties ou plus simplement par une fonction classique de la logique. Les valeurs possibles prises par les entrées/sorties du motifs, peuvent être dans l'ensemble : [0,1,X,↓] [Cai87]

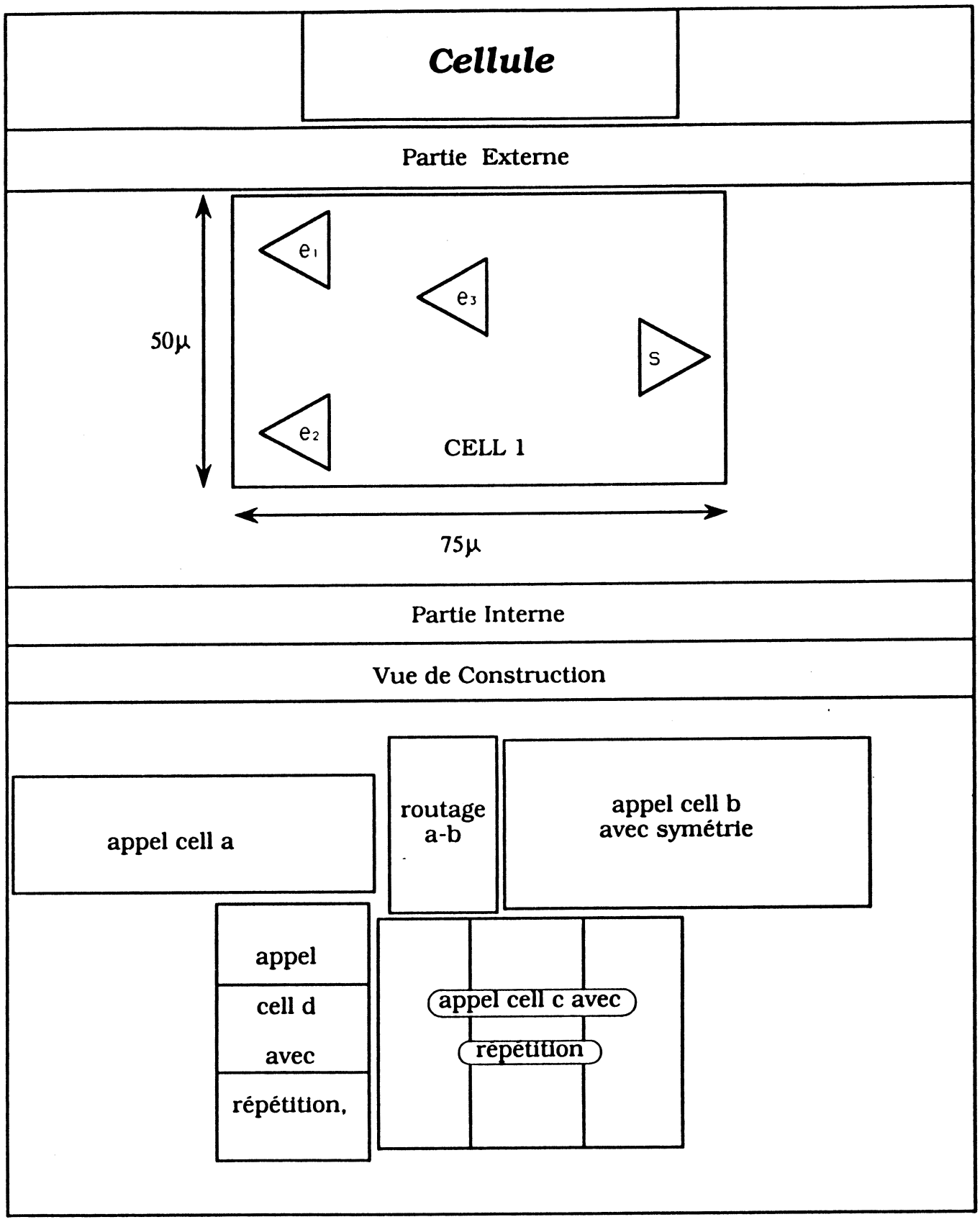


Figure III.3 : Exemple de représentation structurelle d'une Cellule

III.2.2 Représentation électrique

- Pourquoi ?

La représentation électrique est utilisée principalement à deux fins :

— Une description électrique du circuit correspondant aux représentations topologiques et logiques.

— La simulation du fonctionnement soit électrique (SPICE) soit switch multivaluée [Cai86].

Dans le cadre de la simulation ceci permettra de vérifier si le circuit conçu est viable du point de vue électrique.

- Comment ?

Les objets de la représentation électrique sont directement dépendants des objets nécessaires à la simulation. Il faut pouvoir décrire des transistors électriquement, donc parler de résistances et de capacités et pouvoir donner des valeurs aux différents noeuds du schéma électrique induit. Ces valeurs de capacité et de résistance peuvent se déduire des valeurs *topologiques*.

Exemple : Il est possible de retrouver les valeurs de capacité et de résistance d'un transistor à partir de la largeur et de la longueur du canal et du type du substrat (type du transistor)

III.2.3 Représentation topologique

- Pourquoi ?

La représentation topologique constitue la représentation classique principale d'une cellule. En effet il s'agit ici du dessin des masques, c'est à dire du but même de la conception d'un circuit.

- Comment ?

Une vue topologique sera composée :

— de rectangles

— de polygones

Cette vue n'existera en fait que dans les motifs ; pour les cellules, on peut retrouver la vue topologique grâce à la vue de construction, par composition des motifs appelés ou des cellules appelées (récursivement, on peut obtenir une vue topologique *à plat*).

III.2.4 La vue de construction

La vue de construction est liée aux arbres d'appels. C'est la vue que l'on retrouve dans les cellules et qui est constituée d'appels à d'autres cellules ou motifs. Cette vue va permettre de créer une vue de la cellule dite *molle* par opposition à la vue dite *dure*. En effet les appels aux différents générateurs, routeurs, placeurs,... ne seront pas systématiquement expansés sous forme de placement effectif et d'instance de cellule. Dans ce cas, il est possible de conserver des paramètres et d'obtenir des placements relatifs à d'autres cellules sans avoir encore fixé leur boîte englobante, des générations de cellules dépendant de paramètres non encore fixés au moment de l'appel,...

Parmi les instances réalisées on peut trouver trois types différents d'instances :

- L'instance *classique*
- L'ombre
- L'élément partagé

III.2.4.1 L'instance classique

Il s'agit simplement d'un appel à une cellule avec son placement et les différentes actions à effectuer (voir III.3.3.1.2). Il faut noter qu'aussi bien l'ombre que l'élément partagé possèdent des propriétés similaires.

III.2.4.2 Les ombres

Les ombres sont des cellules spéciales permettant de définir une zone de superposition possible avec une autre cellule dans des couches indiquées. Cette possibilité évite à des outils de vérification un certain nombre d'opérations. En effet, l'outil saura immédiatement qu'à cet endroit une superposition est licite. Il est possible ainsi de préparer un passage de bus, de masse ou d'alimentation qui constitueront une seule cellule mais dont les ombres seront présentes sur plusieurs cellules. Ce peut être aussi un avantage pour l'unicité de la description électrique.

La présence du principe des ombres dans un système, permet de simplifier le principe de vérification pendant la conception d'un circuit.

On répertorie deux types principaux d'ombres [Rou87] :

- Les ombres de connexion : on entend par là, une prolongation des connecteurs situés à l'intérieur d'une cellule, jusqu'à la frontière et dans la direction indiquée par le connecteur (voir structure des connecteurs III.3.3.1.1)
- Les ombres de transparence : il s'agit de chemins de passage libre dans certaines couches technologiques, au dessus de la cellule.

Il existe un certain nombre de problèmes associés aux ombres dont les problèmes d'héritage d'ombres , les problèmes de modification d'une cellule (voir Figure III.4) et les problèmes de définition (quand l'utilisateur doit-il définir une ombre ? Celle-ci doit-elle être définie automatiquement à chaque nouvelle cellule ?).

Exemple :

La figure III.4 donne l'exemple d'une cellule C faisant une instance de A et de B. La cellule A comporte des connecteurs dits *prolongés*, ils vont servir à prévoir pour la suite des passages permettant de relier ces connecteurs à l'extérieur. L'assemblage de B avec A (en fait la superposition de B sur A dans C) permet de prévoir un passage de bus au dessus de la cellule A. Cette figure illustre les problèmes suivants :

— Assurer la cohérence en cas de modification de A.

— Pour chaque instance de C, il faut vérifier que le passage des ombres reste licite et déterminer quelles sont les ombres dont doit hériter la cellule faisant appel à C.

III.2.4.3 Les éléments partagés

Lors de la description topologique d'un circuit il est parfois nécessaire de partager des éléments entre deux cellules. Cependant ce partage doit trouver un équivalent dans la description logique ou électrique. La solution simple trouvée à ce problème consiste à déclarer cet élément partagé.

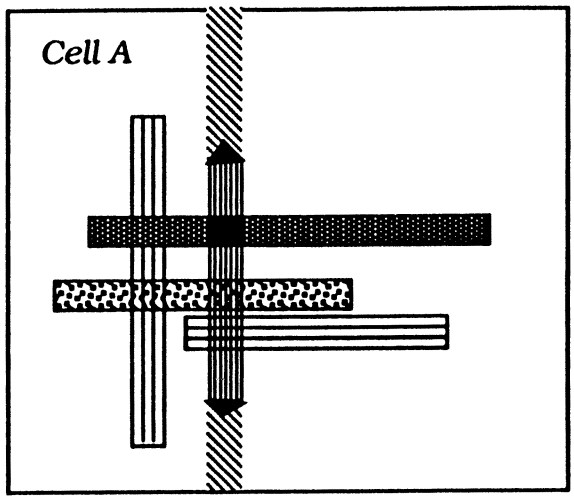
III.2.5 Les liens entre les différentes vues

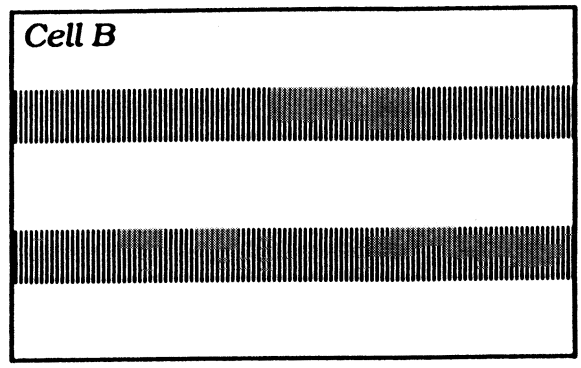
- Pourquoi

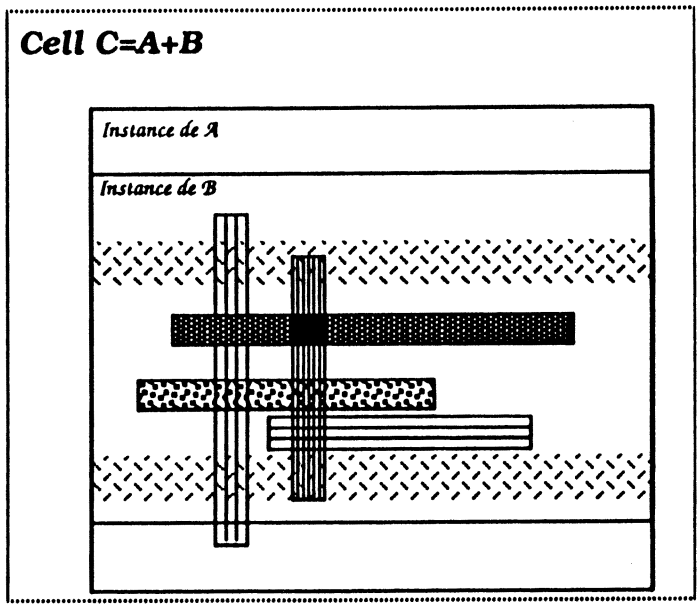
  Il est essentiel dans un système gérant différentes représentation (vues) de cellules, de pouvoir maintenir la cohérence entre les différentes vues. La solution adoptée dans NAUTILE est d'effectuer un traitement unifié de ces différentes vues. Ce traitement unifié restreint cependant les possibilités pour le concepteur de manipuler des vues physiques des cellules construites. Mais d'autre part, lors de la conception d'un circuit, le concepteur ne voudra pas forcément penser en matière de vue topologique ou électrique, mais plutôt de façon générale (se rapprochant d'une vue logique ou même fonctionnelle). C'est pour ces différentes raisons qu'il a été choisi d'utiliser un langage commun de description, et d'agir essentiellement sur une vue dite de *construction*.

- Comment ?

  Le principe d'un langage commun de description revient à définir toutes les fonctions et tous les outils agissant sur une représentation particulière, pour toutes les représentations. Ceci peut se traduire par exemple pour un routeur, agissant







Passage d'ombre d'alu1

Connecteurs prolongés (alu1 ou alu2)

Alu 1

Figure III.4 : Les Ombres

essentiellement au niveau topologique, par des équivalents électriques et logiques (création des connexions électriques requises et calcul des résistances/capacités par la sémantique électrique du routeur).

Le concepteur aura défini pour ses motifs les différentes vues topologique, électrique,... (on peut même imaginer, sous certaines réserves, des systèmes permettant de générer automatiquement, pour des motifs *simples* une vue à partir d'une autre). Pour pouvoir gérer ces différentes vues, on utilise différents principes de gestion de cohérence. On utilise des dates de création de cellule, des dates de modification et des dates d'appel de cellule. Si par exemple une date d'appel est antérieure à la dernière modification de la cellule, il existera' une possibilité d'incohérence qui sera signalée à l'utilisateur sans pour autant l'empêcher d'effectuer sa manipulation.

III.3 La structure de données

On doit faire ici une distinction entre trois types de structures utilisées :

- La structure d'une cellule

Il s'agit de la structure détaillée et interne des cellules et de la gestion des informations contenues dans ces cellules.

- La structure des définitions de cellules

La structure des définitions de cellules permet de définir l'organisation générale des cellules dans la structure de données et les rapports hiérarchiques entre cellules (voir Figure III.5).

- La structure des appels de cellules

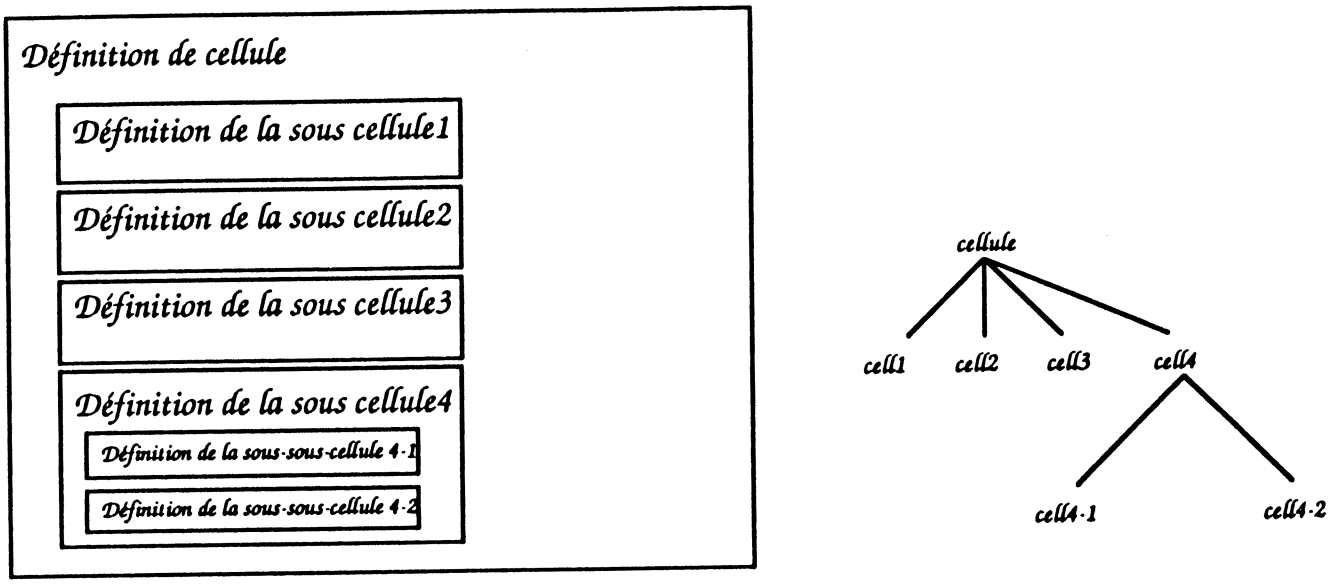
De même que l'on peut définir des cellules à l'intérieur d'autres cellules les appels de cellule définissent un arbre d'appel. La structure de cet arbre correspondra à la structure du circuit réalisé.

Exemple :

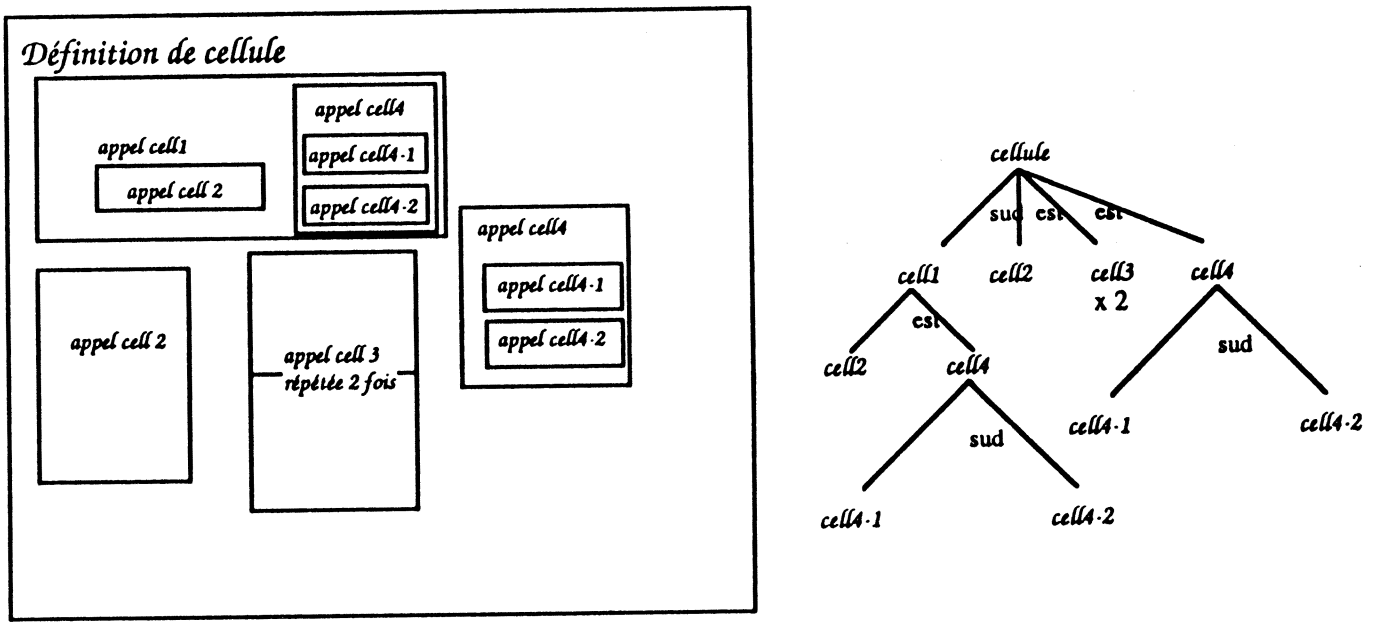
La figure III.5 illustre d'une part (III.5.a) les arbres de définition :

- définition de cellule *visible* uniquement à l'intérieur d'autres cellules (cell4.1 ne sera utilisable que dans cell4 ou dans la descendance de cell4 (cell4.2 et ses descendants).

et d'autre part les arbres d'appels (III.5.b) :



III.5.a : Les arbres de définitions



III.5.b : Les arbres d'appel

Figure III.5 : Représentation de hiérarchies dans NAUTILE

- l'arbre d'appel reflète exactement la structure du circuit représenté.

III.3.1 La structure des définitions de cellules

III.3.1.1 Gestion hiérarchique des noms

Le langage Lisp offre une facilité d'encapsulement des noms appelée package, qui permet de traiter les symboles Lisp avec des possibilités de regroupement.

La technique utilisée par Lisp réside dans le format des noms. Ces noms sont en fait des compositions de plusieurs noms séparés par ":", et peuvent par exemple refléter une hiérarchie. Dans le prototype du système Nautile on utilise pleinement ces possibilités (voir Figure III.7). On utilise une fonction *acces* permettant de sélectionner la cellule courante. Le nom de cette cellule donnera le préfixe courant utilisé pour rechercher ou travailler dans d'autres cellules : ainsi sera défini un arbre de définition de cellules.

III.3.1.2 Les arbres de définition

On appelle arbre de définition de cellules la hiérarchie obtenue dans les définitions de cellules. En effet il serait impossible de donner simplement un nom différent à chacune des cellules que l'on définit. On peut par exemple avoir besoin d'une cellule appelée *sync* pour un additionneur et d'une cellule presque identique mais néanmoins différente pour un multiplieur. On ne peut demander à l'utilisateur de numéroter ou de donner des noms totalement différents à ces deux cellules.

Pour résoudre ce problème on offre la possibilité de définir des cellules à l'intérieur d'autres cellules. Ceci s'apparente tout à fait à la définition de sous-procédure en Pascal. La gestion de ces sous-définitions s'effectue à l'aide des possibilité de package de Lisp à l'aide de fonction d'accès à des cellules (voir Figure III.6 et III.7).

Exemple :

Dans la figure III.6 on définit d'une part une cellule appelée *sync* dans le contexte de l'additionneur et, d'autre part, une cellule différente, appelée elle aussi *sync*, dans un autre contexte (le multiplieur). L'encapsulement des noms Lisp servira à faire la différence entre ces deux cellules: la première s'appelera Additionneur:sync alors que la seconde aura pour nom Multiplieur:sync. L'accès à l'additionneur ou au multiplieur déterminera le préfixe utilisé devant le nom de cellule. En revanche la cellule *mod*, définie dans un contexte englobant l'additionneur et le multiplieur, pourra être utilisé par l'un ou par l'autre et désignera le même objet.

Ce qui pourra donner l'exemple d'utilisation suivante :

```
(acces 'additionnneur)   ; Sélection de environnement:additionneur
(instance 'sync)         ; Instance de environnement:additionneur:sync
(instance 'mod)          ; Instance de environnement:mod
```

```
(fin-acces)              ; Retour dans l'environnement de départ
(acces 'multiplieur)     ; Sélection de environnement:multiplieur
(instance 'sync)         ; Instance de environnement:multiplieur:sync
(instance 'mod)          ; Instance de environnement:mod
(fin-acces)              ; Retour dans l'environnement de départ
```

Note : Les fonctions *acces, fin-acces et instance* seront décrites dans le chapitre IV.

III.3.2 La structure des appels de cellules

De même que la définition d'une cellule consiste à lui donner un nom et une place dans la hiérarchie permettant de la retrouver facilement, l'appel d'une cellule consistera à en matérialiser une occurrence [AnJ84]. La création de ces instances va définir un arbre d'appel qui va correspondre à la structure du circuit (voir Figure III.5).

III.3.3 La structure d'une cellule

III.3.3.1 Les objets manipulés

Avant de présenter la structure de données détaillée du système NAUTILE, nous allons décrire ici les objets de base manipulés dans la structure. La structure de ces objets est directement dépendante des besoins d'information sur ces objets. Toute la structure NAUTILE est orientée objet et les primitives du langage sont pour la plupart des sémantiques dépendant du type des objets.

III.3.3.1.1 Les connecteurs

Parmi tous les objets, les connecteurs sont peut-être les plus importants et ceux impliquant le plus de manipulations. En effet un circuit de quelques centaines de milliers de transistors comprendra quelques millions de connecteurs (il peut y avoir jusqu'à quatre connecteurs par transistor qui vont être répétés dans la hiérarchie sous des noms différents ou simplement dupliqués). Il est donc essentiel que toutes les manipulations sur les transistors soient optimisées (on peut considérer que dans NAUTILE la majeure partie du temps cpu est utilisée dans des opérations sur les connecteurs - voir section IV.2.1)

Les connecteurs font partie du squelette d'une cellule et caractérisent les relations possibles et existantes de cette cellule avec l'extérieur (les autres cellules). Ceci se retrouvera aussi bien aux niveaux électrique et logique, qu'au niveau topologique. C'est pourquoi les connecteurs appartiennent à la vue externe de la cellule et sont partagés par toutes les vues physiques de la cellule.

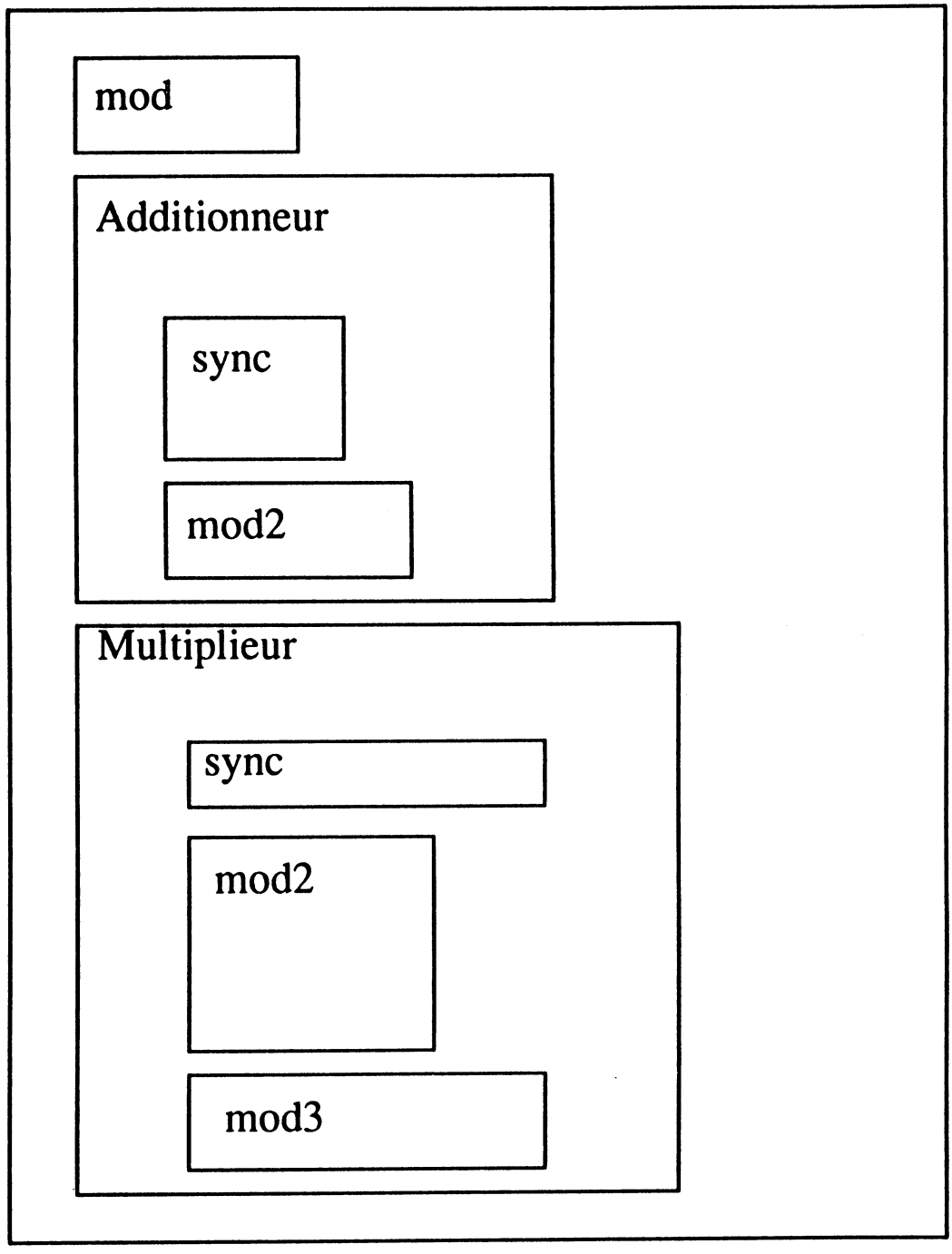


Figure III.6 : Exemple d'application des arbres de définition

Caractéristiques

On recense neuf informations nécessaires à la gestion des connecteurs

— 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 (voir section IV.2.1). L'utilisateur dispose de moyens variés pour retrouver des connecteurs (par leurs coordonnées, leur appartenance à une équipotentielle, leur type,...), et il existe les fonctions nécessaires à ces recherches (voir fonction *trouve-l* dans le chapitre IV).

— 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 cela permet de piloter certains routeurs et outils d'assemblage. On retrouve pour ces niveaux les classiques : *Aluminium1, Aluminium2, Diffusion,...* Mais il est possible pour l'utilisateur d'employer ses propres notations, sachant qu'il existe des fonctions effectuant la traduction si deux types différents de notations sont utilisés (ces fonctions dépendent de paramètres à introduire dans les fichiers technologiques).

— La position

Chaque connecteur *normal* (c'est-à-dire ne provenant pas d'une instance) contient sa propre position relativement à la cellule dans laquelle il se trouve, sinon, s'il a été créé lors de l'instanciation d'une cellule, on peut retrouver sa position par calcul récursif sur l'arbre d'appel d'où il provient. De toute façon dans les deux 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.

— Le type

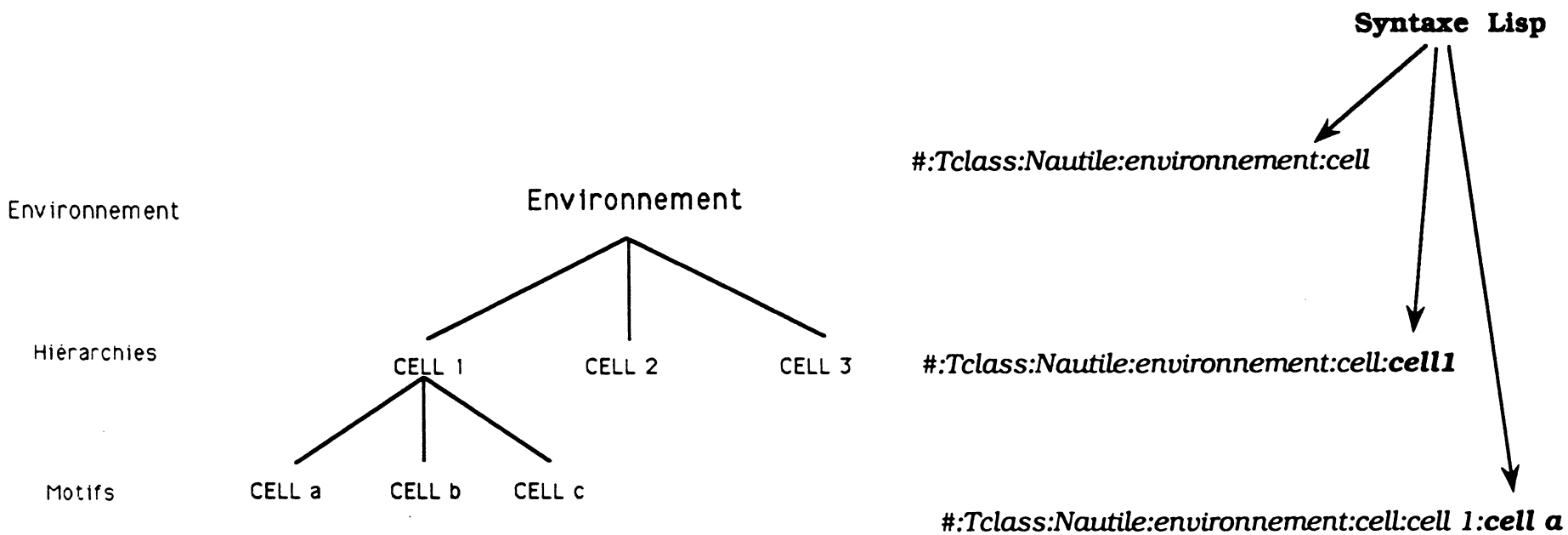


Figure III .7 : Nommage des cellules grâce au principe d'encapsulage Lisp

Chaque connecteur peut se voir associer un type particulier. On peut ainsi avoir un type *entrée*, *sortie*, *alimentation*,... Ces informations peuvent servir pour une vérification de cohérence lors des routages d'assemblage ou même diriger des assemblages élémentaires.

— 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.

— 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 plus loin). Un indicateur en position vrai ou faux déterminera si le connecteur doit être remonté ou non. Les méthodes de choix de la valeur de cet indicateur sont expliquées dans la suite (voir Figure III.8).

Dans la plupart des cas cet indicateur montrera si le connecteur a déjà été utilisé.

— L'équipotentielle

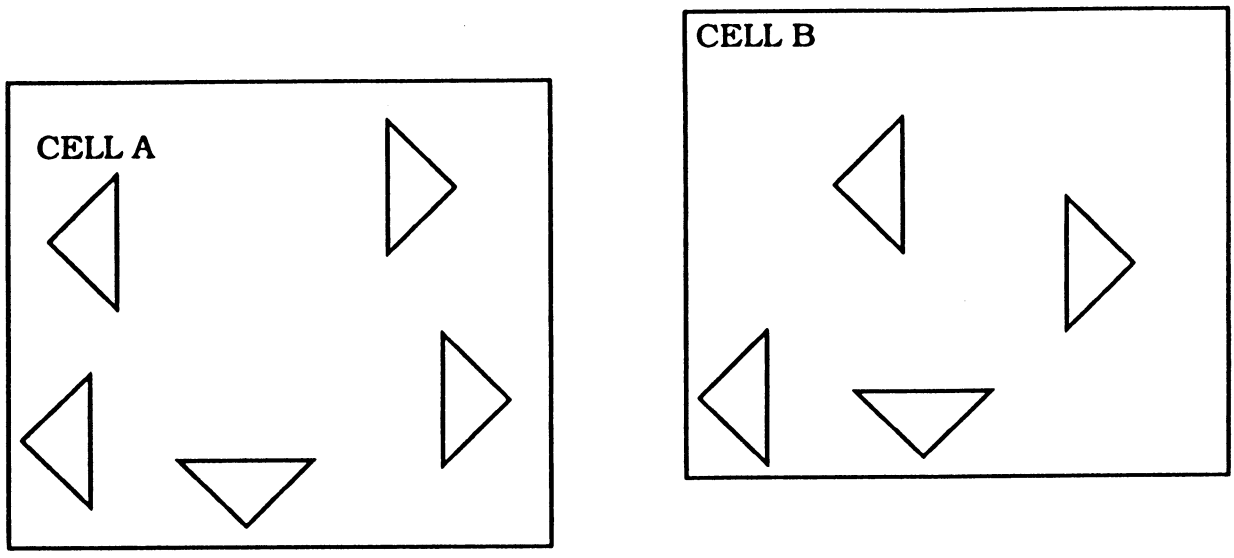
L'équipotentielle définit la notion de connexion multi-connecteurs. Il est possible d'assimiler un ensemble de connecteurs en contact électrique à une même famille. Un certain nombre d'informations doivent être accessibles ou calculables : à quelle équipotentielle appartient tel connecteur ou quels sont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

— La provenance éventuelle

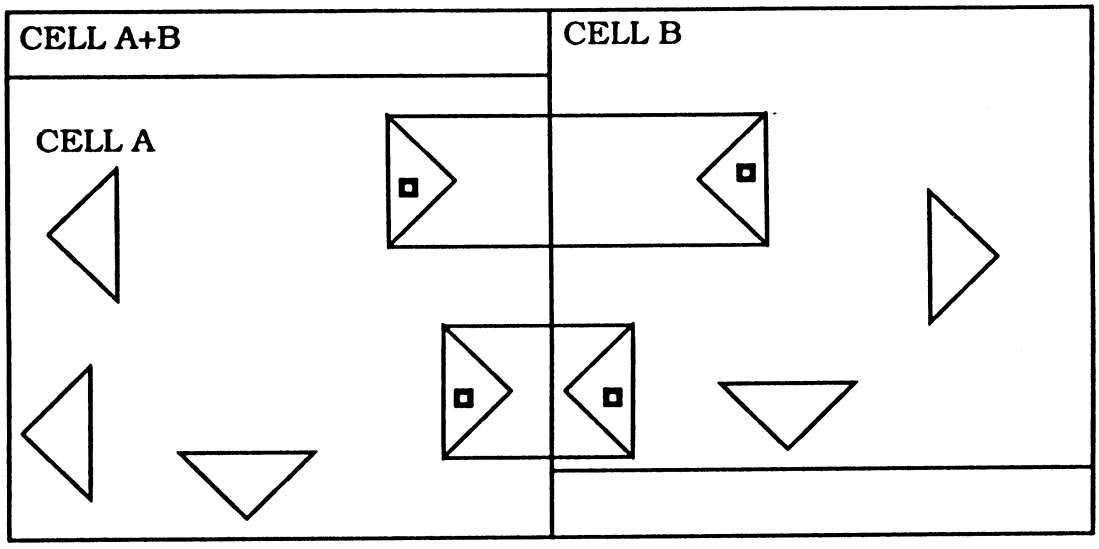
Un certain nombre de connecteurs ne sont pas créés dans la cellule mais proviennent de l'instanciation d'autres cellules (voir remontées de connecteurs IV.2.1). Pour offrir des facilités dans la gestion de la cohérence, il est nécessaire de savoir d'où proviennent les connecteurs (dans NAUTILE les connecteurs de type remonté ont une structure différente des connecteurs créés par l'utilisateur *normalement*).

III.3.3.1.2 Les instances

L'utilisation ou l'appel d'une cellule à l'intérieur d'une autre cellule est appelée instanciation. Ceci va se traduire dans la structure de données par un objet représentant l'appel ajouté au sein de la cellule appelante.



Assemblage de A et B - Marquage des connecteurs occupés



Nouvelle cellule C = A+B

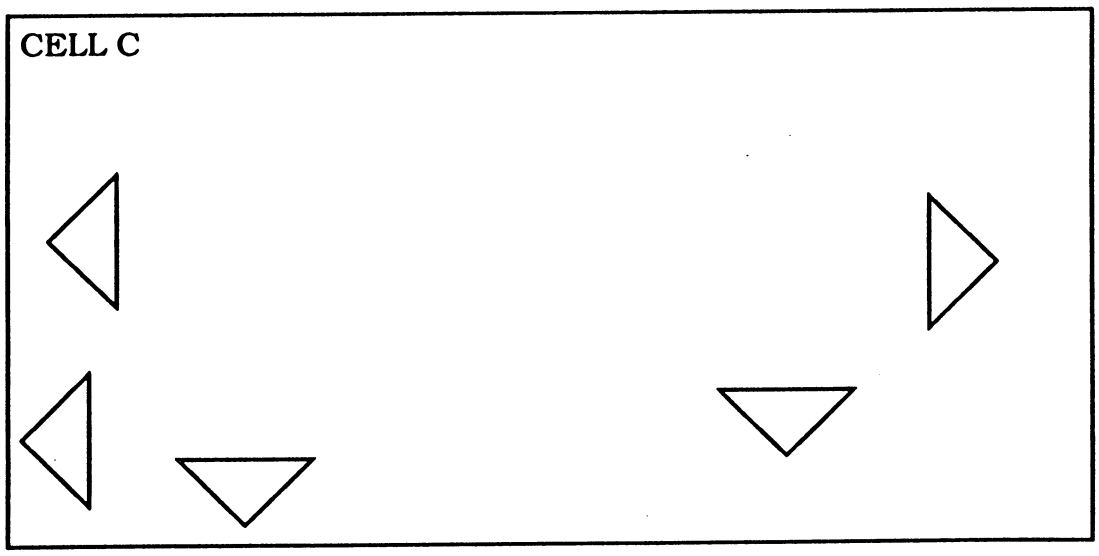


Figure III.8 : Remontée des connecteurs - Marquage des connecteurs

Les instances se retrouvent dans la vue de construction d'une cellule. Ces objets comportent les informations suivantes :

— Le nom de définition de la cellule appelée

Il s'agit du nom hiérarchique complet de la cellule.

— Le nom de l'instance

Il peut être généré par le système à partir du nom de définition ou bien être imposé par l'utilisateur. Ce nom sert à différencier les instances d'une même cellule.

— Des attributs d'appel (topologique, connectique,...)

* La liste de transformations à appliquer à l'instance

Il s'agit ici de répétitions, rotations, symétries,... que l'on applique à la cellule appelée.

* La position de l'instance

La position de l'instance sera donnée par rapport au repère de la cellule appelante.

III.3.3.1.3 Les états

Les objets *états* se situent à l'intérieur du squelette des cellules. Ils servent à la gestion de la cohérence et indiquent les dates de création, de dernière modification d'une cellule, ainsi que les vues présentes en mémoire et l'origine de la cellule.

III.3.3.1.4 Le corps d'une cellule

Les corps des cellules sont identiques pour les motifs et les cellules composées (voir III.2), et comportent une liste de paramètres ainsi qu'une liste de vues. Pour une cellule composée cette liste ne comportera qu'une seule vue, celle de construction, alors que pour un motif il y aura toutes les vues nécessaires aux traitements désirés (voir Figure III.9).

Exemple :

La figure III.9 montre une cellule C résultat de l'assemblage de deux cellule A et B. La cellule C ne pourra être traitée que suivant les vues I et III. En effet, le motif B n'ayant pas de vue II, il est impossible de trouver simplement la vue II de C.

III.3.3.2 La cellule

Une cellule contiendra tous les éléments cités ci-avant (III.3.3.1.1 à III.3.3.1.4) :

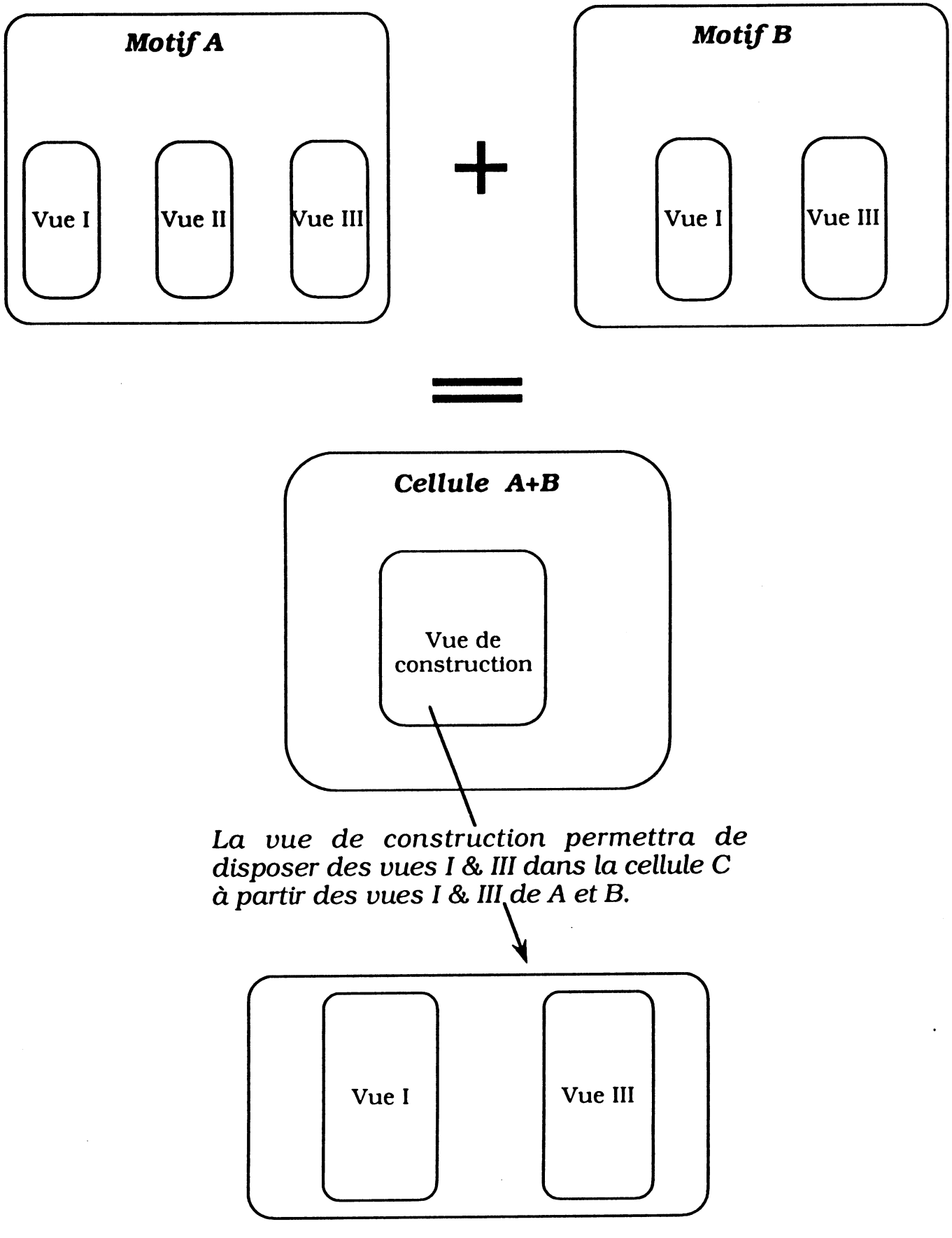


Figure III.9 : Héritage des vues

— Nom

Ce nom va correspondre au nom hiérarchique (cf. III.4.2) de la cellule, comprenant sous forme encapsulée (voir Figure III.7) tous les noms des pères de la cellule.

— Filles

On aura ici une liste des cellules filles de la cellule. C'est-à-dire qu'on aura une liste de noms hiérarchiques complets de cellules considérées comme étant définies sous cette cellule. Ces filles ne seront donc accessibles en instance que par leur mère, leurs *soeurs*, leurs descendants directs et les filles de leurs *soeurs* (voir figure III.10).

— Connecteurs

Il s'agit d'une liste des connecteurs de la cellule.

— Etat

L'état contiendra toute les informations relatives à l'état de la cellule dans le système : les dates de modification, les date d'accès, les adresses des fichiers à utiliser, éventuellement le nom d'un outil générateur,...

— Corps

Le corps de la cellule contient les listes des différentes vues. S'il s'agit d'un motif on pourra trouver différentes vues physique, sinon il s'agira d'une vue de construction.

— Boîte englobante

La boîte de la cellule est remise à jour à chaque nouvelle instance ou lorsque des modifications non prévues sont effectuées (suppression d'une instance, modification de la cellule appelée). Il existe une fonction recalculant la boîte englobante de la cellule. Dans le prototype NAUTILE (chapitre IV) seules les boites rectangulaires sont traitées, mais la version définitive devrait traiter des boîtes de forme polygonales.

— Icône

L'icône définit une représentation abrégée de la cellule pour ses différentes vues, sans devoir traiter celles-ci récursivement et complètement. On peut citer en exemple la représentation graphique topologique (le dessin des masques):
Plutôt que de dessiner entièrement toutes les cellules d'un circuit, on représente simplement certaines cellules (comportant des appels à des sous-cellules...) par une icône représentative. Ceci permet d'avoir une information graphique

utilisable et complète pour le concepteur, sans préjudice de temps passé à des explorations récursives inutiles pour chaque représentation.

III.4 Gestion de la cohérence

La cohérence d'un circuit peut être vue à deux niveaux :

1. un système permettant de gérer différentes vues d'un circuit doit être capable de vérifier la cohérence entre ces différentes représentations
2. le système doit maintenir la cohérence des appels par des méthodes chronologiques, c'est-à-dire vérifier que des modifications *a posteriori* ne viennent pas perturber le circuit.

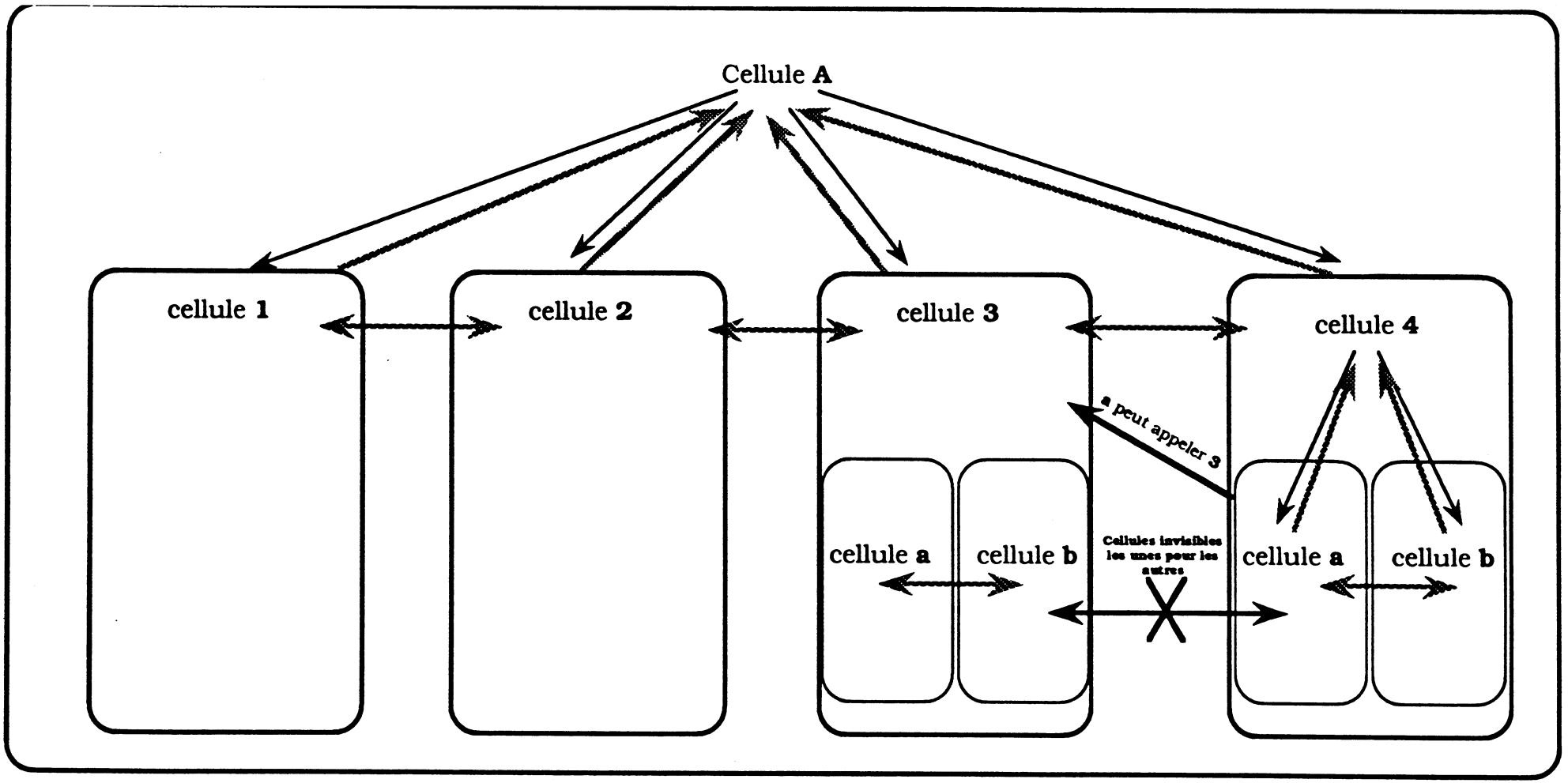
Exemple : Pour une cellule ayant deux vues différentes, il faut que ces vues soient équivalentes, c'est-à-dire que la réalisation physique de cette cellule ait les caractéristiques des deux vues. Au premier niveau il faut que la cohérence d'une vue hiérarchique électrique avec une vue hiérarchique topologique soit maintenue et au deuxième niveau en supposant que le concepteur crée une instance d'une cellule en respectant les règles d'assemblage, et qu'ensuite il modifie la cellule, il faudra à nouveau vérifier que les règles sont toujours respectées.

III.4.1 Cohérence par vérification des dates

La cohérence par vérification de date utilise les objets *états* décrits dans la structure de la cellule. Une méthode simple consiste à vérifier à la demande la cohérence d'une cellule et dans ce cas de comparer les dates d'appel de la cellule avec ses dates de dernière modification. Ces vérifications peuvent s'effectuer lors d'une sauvegarde, ou d'une instance de la *cellule mère*. On aurait aussi pu choisir de lier une cellule à toutes ses instanciations et lors d'une modification de la cellule mettre à jour toutes les cellules appelantes. Les liaisons structurelles nécessaires à cette opération n'ont pas été introduites dans la structure de données, néanmoins cette méthode reste applicable par calcul.

III.4.2 Cohérence par construction

Le système NAUTILE comprend des règles d'assemblage, permettant de concevoir un circuit juste par construction. Ces règles doivent être essentiellement d'ordre électrique et topologique. Ainsi avec un ensemble minimum de règles élémentaires, les erreurs de conception grossières (court-circuit masse et alimentation, gardes non respectées,...) sont évitées. Ces règles, bien que ne permettant pas d'assurer avec certitude que le circuit soit juste, permettent tout de même de faciliter considérablement le travail du concepteur. En effet, les erreurs mises en évidence par ce type de règles seraient complexes à corriger si elles n'étaient détectées qu'au cours d'un DRC final et pourraient nécessiter d'avoir à reprendre totalement l'assemblage depuis le début. En revanche, si le système signale ces erreurs au moment de l'assemblage incriminé,



Légende :

: Est fille de
: Est Mère de
: Est sœur de

Figure III.10 : Hiérarchie au sein des arbres de définition

l'erreur peut être corrigée immédiatement.

Les règles d'assemblage sont appliquées lors de chaque utilisation d'un outil d'assemblage ou lors d'une instanciation. Ces règles permettent de paramétrer la technologie pour les outils d'assemblage.

Les règles d'assemblage présentes dans le système sont essentiellement des règles :

— **Topologiques :**

Lors d'un assemblage automatique, l'outil d'assemblage doit se référer à des règles pour placer la ou les cellules. Lors d'une instanciation, le système préviendra l'utilisateur en cas de placement illicite. Ces règles topologiques couvrent des vérifications de non recouvrement de cellules (sauf cas d'ombre ou d'élément partagé), de distances minimales ...

— **Electriques :**

Les règles électriques servent à vérifier de façon sommaire (une vérification totale prendrait un temps rédhibitoire), si le circuit est correct et viable électriquement. Ceci peut inclure des vérifications de cohérence élémentaires (ne pas relier une masse à une alimentation, vérifier que tous les transistors ont une possibilité de connexion...)

III.5 Conclusion

Le système NAUTILE à l'instar de SKILL, DPL ou GDT (voir chapitre II) offre un nouvel environnement pour la description de modules générateurs de circuits ; il peut ainsi être utilisé pour créer un compilateur de silicium. Il permet de gérer de façon cohérente la conception d'un circuit, gardant à jour des représentations aussi bien topologiques qu'électriques ou logiques. Il utilise pour cela un langage unifié de description.

IV. NAUTILE : Les aspects pratiques - Réalisation d'un prototype

Une première version du système NAUTILE à déjà été réalisée sous la forme d'un prototype. Ce prototype a permis au compilateur de silicium SYCO de réaliser la partie contrôle du 6502 [GeJ87] [Ger87].

Nous allons décrire ici cette première version et ses caractéristiques techniques.

IV.1 Apport de Ceyx - Le_Lisp (langage orienté objet)

Le choix du langage Le_Lisp [CDD86] et plus particulièrement Ceyx [Hul84] (qui offre des possibilités de programmation orientée objet) pour réaliser un prototype du système NAUTILE a permis d'accélérer la mise en oeuvre.

Qu'apporte Ceyx au langage Le_Lisp ?

Ceyx constitue essentiellement un apport de nouvelles structures au langage Le_Lisp. Ceci permet de manipuler des structures équivalentes aux *enregistrements* (ou records) de Pascal. A ces structures sont associés des espaces sémantiques organisés hiérarchiquement. Il permet aussi d'ajouter des propriétés sémantiques à ces structures, c'est-à-dire qu'on peut définir un certain nombre de fonctions exécutant différentes opérations suivant l'objet auquel elles s'appliquent, cette possibilité pouvant aller jusqu'à établir des arbres de fonctions correspondant à des structures hiérarchiques. Des applications classiques se retrouvent dans les fonctions du plan (points, rectangles, intersections de rectangles,...).

On peut recenser un certain nombre d'avantages offerts par Le_Lisp :

IV.1.1 L'interactivité d'un système interprété

Pour mettre au point un logiciel il est très intéressant d'utiliser un langage interprété. En effet dans un langage interprété, il n'y a pas de compilation ce qui permet de gagner du temps et de voir immédiatement les conséquences d'une modification d'une partie de programme.

IV.1.2 Facilités de développement

Le langage Le_Lisp offre des facilités diverses, entre autres un outil de deboggage particulièrement évolué permettant de gérer la pile des appels de fonctions, de visualiser les valeurs des variables, d'avoir des sémantiques particulières d'affichage des structures de données, de modifier ces structures, ...

IV.1.3 Structure

Les possibilités de structuration offertes par Ceyx ont permis de traduire directement la structure nécessaire à NAUTILE en une structure informatique utilisable et proche du circuit à concevoir. Ceyx permet d'utiliser dans un langage de type lisp (Le_Lisp) les structures classiques de Pascal et C.

IV.1.4 Héritage sémantique

Les propriétés d'héritage sémantique offertes par le langage Ceyx sont riches en possibilités. En effet, elles permettent pour les cellules ayant des aspects multiples, de décrire des sémantiques pour chacune des vues électrique, logique, topologique,... . Suivant le mode de fonctionnement ou suivant le type de l'objet à traiter, lors de l'envoi du message aux objets, la sémantique appropriée sera appliquée.

Dans le futur il est possible d'envisager, lorsque la phase de prototypage sera achevée et que le système devra offrir de meilleures performances en rapidité, d'utiliser un autre langage tel que C ou mieux encore, d'utiliser une machine Lisp. Ceci devrait permettre de conserver les avantages d'un système interactif tout en ayant une vitesse d'exécution compétitive avec un langage compilé.

Dans la suite nous allons décrire de façon pratique le prototype réalisé. On sera amené à décrire un certain nombre de fonctions, parmi lesquelles il faudra distinguer deux types de fonctions décrites :

— Les fonctions *utilisateur*

— Les fonctions *système*

Les secondes se différencient des premières par le symbole ":" en tête du nom (exemple : la fonction utilisateur *acces* et la fonction système *:acces*, l'une faisant appel à l'autre)

IV.2 Les primitives de base

Les fonctions décrites ici sont les fonctions de base du système NAUTILE. Elles constituent le noyau même du système.

Le système NAUTILE fonctionne comme un éditeur de structure de données. Il permet de fabriquer des cellules :

— création de cellules,

— modification de cellules,

— accès à une cellule.

Il permet aussi de gérer l'environnement de travail :

— gestion des bibliothèques,

— gestion de l'environnement de l'éditeur,

— organisation de la mémoire et des différents fichiers.

IV.2.1 Les primitives de création d'objet NAUTILE

On dispose des fonctions permettant de créer les objets de base de la structure de données du système NAUTILE :
:mkcell :mkconnect :mkcorps :mketat :mklapp :mkmext :mkmint :mkmotif :mkorig :mkrect :mkrouteur :mkterm
Les fonctions citées ici sont les primitives de création des objets du système. En règle générale elles ne seront pas utilisées par un concepteur mais uniquement dans le cadre de modification des fonctions système.

IV.2.2 Primitives systèmes

Les fonctions décrites ici ne sont en principe pas destinées à l'utilisateur, mais permettent de mieux comprendre le fonctionnement des autres fonctions.

(:add-hierar <N-cellule>)
(:ret-hierar <N-cellule>)

La fonction *:add-hierar* permet d'ajouter dans la hiérarchie de définition de cellule de nouveaux éléments, elle effectue la distinction entre une nouvelle hiérarchie créée et un nouveau maillon dans une hiérarchie existante (mise à jour de la liste des hiérarchies ou des filles de la cellule courante). Cette fonction ne manipule que des noms de cellule, et non pas les objets eux-même (exemple : pour ajouter une nouvelle hiérarchie, on ajoute dans la liste des hiérarchies le symbole *nom* de la nouvelle cellule).

La fonction *:ret-hierar* permet de supprimer une cellule existante, qu'elle soit au coeur d'une hiérarchie ou qu'elle soit une hiérarchie elle-même. Cette fonction supprime en même temps les cellules filles et tient à jour la liste des hiérarchies.

Le principe de gestion des cellules utilise les possibilités de lisp dans la gestion des noms et des symboles, et l' *oblist*. Pour cette gestion des noms on utilise deux fonctions de base :

(:nomact <cellule>)
(:recherche <cellule>)

Comportement des fonctions *:nomact* et *:recherche* :

La fonction *:nomact* renvoie un nouveau symbole composé du nom de la cellule courante suivi du nom donné.

Exemple : la cellule courante a pour nom réel #:environnement:cell-c
(:nomact 'nouveau) renverra #:environnement:cell-c:nouveau

De la même façon la fonction *:recherche* permet à partir d'un nom utilisateur (non composé) de retrouver la cellule désirée dans la hiérarchie des cellules (voir exemple sur la Figure IV.1) :

Algorithme :

1) Recherche :

(Recherche nom) = (chercher nom nom-cellule-courante)

; Pour rechercher le nom donné il faut appeler la fonction chercher
;avec les arguments nom et nom-cellule-courante

2) Chercher :

(Chercher nom package) = Si la variable package:nom n'existe pas
Alors on applique Chercher à nom et [package -1]
Sinon package:nom

Note : [package -1] consiste a retirer la dernière partie du *package*

Exemple :

si package = #:environnement:cell1:cell2 alors
[package -1] = #:environnement:cell1

IV.2.3 Définition de cellules

Fonctions servant à la définition de nouvelles cellules

(defcell <nom de cellule>)
(defmotif <nom de motif>)
(:defcell <nom de cellule> <liste de paramètres> <liste de connecteurs> <etat de la

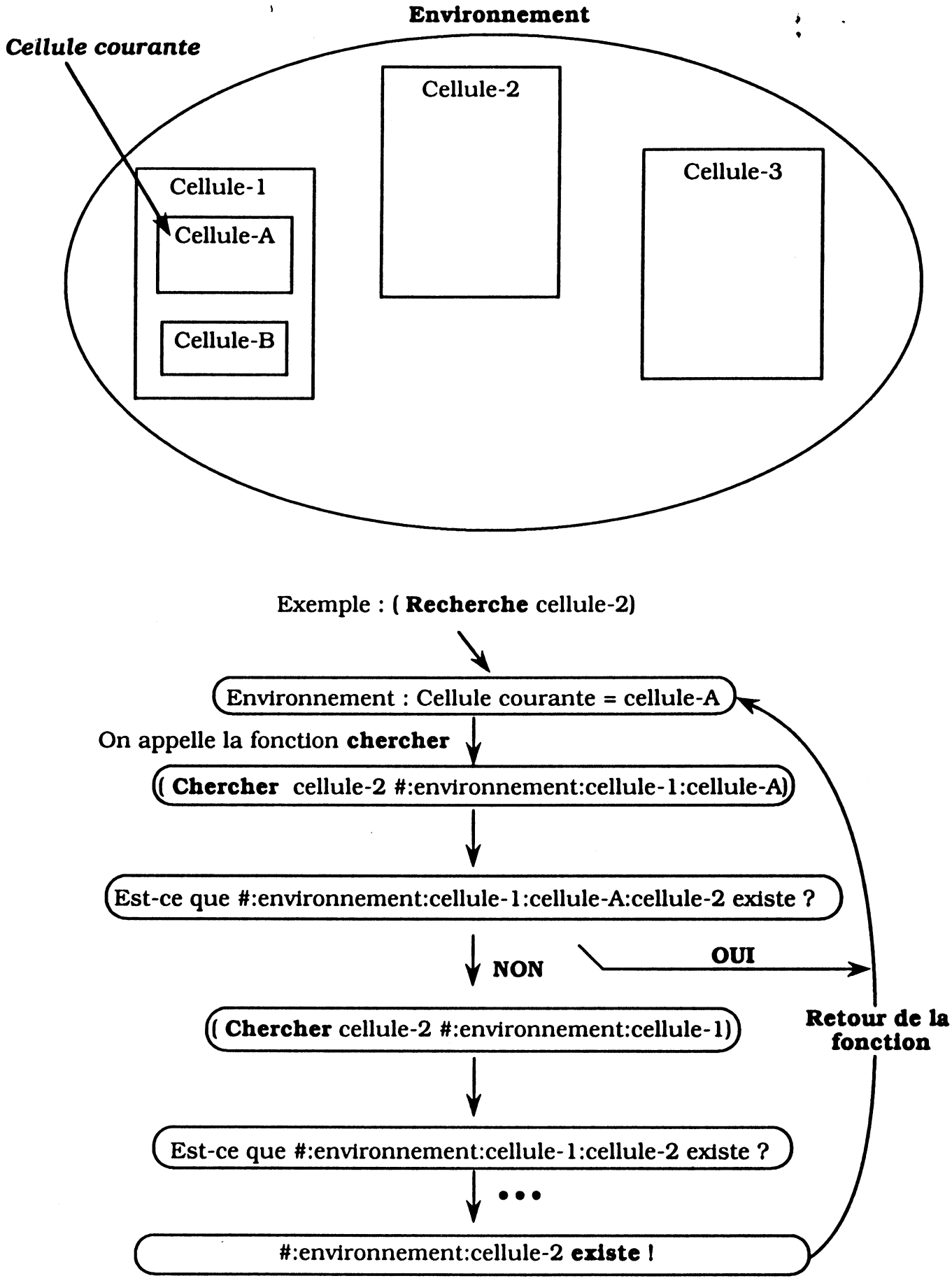


Figure IV.1. : Recherche d'une cellule dans l'environnement

cellule> <corps> <boîte> <indic de cellule ou motif>)

Les deux fonctions *defcell* et *defmotif* sont utilisées pour créer de nouvelles cellules. Elles permettent de définir dans la structure de données au niveau de la cellule courante une nouvelle cellule de nom <nom de cellule>. Ces deux fonctions font appel à la fonction système *:defcell.*

Une gestion des noms de cellules est associée à chaque nouvelle définition de cellule. Chaque nouvelle cellule créée va avoir un nom complet composé de son chemin dans la hiérarchie ainsi que de son nom local.

Comportement des fonctions *defcell* et *defmotif* :

Pour la fonction *defcell* :
Appel de *:defcell* avec un nouvel *état* (on utilise la fonction *:mketat* et l'indicateur de cellule).

Pour la fonction *defmotif* :
Appel de *:defcell* avec un corps dans le mode courant (topologique ou autre), un nouvel état et un indicateur de motif.

Application de la fonction *:defcell* (fonction système dépendant de la cellule courante, voir la fonction *acces*) :

On recherche le nouveau nom en tenant compte de la hiérarchie (:nomact)
On efface éventuellement de la hiérarchie des cellules, une cellule portant le même nom (:ret-hierar), on ajoute dans la hiérarchie le nom de la cellule (:add-hierar)
On définit une nouvelle variable portant le nom obtenu par *:nomact* et contenant la nouvelle cellule créée à partir des arguments (on applique :mkmotif ou :mkcellule)

IV.2.4 Ouverture/fermeture de cellules

Une fois une nouvelle cellule définie, on peut y ajouter des éléments. Pour accéder à cette cellule on utilise une fonction *acces* qui permet de spécifier que toutes les opérations suivantes jusqu'au prochain *fin-acces* seront effectuées sur cette cellule. Cette cellule est appelée cellule courante et toute nouvelle cellule définie alors, sera dite *fille* de la cellule courante.

Fonctions permettant d'accéder à une cellule :

(acces <nom de cellule>)

(accesa <nom hiérarchique explicite de cellule>)
(fin-acces {<nom de cellule>})

La fonction fin-acces peut être utilisée en correspondance à un *acces* ou pour préciser le nom de la cellule que l'on ferme, et fermer ainsi toutes les cellules ouvertes depuis.

Exemple :

```
(defcell 'a)
        (acces 'a)
        (defcell 'b)
        |       (acces 'b)
        |       (defmotif 'c)
        |       |       (acces 'c)
        |       |       |...
        |       |       (fin-acces)
        |       |...
        (fin-acces 'a)
```

Comportement des fonctions *acces* et *fin-acces* :

1) acces <Nom de cellule>
- On recherche le nom réel de la cellule à laquelle on veut accéder (fonction *:cherche*). On vérifie en même temps que la cellule existe bien.

- On ajoute, dans la pile des accès, la cellule trouvée et on la place dans le champ <Cell-courante> de la structure générale de donnée (Voir dans la structure de données la classe *STRUCT* Annexe II)

- On renvoie le nom complet de la cellule accédée

2) fin-acces {<nom de cellule>}

- Si il n'y a pas d'argument : dépiler la pile des accès et mettre à jour la cellule courante avec la nouvelle tête de pile

Sinon :

dépiler la pile jusqu'à trouver la cellule de nom donné et mettre à jour la cellule courante.

IV.2.5 Sauvegarde - Restauration

NAUTILE offre plusieurs possibilités de sauvegarde. L'un des principes majeurs de NAUTILE réside dans son aptitude à gérer des cellules décrites dans d'autres systèmes. On peut ainsi travailler sur une cellule faisant appel à des cellules *lubrick*, *rnl*,... Le corps électrique sera en *rnl*, le corps topologique en *lucie*,... On peut ainsi travailler de deux façons différentes : utiliser des fichiers contenant la description de la cellule dans son langage d'origine, ou bien traduire la cellule en NAUTILE. De même la réciproque est possible, lors de la sauvegarde d'une cellule, on choisit le langage dans lequel on va décrire la cellule.

On dispose pour cela de deux fonctions générales : *charge* et *genere*

(charge <systeme> <fichier> <cellule>) ; chargement d'une cellule
(char-lib <systeme> <fichier> <cellule>) ; chargement d'une cellule de bibliothèque
(genere <systeme> <fichier> <cellule>) ; génération d'une cellule
(gen-lib <systeme> <fichier> <cellule>) ; génération d'une cellule de bibliothèque

On utilisera des bibliothèques de fichiers ordonnés de façon hiérarchique qu'on appelle *directory* (terme provenant du système d'exploitation UNIX) pour les sauvegardes.

On notera qu'il existe deux types de fonctions de sauvegarde-restauration, les fonctions *normales* générant la cellule dans un fichier dans le directory courant, et les fonctions *spéciales* générant la cellule dans un directory de bibliothèque correspondant au système choisi (LUBRICK, LUCIE, RNL, ...).

Fonctions NAUTILE permettant la gestion des bibliothèques :
cd, existe, ll, pwd, param.

Ces fonctions permettent d'initialiser et de visualiser les valeurs par défaut des adresses des bibliothèques. La fonction *cd* permet de changer le directory de sauvegarde par défaut des cellules du système cité.
Exemple :
(cd LUBRICK /users/naut/dupont/lubr) permettra au prochaines sauvegardes de cellules sous leur forme LUBRICK de s'effectuer dans le répertoire */users/naut/dupont/lubr*.

La fonction *existe* permet de tester l'existence d'un fichier d'un système particulier (les noms donnés aux fichiers sont étendus automatiquement d'un préfixe et d'un suffixe dépendant du système). La fonction *ll* permet de voir la liste des fichiers contenant des cellules décrites dans un système particulier, cette fonction recherche les fichiers dans le directory courant et dans les directory's de bibliothèques.
Exemple :

(ll LUBRICK) sous NAUTILE correspondra à l'exécution sous UNIX de : *$ ls -CF /users/naut/dupont/lubr.*
La fonction *pwd* permet de rappeler le directory choisi par défaut pour un système. Enfin la fonction *param* permet de régler les paramètres des fonctions *génère* et *charge* suivant les différents systèmes (taille des noms générés,...).

Exemple

```
(de assemb (cell mot x y bibli directory système1 cell1-bibli fich1
                système2 cell2-local fich2)

 ; Définition d'une nouvelle fonction

 (defcell 'ASSEMBL) ; Définition d'une nouvelle cellule
        (acces 'ASSEMBL)   ; Accès à la nouvelle cellule.
        | ...
        (foo x y)                    ; actions diverses sur la cellule.
        | ...
        (defmotif mot)     ; Définition d'un sous-motif

                (acces mot)          ; Accès au motif

                (charge système2 cell2-local fich2)

                ; chargement d'une vue du motif

                (cd système1 bibli)
                (char-lib système1 cell1-bibli fich1)

                ; chargement d'une seconde vue depuis la
                ; bibliothèque.

                (fin-acces)

        (fin-acces))
```

IV.2.6 Gestion des connecteurs

Les connecteurs ont posé de nombreux problèmes dans la version de base de NAUTILE. En effet il fallait pouvoir *remonter* les connecteurs d'une cellule appelée dans la cellule appelante. Deux solutions ont été envisagées :

— avoir une structure de type pointeur pour les connecteurs originaires d'une autre cellule, et ne conserver que leur nom, leur provenance et éventuellement des noms d'équipotentielles ou types différents, et retrouver par calcul toutes les autres informations suivant les besoins,

— dupliquer toutes ces informations dans les connecteurs.

IV.2.6.1 Gestion par table Hashcode et recherche hiérarchique

Dans un premier temps la première solution fut implantée avec deux types de structures différents, l'un pour les connecteurs remontés et l'autre pour les connecteurs *originaux*. Cette solution utilisait une table contenant *in extenso* les 200 derniers connecteurs sur lesquels on avait demandé des informations, ceci permettant d'éviter des calculs répétés et des déplacements trop fréquents dans la structure. Cette table était gérée par un système classique de table de *hashcode*. Cette méthode offrait deux avantages immédiats:

— Gestion dynamique des connecteurs

— Gain de place mémoire

Cette gestion était totalement transparente pour le système ou l'utilisateur, car utilisait les principes d'envoi de sémantique.

exemple : pour connaitre la position d'un connecteur **C** on envoie le message Position au connecteur, si **C** est normal le message provoque l'envoi de la position de C présente dans sa structure, sinon le système regarde dans la table si le connecteur a déjà été calculé; dans ce cas il renvoie sa position présente dans une structure similaire à un connecteur normal dans la table. Enfin si le connecteur n'a pas été calculé, il regarde de quelle instance il provient et calcule sa position; il est d'ailleurs possible que pour calculer sa position, il ait besoin d'envoyer un message Position à d'autres connecteurs.

Cependant le temps d'exécution était considérablement augmenté en raison des parcours hiérarchiques de la structure.

IV.2.6.2 Gestion par remontée totale

La deuxième méthode fut beaucoup plus simple d'application et consista à garder les connecteurs avec toutes leurs informations dupliquées plus l'information de la cellule de provenance. Ceci impliquait une place mémoire supérieure mais permettait de garder, moyennant des procédures de mise à jour, une structure dynamique.

IV.2.6.3 Ajout de connecteurs

Nous avons ici les différentes fonctions permettant d'ajouter des connecteurs dans une cellule.

(addconn <cellule> <sens> <x> <y> <lar> <niv> <genre> <equi> {<nom>})

La fonction addconn permet d'ajouter dans une cellule un connecteur en précisant tous ses paramètres (équipotentielle, position, largeur, niveau, sens) et éventuellement de donner un nom au connecteur. De façon générale ce nom ne sera pas donné par l'utilisateur mais généré de façon automatique. On peut préciser que le système ne dispose que du nom pour gérer les connecteurs, il est donc essentiel pour lui qu'à l'intérieur d'une même cellule tous les connecteurs aient des noms différents. Par contre l'utilisateur dispose des position, équipotentielle, niveau et type pour reconnaître les connecteurs.

Exemple :

```
(defcell 'ASSEMB) ; Définition d'une nouvelle cellule
        (acces 'ASSEMB)   ; Acces à la cellule
        (addconn 'ASSEMB 'Nord 5 3 2 'alu1 'entree 'af-12 ())

        ; ajout d'un connecteur sur la couche alu1 de type entree dans
        ; la direction  Nord, à relier à l'équipotentielle af-12 en
        ; x=5 y=3 et de largeur 2

        (fin-acces)
```

IV.2.6.4 Consultation des connecteurs

On dispose de fonctions permettant de consulter les connecteurs présents dans une cellule, d'extraire des connecteurs dans une direction,...

(C? <liste de directions>)
(conn-dir <dir> <cellule> {<occupation>})
(depconn-dir <dir> <cellule> <position> {<occupation>})
(conn-dirp <dir> <cellule> {<occupation>})

C? donnera la liste des objets connecteurs de la cellule courante dans une liste de directions données.
conn-dir renvoit les objets connecteurs, et les marquent éventuellement s'ils ont déjà été utilisés. Cette possibilité prend toute son importance dans les fonctions *depconn-dir* et *conn-dirp* qui permettent de récupérer une copie des objets connecteurs translatés dans une direction. Ces objets peuvent servir aux routeurs, répondent aux mêmes sémantiques que les objets connecteurs mais ne nécessitent, aucun calcul lors des accès aux champs des connecteurs. Ces objets doivent être libérés après utilisation pour pouvoir réutiliser la place mémoire inutilement occupée.

IV.2.6.5 Recherches suivant un champ

Au cours des opérations sur les connecteurs, il est souvent nécessaire de repérer des connecteurs par les valeurs des champs des objets, on a pour cela la fonction *trouve-l* :

(trouve-l <valeur> <liste d'objet> {<champ>})

trouve-l va chercher dans la liste d'objets les objets dont le champ *champ* a pour valeur *valeur*. Par défaut le champ recherché sera le champ *Nom.*

Algorithme :

[1] trouve-l Nom liste sémantique (à laquelle doivent savoir répondre les éléments de la liste)

élément <- tête de liste
résultat <- ()

Tant que la liste n'est pas vide faire :
| Si : Nom = sémantique envoyée à élément
| |
| | Alors : on ajoute élément à la liste résultat
| |
| Fin de Si
|
| Avancer dans la liste
|
Fin de Tant que

Exemple : On a dans la variable *lsc* une liste de connecteurs et l'on veut retrouver les connecteurs de type *Ent* :

(trouve-l 'Ent lsc 'Type)

IV.2.6.6 Remontée de connecteurs

Le problème de remontée des connecteurs est induit par le problème de l'instanciation, de la gestion de la cohérence et d'une gestion dynamique de la structure de données.

Nous abordons ici un des problèmes principaux de NAUTILE, la gestion des connecteurs et leur remontée.

Pour chaque instanciation d'une cellule A dans une cellule B, il faut décider quels connecteurs de A devront être présents dans B. Si par la suite il faut détruire ou déplacer A, il faudra déplacer ou détruire ces connecteurs (voir Figure IV.2).

Il a fallu chercher une structure allégée pour les connecteurs remontés pour deux raisons:

— pour garder une gestion dynamique des connecteurs, la plupart des informations doivent être recalculées en fonction des autres connecteurs ou cellules ; une modification de ces derniers sera répercutée immédiatement pour toute demande d'information

— Afin d'utiliser un minimum d'espace mémoire.

(:monteg <Lconnect> <Position> <N-inst> <Cellule> <Nombre> {<Liste de changement de nom>})

La fonction *:monteg* est utilisée par les fonctions d'instanciation ou d'assemblage pour *remonter* les connecteurs.

Algorithme :

1) (monteg Lconnect Position N-inst Cellule N)

Pour chaque *connecteur* dans la liste de connecteurs Lconnect faire :
Si le connecteur n'est pas *occupé*

Alors : Ajouter N fois dans la cellule courante un nouveau connecteur avec un nouveau nom. On ne garde dans ce nouveau connecteur que les informations suivantes:

— Le nom de l'instance d'où il provient

— Le nom du connecteur d'où il provient

(N correspond au numéro de répétition : si une instance correspond à une cellule qu'on répète N fois on devra remonter N fois chaque connecteur).

Toutes les autres informations sur le connecteur seront recalculées à la demande. Cependant il est possible de forcer une valeur différente de celle que l'on obtiendrait par le calcul.

Exemple :

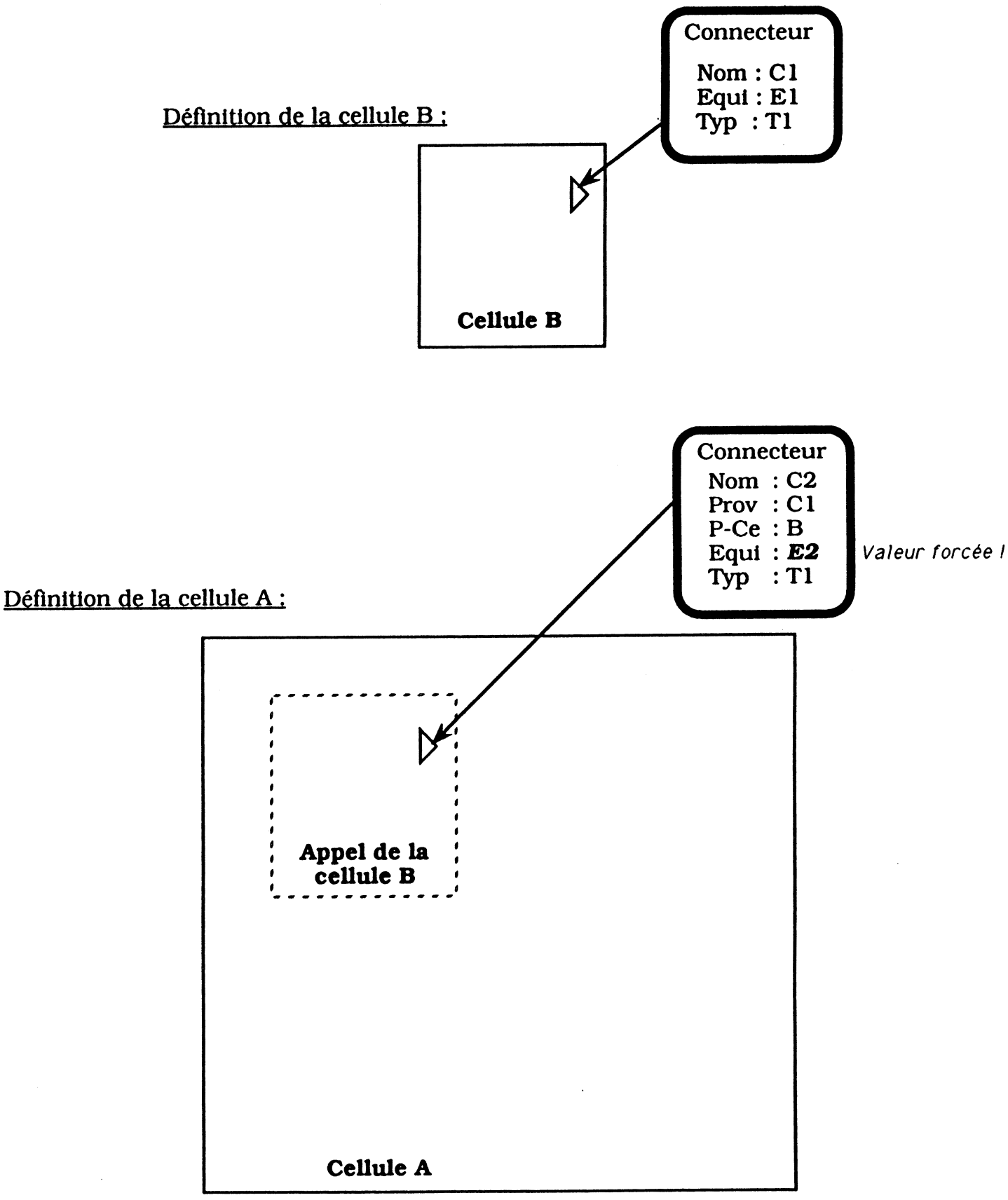


Figure IV.2. : Exemple de remontée de connecteur

Soit le motif B qui possède le connecteur C1. On instancie B dans la cellule A. Le connecteur C1 est remonté dans A (et s'appelle C2 dans A), mais la valeur donnée pour son équipotentielle doit être différente de celle du connecteur C1 de B. Il est possible de forcer la valeur du paramètre *équipotentielle* (voir Figure IV.2).

Le principe de remontée des connecteurs est le suivant :

Lors de la demande d'une information non présente dans la structure d'un connecteur remonté, on va consulter une table de connecteurs contenant les derniers connecteurs calculés (les 200 derniers). Si ce connecteur est présent dans la table on donne l'information, sinon on va devoir retrouver par calcul le connecteur. Pour cela, on utilise la fonction *calcul-conn* qui recherche la provenance du connecteur (instance et connecteur d'origine) et va calculer entièrement le connecteur. Cette fonction est récursive et peut donc s'appeler à nouveau si le connecteur d'origine est lui-même un connecteur remonté non présent dans la table des connecteurs. Cette table des connecteurs est accédée par une méthode de *hashcodage* (voir Figure IV.3 et IV.4).

La figure IV.3 explique le fonctionnement de la table des connecteurs : lors d'une demande d'information sur un connecteur deux conduites sont possibles :

— **Le connecteur est** *normal* : Dans ce cas les informations sont accessibles immédiatement dans la structure de données.

— **Le connecteur est** *remonté* : Il faut consulter la table des connecteurs : a) le connecteur est présent, les informations sont disponibles et à jour ; b) le connecteur est absent et il faut le calculer et le ranger dans la table des connecteurs.

Il est a noter qu'il est nécessaire de vider la table de Hashcodage après toute modification *majeure* de cellule (par exemple : supression de cellule , déplacement,...)

Exemple :

La figure IV.4 donne un exemple d'utillisation de cette table de connecteur :
l'utilisateur demande une information sur le connecteur C6318, le système code C6318 et obtient la valeur 3, il va regarder dans la troisième entrée de la table des connecteurs si C6318 y est présent. C6318 n'ayant pas été calculé sa version élémentaire est utilisée où l'on voit qu'il provient de C215. C215 est un connecteur remonté mais présent dans la table des connecteurs. Un simple calcul (translations de vecteurs pour les positions : position de C215 composé avec le repère de C6318) permet de retrouver les informations et de mettre à jour la table des connecteurs.

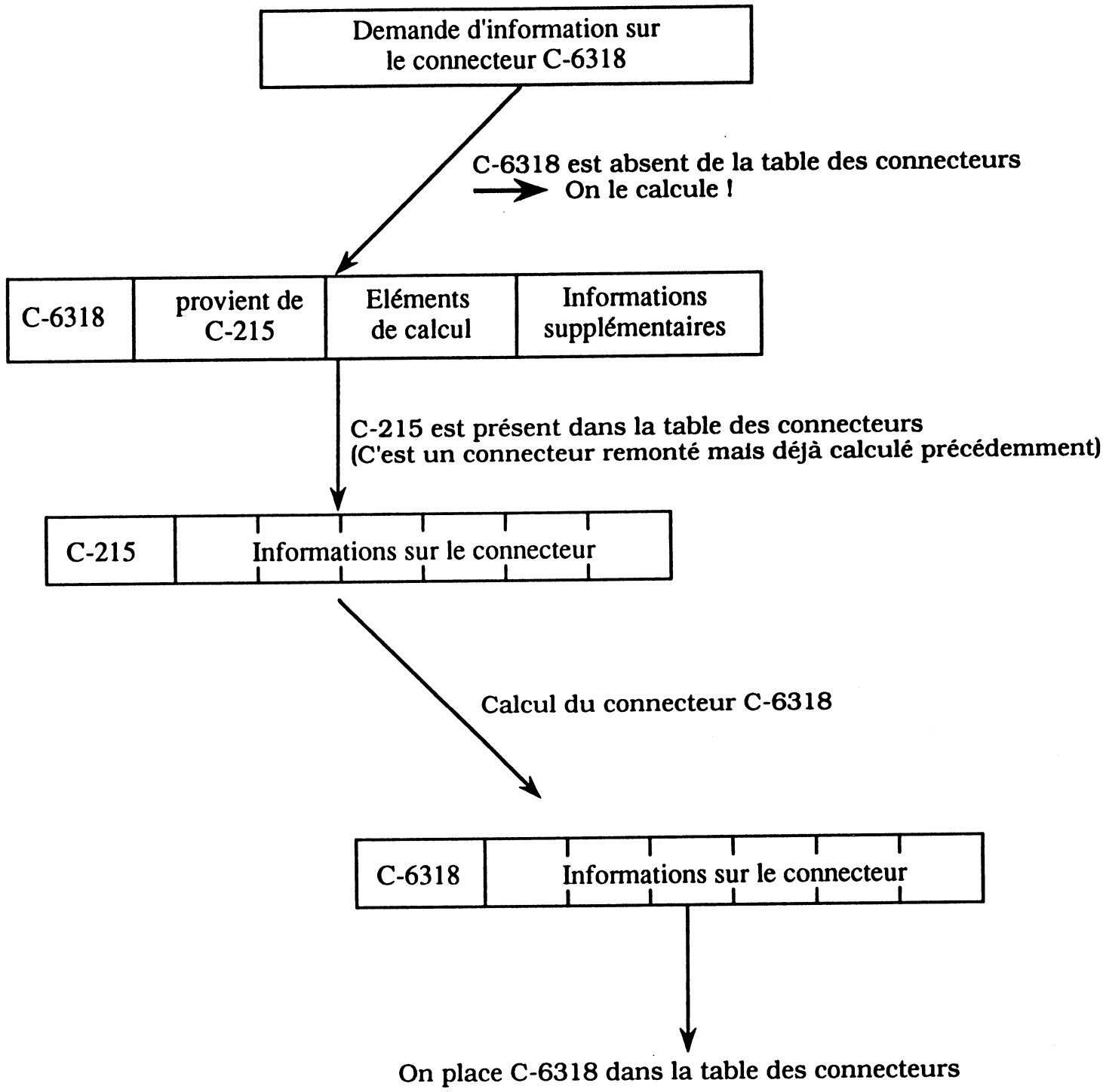


Figure IV.3 : Mécanisme de gestion des connecteurs : "Exemple de calcul de connecteur"

IV.2.6.7 Marquage des connecteurs

A différents moments il est nécessaire de pouvoir décider qu'un certain nombre de connecteurs dans une direction et éventuellement provenant d'une instance particulière, ne doivent plus être remontés dans la suite des opérations. Pour cela on dispose de la fonction *marque*. En combinant les fonctions *:marque* et *trouve-l* on peut marquer les connecteurs en utilisant différents types de critères (position, type, équipotentielle,...).

(:marque <direction> {<nom d'instance>})

Cette fonction agit sur les connecteurs présents dans la cellule courante. Les principaux outils utilisant cette fonction sont les outils d'assemblage (un connecteur ayant servi à un assemblage sera en général marqué car il ne servira plus à d'autres assemblages)

IV.2.6.8 Connexions spécifiées réalisées par un rectangle

Les fonctions décrites ici font partie des primitives élémentaires d'assemblage. Elles permettent d'assembler des connecteurs situés les uns en face des autres par de simples rectangles, et tiennent à jour les vues non topologiques.

(connect <N-conn1> <N-conn2> <N-motif> {<occupe?>})
(connect-E <N-motif> <Equipot>)

La primitive de base est *:connect* qui assemble deux objets connecteurs et on a les deux fonctions utilisateur *connect* et *connect-E* qui pour l'une assemble deux connecteurs en utilisant un motif, et pour l'autre assemble tous les connecteurs de même nom d'équipotentielle (attention les connecteurs sont supposés être en ligne).

IV.2.7 Instanciation de cellules

IV.2.7.1 Sémantique générale

Description des primitives d'instanciation de nouvelles cellules :

(instance <cellule> <N-instance> <Position> <Lparam>)
et les fonctions associées :
instancel instance-C instancel-C
(:addinst <cellule> <instance>)

Les fonctions *instance* diffèrent par les objets qu'elles retournent et par leurs actions sur les connecteurs. Les fonctions *instancel* et *instancel-C* retournent uniquement le nom de la nouvelle instance créée alors que *instance* et *instance-C* renvoient le nouvel

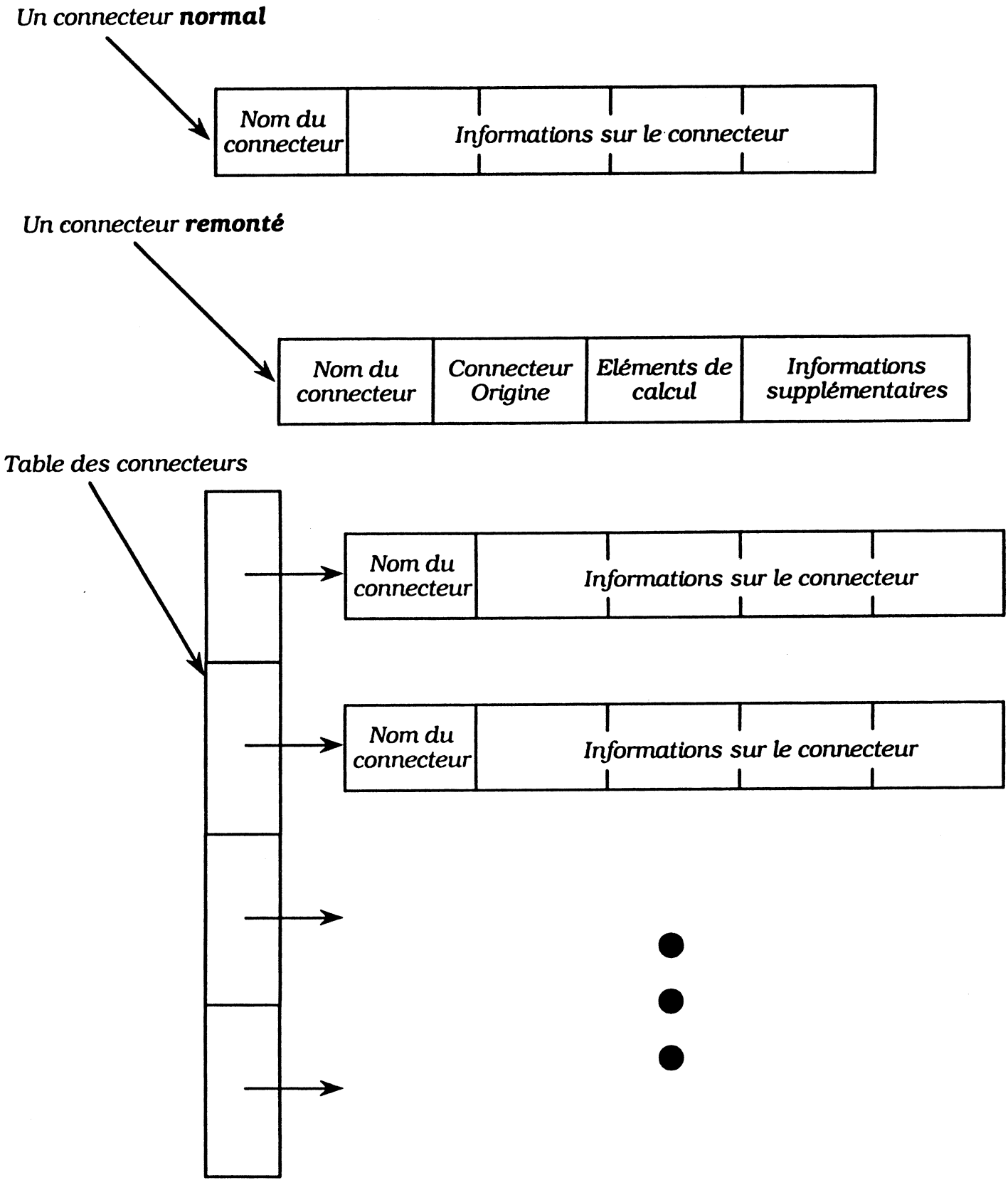


Figure IV.4 : Mécanisme de gestion des connecteurs : "Structure des connecteurs"

objet créé, et alors que *instance* ne remonte aucun connecteur dans la cellule appelante, *instance-C* remontera tous les connecteurs non marqués.

Notons aussi la fonction *:addinst* qui est une fonction système permettant d'ajouter un objet de type instance dans une cellule. Cette fonction est utilisée par les fonctions d'instanciation et effectue les opérations de vérification de cohérence (voir les fonctions *:MAJ*)

Exemple :

```
(defcell 'AS)
        (acces 'AS)
        (instance 'mot1 'p1 '(10 . 20) ())
        (fin-acces)
```

Lors de l'instanciation d'une cellule il est possible d'effectuer un certain nombre d'opérations géométriques (symétries,...). Nous allons décrire ici les possibilités offertes.

IV.2.7.2 Répétitions

Les répétitions permettent lors de l'instanciation d'une cellule de la dupliquer dans une direction donnée avec un décalage donné.

(rep <nbre de fois> <direction> <décalage>)

IV.2.7.3 Rotations

On dispose de 3 types de rotations (permettant des rotations de 90, 180 et 270 degrés).

(rot <angl> <centre de rot>)

IV.2.7.4 Symétries

On dispose des symétries classiques (par rapport à l'axe des x , des y, passant par un point)

(sym <axe> <point>)

La syntaxe générale dans l'instanciation des cellules est :

(instance <cellule> <N-instance> <Position> <Lparam>)
avec <lparam> correspondant à la liste des transformations à appliquer à l'instance créée.

Exemple :

```
(defcell 'AS)
(acces 'AS)
(instance 'mot1 () '(20 . 10) '((rep 3 'Nord 0)(sym 'Ox '(20 . 0))))
(fin-acces)
```

IV.2.8 Fonctions sur les boîtes

Lors de la création d'une cellule celle-ci n'a pas de boîte prédéfinie, mais dès la première instance ou le premier rectangle ajouté, la gestion de la boîte englobante est tenue à jour.

(B? {<cellule>})

Cette fonction renvoie la valeur de la boîte englobante de la cellule courante. Si la cellule n'est pas précisée cette fonction concerne la cellule courante.

(cal-boi <boîte> <liste de transf>)
(calc-boi {<cellule>})

Les fonctions *:calc-boi* et *calc-boi* permettent de recalculer les boîtes des cellules et sous-cellules. La fonction *:cal-boi* est utilisée pour calculer les effets des transformations sur une boîte. Cette fonction est utilisée par les précédentes. On peut voir ci-dessous l'algorithme utilisé :

Algorithme :
[1] mise à jour de la valeur d'une boite
(Calc-boi elt boite)

Si c'est une cellule : Si le corps est vide : boite = ((0 . 0) 0 . 0)
Sinon : on applique [1] au corps

Si c'est un motif : Si le motif est externe : la valeur est inchangée
(elle a du être lue lors de la lecture
du motif en bibliothèque)

Si le motif est interne : boite = PGRC des rectangles de
la vue topologique

(PGRC = Plus Grand Rectangle Commun)

Si c'est un corps : Pour chaque élément i appartenant au corps :
boite = appliquer [1] à la boite
actuelle et à l'élément i.

Si c'est une liste d'appels de cellule : Union [boite des cellules instanciées composée avec la position de l'appel] composée avec la liste de transformations

Si c'est un appel d'outil : Application de l'évaluateur spécifique
associé à l'outil appelé

[2] Calcul des transformations (ex : répétition)

(calcul-transfo boite L-transformation)

Pour chaque transformation de L-transformation faire :
Si transf = rep [facteur direction decalage]
Boite = Union [translation boite
et
si direction = nord :
(0 . (facteur-1) * (decalage + hauteur de la boite))
si direction = est :
((facteur-1) * (decalage + largeur boite) . 0)
si direction = ouest :
((1-facteur) * (decalage + largeur boite) . 0)
si direction = sud :
(0 . (1-facteur) * (decalage + hauteur de la boite))
]

IV.2.9 Changement de mode

Les fonctions décrites ici permettent les actions sur les différents modes existants (mode topologique : Ctopo ; mode logique : Clogi ...).

Mode corps? vide

La fonction *:Mode* permet de selectionner le mode dans lequel on souhaite travailler. Toutes les actions sont effectuées dans le mode dit *courant*. La fonction *corps?* permet de manipuler un corps spécifique d'une cellule. Il suffit de préciser le type de corps (Ctopo, Celeç...). Le résultat de la fonction corps étant un objet différent de *nil* dans tous les cas (qu'il soit vide ou non), on dispose de la fonction vide qui renverra nil si le corps est vide et qui sinon renverra le contenu effectif du corps.

IV.2.10 Primitives spécialisées

Ce paragraphe décrit les primitives spécialisées dans la manipulation d'objets, d'ajout d'objets pour les différentes vues.

IV.2.10.1 Les primitives topologiques

Ajout de rectangles :
(addrect N-cellule x y Dx Dy niveau)
La fonction addrect permet d'ajouter un nouveau rectangle dans un motif en tenant à jour la boite englobante du motif.

IV.2.10.2 Les primitives électriques et logiques

Le système doit offrir des fonctions équivalentes à la fonction topologique *addrect*, permettant d'ajouter dans le modèle électrique ou logique les éléments correspondant à la vue topologique.

Pour la vue électrique l'élément de base sera le transistor :
(addtransist N-cellule L-connect type); type pouvant valoir : P, N, etc
Pour la vue logique l'élément de base sera la porte logique : Ceci comprend l'inverseur, le non-et logique et le non-ou logique :

(addporte N-cellule L-connect type) ; type pouvant valoir : Inv, Nand, Nor, etc

IV.3 Les outils de base

Il s'agit des outils de base offerts par le système NAUTILE. Nous décrirons essentiellement des outils de placement et de routage, en effet les générateurs de structures régulières (type PLA's ou RAM's,...) n'ont pas encore été installés.

Placement de cellules :

On dispose pour l'instant de trois types de placement (avec ou sans routage).

IV.3.1 Placement direct

Ces outils de placement permettent de placer une cellule *collée* à une autre en connectant ensemble les connecteurs placés les uns en face des autres (voir figure IV.5).

(direct <direction> <N-appel1> <appel2> <arg>)
directa direct-C directa-C

Exemple :

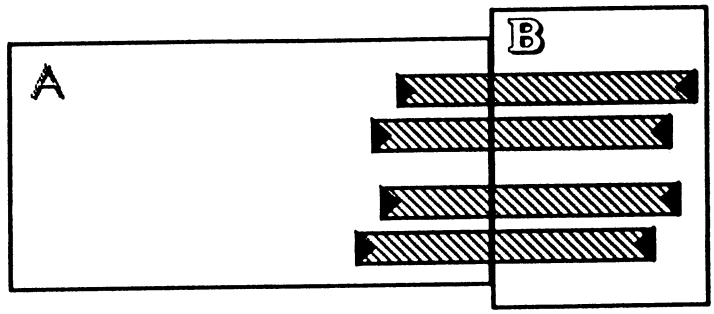
```
(defcell 'A)
        (acces 'A)
        (chargeq NAUTILE fich1 A)
        (fin-acces)
        ;Définition et chargement d'une nouvelle cellule A
(defcell 'B)
        (acces 'B)
        (chargeq NAUTILE fich2 B)
        (fin-acces)
        ;Définition et chargement d'une nouvelle cellule B
(defcell 'AS)
        ;Définition de la cellule AS
        (acces 'AS)
        (direct-C 'Est
                (instance1-C 'A () '(0 . 0) ())
                B )
        ;On assemble B avec A vers l'Est
```

IV.3.2 Les différents interfaces

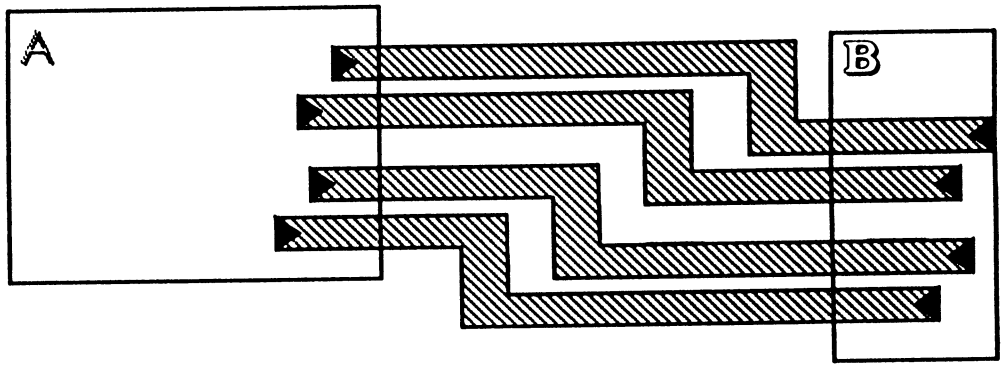
Les routeurs qui vont être décrits dans la suite ont été réalisés dans le cadre du projet GEODE [Mas86]. Ces routeurs permettent de réaliser deux types de routage : le routage mono-couche (Riviera) et le routage bi-couche (Corynthe) ou *channel routing*. Il est possible de traiter des canaux à bord crénelés, avec des terminaux sur grille ou hors grille (le routeur bicouche utilise Riviera pour ramener dans une première étape les terminaux sur grille).

La mise en oeuvre de ces routeurs a été faite dans NAUTILE.

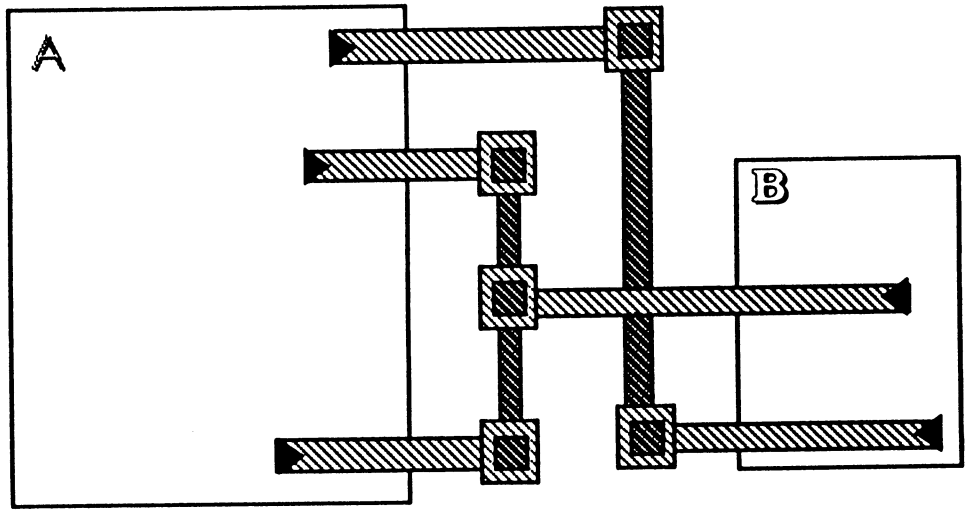
Les routeurs de GEODE sont accessibles par diverses sémantiques : on peut leur envoyer par message la distance entre deux fils de même couche, entre un via et un fil, le déport autorisé d'un fil sur un via ainsi que les matériaux utilisés. Tous ces paramètres sont réglables à l'aide d'un fichier *technologique* qui contiendra pour les différents matériaux utilisés les distances à donner.



[a] Assemblage Direct (direct)



[b] Routage Mono-couche (dev)



[c] Routage Bi-couche (rout)

Fig IV.5 : Les différents types d'assemblage

L'appel d'un routeur (nous donnerons ici l'exemple du routeur bicouche) doit s'opérer en deux étapes :

— On doit connaître le sens du routage (vers l'est, le nord,...), et instancier les différentes cellules formant la première rive du canal. L'instanciation s'effectuera en *remontant* les connecteurs et en affectant aux noms d'équipotentielles les mêmes noms pour les connecteurs à relier ensemble.

— Il existe deux méthode d'appel du routeur : soit il faut donner au routeur les cellules à instancier sur l'autre rive soit (si les cellules ont déjà été instanciées) le routeur effectue le routage si la place disponible est suffisante en reconnaissant les connecteurs de l'autre rive par leur direction (il est possible de donner d'autres moyens de reconnaissance, voir la fonction *trouve-l*). Il est possible aussi de spécifier les frontières réelles du canal (bords crénelés, correspondants aux frontières des cellules), ceci se fera automatiquement lorsque les cellules pourront recevoir des frontières polygonales.

Le routeur mono-couche ne tient évidemment pas compte des équipotentielles et relie dans l'ordre les connecteurs d'une rive et de l'autre.

IV.3.3 Routage élémentaire mono-couche (utilisation des outils GEODE)

On dispose d'une fonction *dev* permettant le routage mono-couche également appelé dévoiement, entre deux cellules. (Il n'y a pas de contrainte sur le nombre de cellules à assembler) (voir figure IV.5).

(:dev <direction> <N-appel1> <appel2> <lmina> <arg>)

IV.3.3.1 Définition d'un problème de routage mono-couche

Région de routage

C'est une région comprise entre deux blocs et pouvant servir au routage. De plus on a les lois suivantes :

a. Une seule couche est disponible.

b. Toutes les bornes sont sur la même couche.

c. Chaque réseau a exactement deux bornes.

d. Les bornes sont placées de telle facon qu'aucun croisement de fils ne soit nécessaire.

Il est possible de préciser le problème en un problème de *river-routing* en ligne droite grâce aux règles suivantes :

a. La région est déterminée par deux fonctions opposées, monotones soit en X, soit en Y.

b. Toutes les bornes sont sur des segments horizontaux (ou verticaux) des frontières si les deux fonctions sont monotones en X (ou en Y)

c. Chaque réseau a une borne sur chacune des deux frontières.

IV.3.3.2 Méthode ([HSU 83])

Définitions

— Une liste de segments est **continue** si le point de départ de chaque segment, excepté celui du premier, est commun avec le dernier point du segment précédent.

— Un **chemin** est une liste continue de segments, en alternance horizontaux et verticaux.

— Une borne connectée avec le premier segment du chemin sera appelée borne de **départ,** une borne connectée avec le dernier segment du chemin sera appellée borne d' **arrivée.**

Choix du sens des chemins

On choisit de décrire les réseaux dans le sens inverse des aiguilles d'une montre. Pour choisir le plus court chemin on calcule la portion de frontière parcourue dans le sens trigonométrique pour aller de la borne de départ à celle d'arrivée et on la compare avec la valeur totale de la longueur de la frontière.

Ordre de routage

Un réseau est routé lorsque tous les réseaux ayant une borne le long de la portion de frontière parcourue dans le sens trigonométrique, auront été routés.

Pour déterminer l'ordre de routage on gère une liste circulaire contenant toutes les bornes dans le sens trigonométrique.

-) Algorithme d'ordonnancement des réseaux.

---

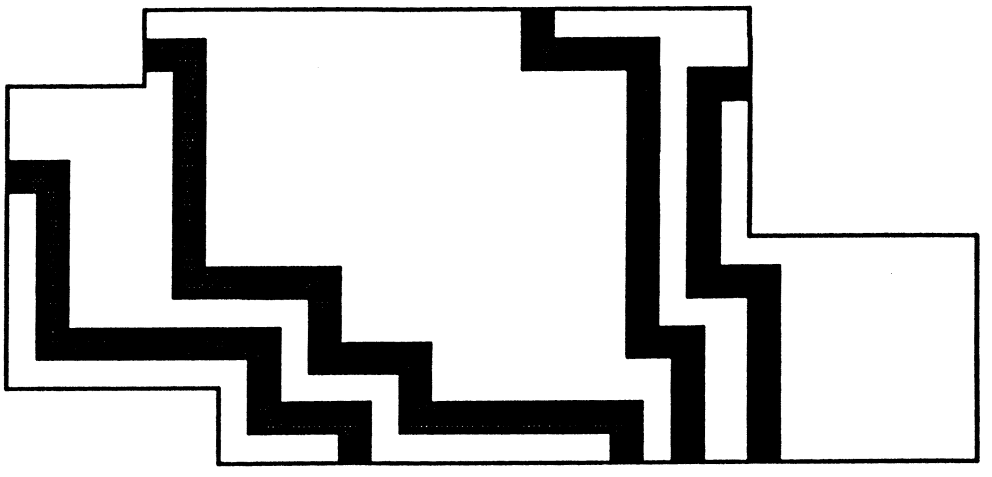
(1) pile PL = vide;
i = 1;

```
N = nombre total de réseaux;
T = une borne quelconque de la liste circulaire;
toutes les bornes de départ sont marquées NON-ENTRE;
toutes les bornes de sortie sont marquées NON-CORRE;

(2) TANT-QUE i <= N FAIRE
    DEBUT
    SI T est une borne de départ marquée NON-ENTRE ALORS
      DEBUT
      empiler T dans PL;
      marquer T ENTRE;
      FIN
    SINON
      DEBUT
      SI T est une borne d'arrivée marquée NON-CORRE ALORS
        DEBUT
        SI T et sommet de PL appartiennent au même réseau ALORS
          DEBUT
          marquer T CORRE;
          depiler PL;
          assigner le numéro i au réseau;
          incrémenter i d'une unité;
          FIN
        FIN
      FIN
    T = prochain borne dans la liste circulaire;
    FIN;
```

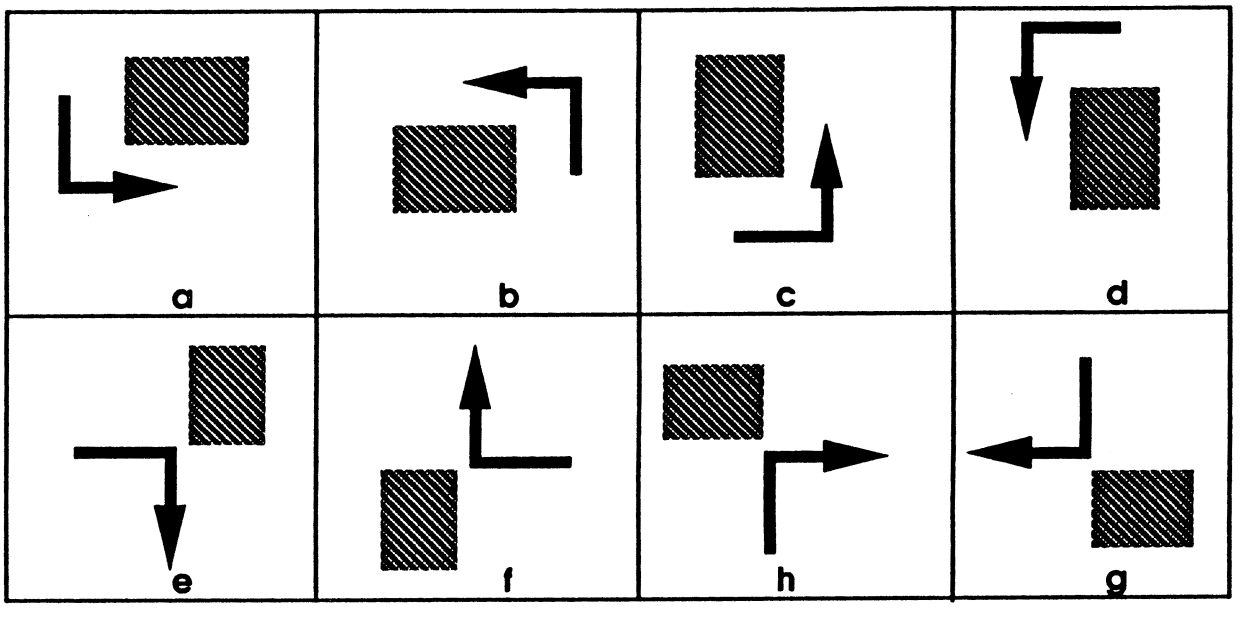
Les réseaux seront routés par ordre croissant de leur numéro d'assignement (figure IV.6 [a]).

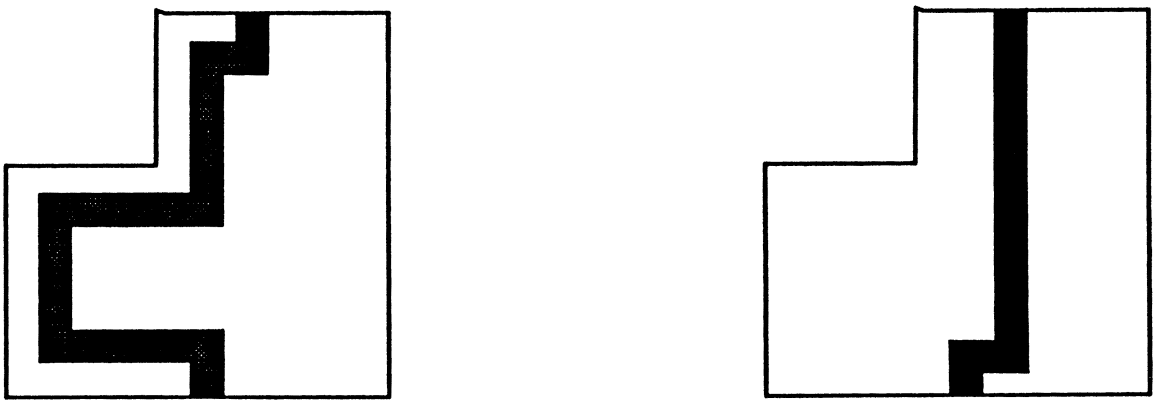
Pour chaque réseau on construit deux listes de segments de contraintes qui sont créées à partir des segments de la frontière et du routage déjà existant, *exposé* au réseau courant. Un chemin est alors formé dans le sens trigonométrique, le plus près possible de la liste des contraintes, commençant à la borne de départ et terminant à la borne d'arrivée du réseau. On vérifie ensuite que le chemin trouvé ne viole aucune règle de dessin avec l'autre liste de contraintes. En cas de violation le routage est impossible pour manque de place .

Minimisation des coins



[a] Exemple de routage mono-couche





[b] [c] Redressement des coins

IV.6 : Routage Mono-couche

Après l'étape précédente on dispose d'une solution non-optimale. Chaque réseau a un chemin qui lui est associé.

Il faut à présent optimiser cette solution en supprimant les coins inutiles, dirigés vers l'intérieur de la surface de routage. Ceci est effectué réseau par réseau, dans l'ordre inverse du précédent.

En tenant compte de l'ordre trigonométrique choisi on peut transformer tous les coins de type a, b, c ou d par des coins de type e, f, g ou h en les *retournant* (voir figure IV.6 [b] [c]).

Avant de retourner un coin il faut vérifier que les règles de dessin sont observées. Si on détecte une violation de ces règles on passe au coin suivant sans modifier le coin courant.

IV.3.3.3 Application au routage en ligne droite

Conventions

On appellera une des frontières la frontière du haut et l'autre celle du bas. Soit Xhi (resp. Xbi) la coordonnée en x de la borne du réseau i sur la frontière haute (resp. basse).

SI Xhi < Xbi ALORS le réseau est dit descendant.
SI Xhi = Xbi ALORS le réseau est dit trivial.
SI Xhi > Xbi ALORS le réseau est dit montant.

Algorithme

Cet algorithme procède réseau par réseau, de la gauche vers la droite, en générant un chemin par réseau.

(1) POUR CHAQUE réseau i de gauche à droite FAIRE

SI i est trivial ALORS

rejoindre les deux bornes par un segment vertical;

SINON

générer une liste continue des contraintes, L, cette liste comprend le dernier réseau routé, ou si c'est le premier, les portions de frontière haute (resp. basse) comprises entre le dernier réseau et le réseau courant si celui-ci est montant (resp.

descendant).

(2) Générer une liste continue de segments, séparée de L par l'espacement minimum entre deux fils.

(3) Générer deux segments verticaux depuis les deux bornes du réseau i jusqu'à la liste obtenue dans (2). Détruire de la liste obtenue, les segment situés à gauche de la borne haute pour les réseaux descendant et basse pour les réseaux montant. Le résultat est une liste continue de segments qui constitue le routage du réseau i.

Remarque

Une fois le routage effectué, on peut connaitre la largeur de canal nécessaire et éventuellement rapprocher les deux blocs (en effet le routage des réseaux montant (resp. descendant) est fait de telle sorte que les *coins* sont tous regroupés en haut (resp. bas) du canal.

IV.3.4 Routage bi-couche (utilisation des outils GEODE)

La fonction *rout* est similaire dans son utilisation à la fonction de dévoiement, faisant intervenir de plus les notions d'équipotentielles (les connecteurs à relier entre eux doivent avoir les mêmes noms d'équipotentielle) et introduisant la possibilités de canaux à bords irréguliers. *(:rout <Direction> <N-appel1> <appel2> <arg>)*
poser-seg poser-via

Exemple d'utilisation :

```
(de assemblage (x y z)

 (defcell 'AS)
 (acces 'AS)

 ; Ouverture de la cellule d'assemblage AS

 (defcell 'Ns1)
 (acces 'Ns1)

 ;Définition et ouverture de Ns1 (sous cellule de AS)

 (addconn-l 10 'Nord
          (20 28 36 44 52 60 68 76 84 92)
```

```
            (cirlist 110)
            2 'Alu 'bus-1
            '(p uy y r d x x y ui uy))

;Ajout d'une liste de 10 connecteurs dans Ns1

(boite 'Ns1 '((0 . 0) 100 . 110)
(fin-acces 'Ns1)

(defcell 'Ss1)
(acces 'Ss1)

;Définition et ouverture de Ns1 (sous cellule de AS)

(addconn-l  10 'Nord
            (10 20 30 40 50 60 70 80 90 100)
            (cirlist 0)
            2 'Alu 'bus-1
            '(ui d p y uy r a z a z))

;Ajout d'une liste 10 de connecteurs dans Ss1

(boite 'Ss1 '((0 . 0) 110 . 110))

; Creation de la boite de Ss1

(fin-acces 'Ss1))

(instance 'Ns1 'p1 (cons x y) ())
(rout 'Nord 'p1 'Ss1 z)

;Routage effectué

(Geff)
(D? 'AS '(0 . 0))

; Représentation sur l'écran du résultat

(fin-acces))
```

IV.3.4.1 Définition d'un problème de routage dans un canal

Définition d'un canal

On peut définir un problème de routage dans un canal par les éléments suivants:

1. Un canal a une longueur L. La plupart des routeurs placeront en x=0 l'extrémité Ouest du canal et en x=L+1 l'extrémité Est. Ainsi on aura une colonne verticale pour chaque valeur de x entre 1 et L.
2. On associe deux listes, Nord = (N1,N2,...,NL) et Sud = (S1,S2,...,SL) au canal,avec Ni (resp Si) = numéro du réseau auquel doit être relié le ième connecteur Nord (resp Sud), ou 0 si un tel réseau n'existe pas.
3. On dispose de deux ensembles E et O contenant les numéros des réseaux devant sortir des cotés Est et Ouest du canal.

Définition de la solution

De même la solution à ce problème contiendra les éléments suivants:

1. W la largeur du canal (en fait le nombre de pistes utilisées). Cela correspondra aux coordonnées y du canal.
2. Pour chaque réseau n, un ensemble de segments verticaux et horizontaux, connectés entre eux, dont les extrémités sont des points de la grille de coordonnées (x,y) telles que :

   0<y<W+1 en ayant x=i et y=0 (ou y=W+1) si : Ni=n (ou Si=n). De plus x<1 (ou x>L) sont légales mais doivent être évitées dans la mesure du possible.

   Et répondra au conditions suivantes :

3. Si O (resp E) contiennent un réseau n, alors il existe un segment dont une extrémité a pour coordonnées (0,j) (resp (L+1,j))
4. Deux segments de même direction sont placés sur une même couche, et ne doivent pas entrer en collision s'ils n'appartiennent pas au même réseau.
5. Deux segments du même réseau et de directions différentes, ayant une extrémité commune sont dits connectés par un via (si les segments n'appartiennent pas au même réseau, ils se croisent).

On peut définir la densité d'un canal par le nombre maximum de réseaux ayant des connecteurs sur les deux côtés de la ligne x=i pour tous les i (en excluant les connecteurs pouvant être reliés par un segment vertical)

Ou : W= Sup |Reseaux n tels qu' il existe Ni=Sj=n et i<j et i<=k<=j |
k=1..L

Donc la densité de ce canal est égale à 3.

IV.3.4.2 Left Edge Algorithm et dérivés ([Deu76] [Yos84],)

Définitions préliminaires

— Graphe des contraintes verticales:

Ce graphe représente les contraintes verticales imposées par les segments verticaux (voir exemple figure IV.7).

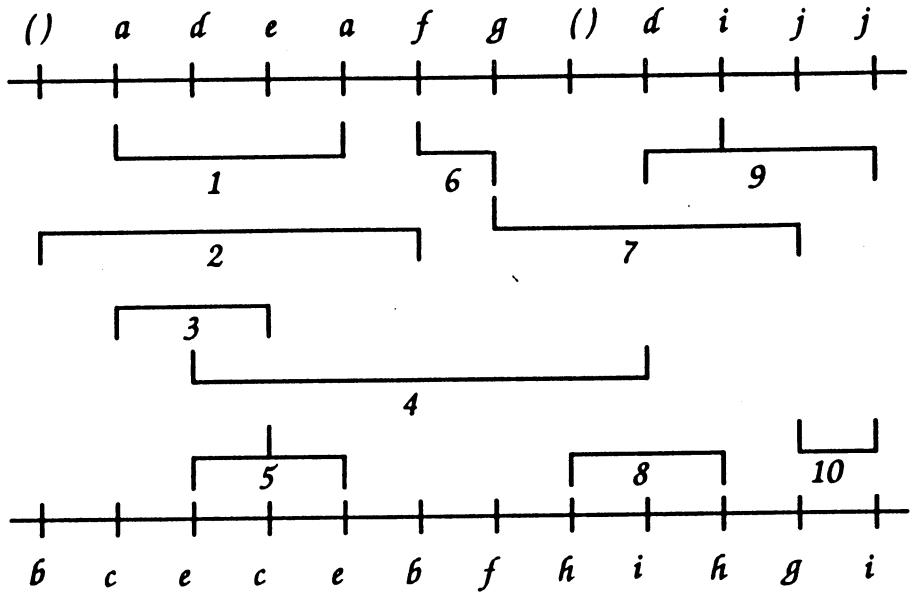
— Zone de représentation des segments horizontaux:

Un segment horizontal est déterminé par ses extrémités Ouest et Est. Soit S(i) l'ensemble des équipotentielles ayant des segments passant par la colonne i. Etant donné que les différents segments ne doivent pas se chevaucher, il en résulte que les segments appartenant à S(i) ne doivent pas occuper la même piste horizontale. Cette condition doit être satisfaite sur chacune des colonnes i. De plus il est uniquement nécessaire de considérer les S(i) qui ne sont pas sous-ensembles d'autres S(i). On assigne donc aux colonnes où S(i) est maximum un numéro de zone (voir figure IV.8). Le nombre maximum d'éléments dans une zone est appelé **densité locale,** et le nombre maximum d'éléments de toutes les zones est appelé **densité maximale.**

Description du Left Edge Algorithme ([Deu76])

Le principe de cet algorithme est de traiter les pistes horizontales une par une depuis le Nord jusqu' au Sud.

```
POUR CHAQUE  piste i FAIRE

  TANT QUE il existe un segment plaçable FAIRE

      placer  sur la piste i  le segment le plus  à l'Ouest des
      segments plaçables

    (Note : Un  segment  est plaçable  si tous ses  ancêtres ont
          déjà été placés et si aucun  segment de la même zone
          n' est déjà placé sur la piste i)

  FIN TANT QUE

FIN POUR
```



Donnera le graphe des contraintes suivant :

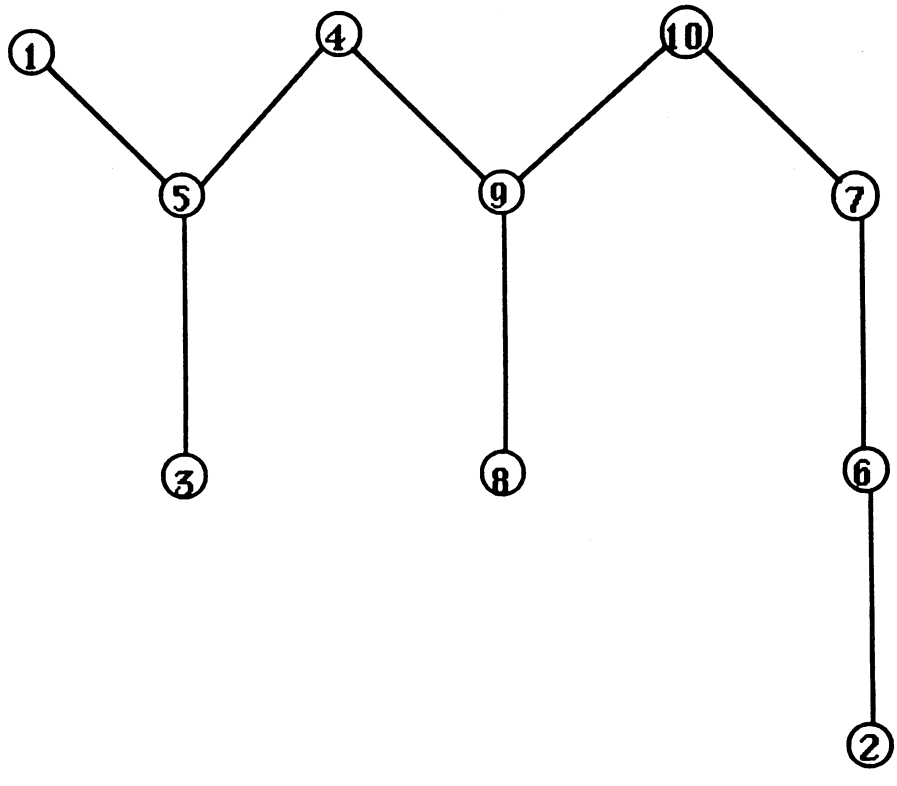


Figure IV.7 : Graphe des contraintes

Cet algorithme produira toujours une solution s'il n'y a pas de boucles dans le graphe des contraintes verticales.

Amélioration par la méthode de Yoshimura ([YOS-84])

Une des caractéristique du *Left Edge Algorithm* est de placer les segments sans tenir compte de la hauteur du graphe des contraintes. Si la longueur du plus long chemin partant du segment à placer, dans le graphe des contraintes, est L, alors il reste au moins L pistes à remplir (à chaque itération). Il est donc préférable de choisir les segments à placer de telle sorte que le plus long chemin diminue de longueur. De plus pour choisir quel segment placer, la notion de place perdue doit intervenir, en effet il est préférable de router en priorité les segments passant par des zones *haute densité*. On utilise donc une notion de poids pour chaque segment permettant de décider quel segment placer.

— Calcul du poids d'un segment donné:

Posons :

Zn = Ensemble des zones auquelles appartient le segment n dans la représentation des zones.
|Zn| = Nombre de zones dans Zn
Ln = Longueur du plus long chemin qui comporte le segment n dans son graphe des contraintes verticales.
Dn = Degré de l'horizontale n dans le graphe des contraintes verticales (nombre de fils de n).
P = Paramètre.
Dmax = Densité maximale.
Dz = Densité locale.
G = Fonction paramètre.
Uz = G (Dmax - Dz).

et soit :

$$W_n = \sum_{z \in Z_n} U_z + (P * L_n + D_n) * |Z_n|$$

Wn sera ici la fonction de poids souhaitée dans le paragraphe précédent.

Autres améliorations possibles

Colonne	S(i)	Zone
1	2	1
2	1 2 3	
3	1 2 3 4 5	
4	1 2 3 4 5	
5	1 2 4 5	
6	2 4 6	2
7	4 6 7	3
8	4 7 8	4
9	4 7 8 9	
10	7 8 9	
11	7 9 10	5
12	9 10	

Figure IV.8.a : Table de représentation des zones

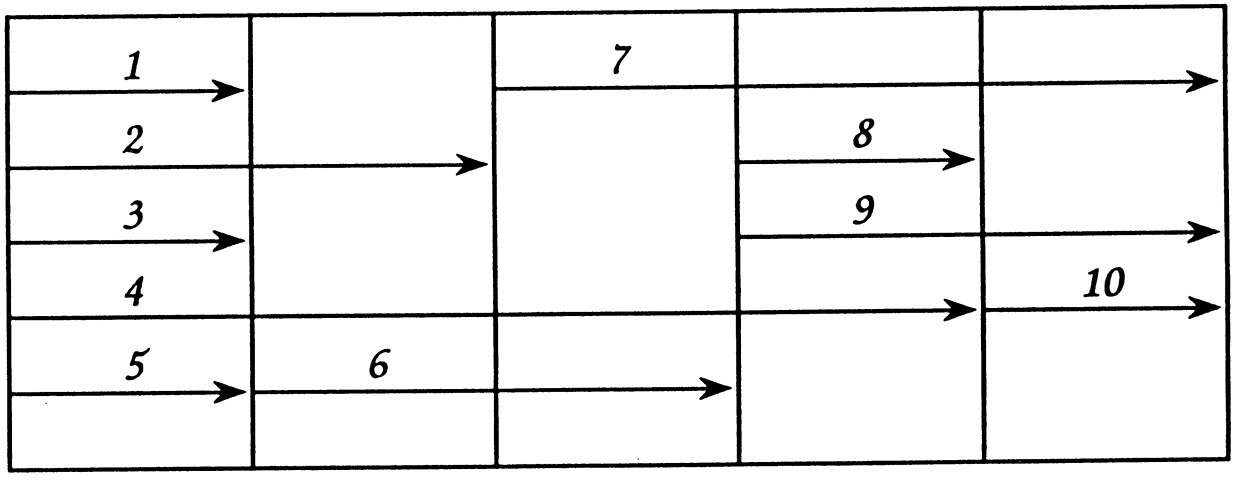


Figure IV.8.b : Graphe des zones

Cet algorithme se programme facilement et permet de faire des adaptations: la version disponible dans NAUTILE du routeur bicouche GEODE [Mas86] permet de router dans un canal aux bords irréguliers ou crénelés, d'interdire des zones du canal aux vias (changements de couche) et de briser les chaînes de contraintes occasionnant des boucles dans le graphe des contraintes.

Ceci est obtenu en intervenant simplement sur le poids des segments (Pour interdire une zone du canal à un segment on altère son poids en lui donnant une valeur infinie dans cette zone). Les résultats obtenus restent corrects quant à la vitesse d'exécution et au nombre de pistes occupées (voir exemple Figure IV.9)

IV.4 Utilitaires divers

Nous décrivons ici des utilitaires en rapport direct avec la gestion du système (destruction d'éléments de la structure, affichages textuels, graphiques,...).

IV.4.1 Effacement de cellules

Il est fréquent de rencontrer des problèmes de place mémoire dans le système. NAUTILE fonctionne sous le système Le_Lisp qui utilise une gestion de mémoire attribuée par catégories d'objets. On a par exemple un espace mémoire alloué pour la zone des listes, pour les vecteurs, pour les chaines de caractères, pour les symboles,... Une cellule va occuper de la place dans un certain nombre de ces zones. La place occupée par ces cellules, peut être récupérée simplement en libérant le symbole associé à la cellule.

IV.4.2 Effacement d'instances

Il existe des fonctions permettant d'effacer des instances déjà créées. Ces fonctions vont devoir remettre à jour la boite englobante de la cellule courante (voir *cal-boite*) et effacer les connecteurs remontés provenant de cette instance. On dispose de *rinst et rinsta* pour ces opérations (effacement d'une instance particulière ou bien de toutes les instances provenant d'une même cellule).

IV.4.3 Effacement de fonctions

Il est apparu en travaillant dans un environnement restreint que l'on pouvait parfois manquer de place mémoire. Pour cette raison il est apparu nécessaire de supprimer parfois de l'environnement des fonctions non utiles. Cet effacement peut s'effectuer sélectivement suivant les *packages* des fonctions (ces fonctions proviennent directement d'utilitaires Le_Lisp déjà existants):

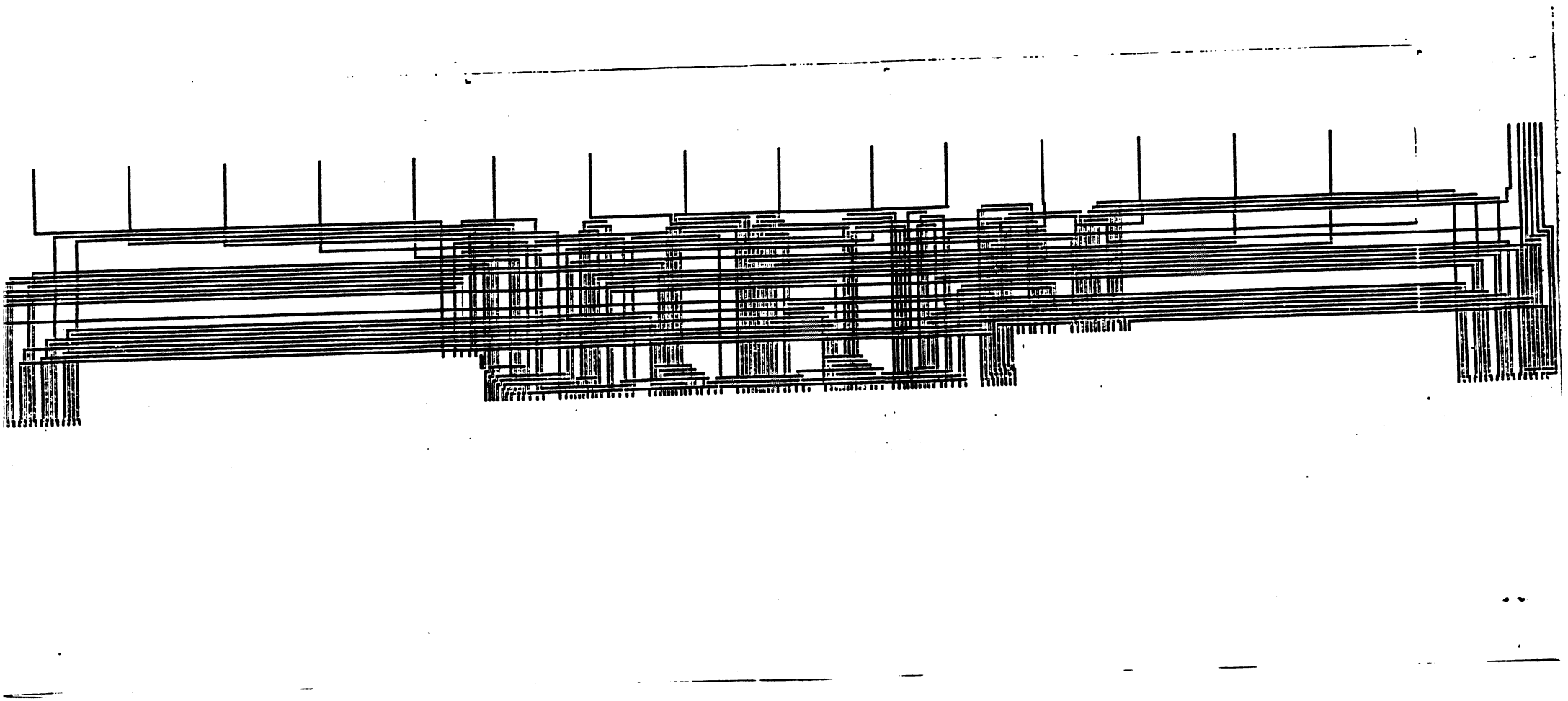


Figure IV.9 : Exemple de routage

(efface <pckg1> <pckg2>...<pckgn>)

IV.4.4 Représentation de cellules

On dispose de plusieurs méthodes de représentation des cellules pour le concepteur, deux méthodes purement textuelles et une graphique :

— le programme permettant de générer la cellule (suite de fonctions du système NAUTILE - voir exemple SYCO IV.3),

— une forme *lisible* de la structure de données (on utilise des sémantiques spéciales d'impression - voir exemple),

— la représentation sur écran graphique.

IV.4.4.1 Mode textuel

Il existe sous Le_Lisp la possibilité de redéfinir les sémantiques d'impression d'une structure de données. Cette possibilité a été utilisée pour obtenir une représentation écrite de la structure de données. Ceci se traduit par la définition d'une fonction d'écriture spéciale pour chacun des objets élémentaires de la structure de données :

Type général de l'algorithme utilisé :

Si l'objet à imprimer est de type STRUCT :
    Imprimer *Mode Courant* suivi de la valeur du champs Mode courant
    Imprimer *Liste de hiérarchie*
        Appliquer imprimer pour chacun des éléments de la
        liste des hiérarchies
Si l'objet est une cellule
    Imprimer *Nom* suivi de la valeur du champs Nom
    Imprimer *Filles* suivi de la liste des noms des filles
Si ...

Chacun des objets voit ainsi définir sa propre sémantique d'impression.
Les fonctions suivantes sont utilisées :

—*long-impr (changement du type d'impression),*

—*A? (recherche du nom exact de la cellule et renvoi de l'objet)*

—print

IV.4.4.2 Mode graphique

La mise en oeuvre du premier prototype du système NAUTILE n'a pas eu pour but la réalisation d'un nouvel éditeur graphique ; cependant, il est indispensable de disposer d'un *mini* éditeur graphique, permettant une représentation graphique des cellules dessinées. Pour cela on a recréé quelques fonctions de base graphiques : *Gframe Ggraph Ginit Gpos Gtext Gvect* .
Ces primitives graphiques permettent de dessiner un rectangle, de passer en mode graphique, d'initialiser le terminal, de positionner le curseur, de passer en mode texte et de dessiner un vecteur. Ces fonctions sont utilisables sur deux types de terminaux :

— Colorix 90 [Gui84]

— Tektronix couleur série 41
Le principe de représentation graphique des cellules est identique à celui décrit précédemment.

Type de l'algorithme :

Si l'objet est une cellule :
    Dessiner sa boite englobante
    Dessiner chacun de ses connecteurs
    Pour chaque appel :
        Dessiner chacunes des cellules appelées avec les
        transformations indiquées aux positions indiquées
            (éventuellement dessiner l'icône)
Fin de Si

Si l'objet est un connecteur :
        Dessiner un triangle dans la direction indiquée de la
        taille indiquée.
Fin de Si

Si l'objet est un motif
        Dessiner chacun des rectangles du motif
            (éventuellement dessiner l'icône)
Fin de Si

On peut citer les fonctions utilisées : *D? dessiner interdit I-trx I-try co coefx coefy*

:repres aff-connect affic affic1

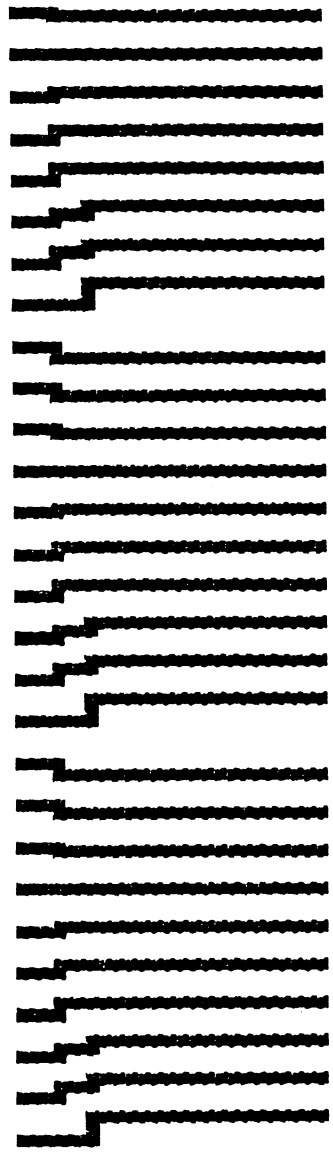


Figure IV.10 : Exemple de routage mono-couche obtenu à l'aide de NAUTILE

V. Liaison avec SYCO - Réalisation d'un Générateur de partie controle

Le système NAUTILE a été utilisé dans le cadre du projet Sycomore pour réaliser un Générateur de Partie Controle (GPC) [GeJ87] [Ger87]. Ce générateur se décompose en deux parties :

— Un générateur de PLA Cmos

— Un programme d'assemblage de parties contrôles (placement routage) à partir d'une bibliothèque de cellules.

Pour générer une partie contrôle GPC doit générer une bibliothèque de PLA's puis va assembler ces cellules en ciblant une architecture type.

On retrouve dans les arbres d'appels et de définitions créés l'architecture physique du circuit intégré. La partie contrôle va être divisée en plusieurs étages qui seront interconnectés par 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). On retrouve la structure SYCO (voir II.2.2.3 et fig V.1) dans l'organisation des cellules.

Module Générateur de Partie Contrôle

```
;Il s'agit ici d'une version simplifiée d'un module générateur d'un
;étage quelconque de la partie contrôle.

(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
```

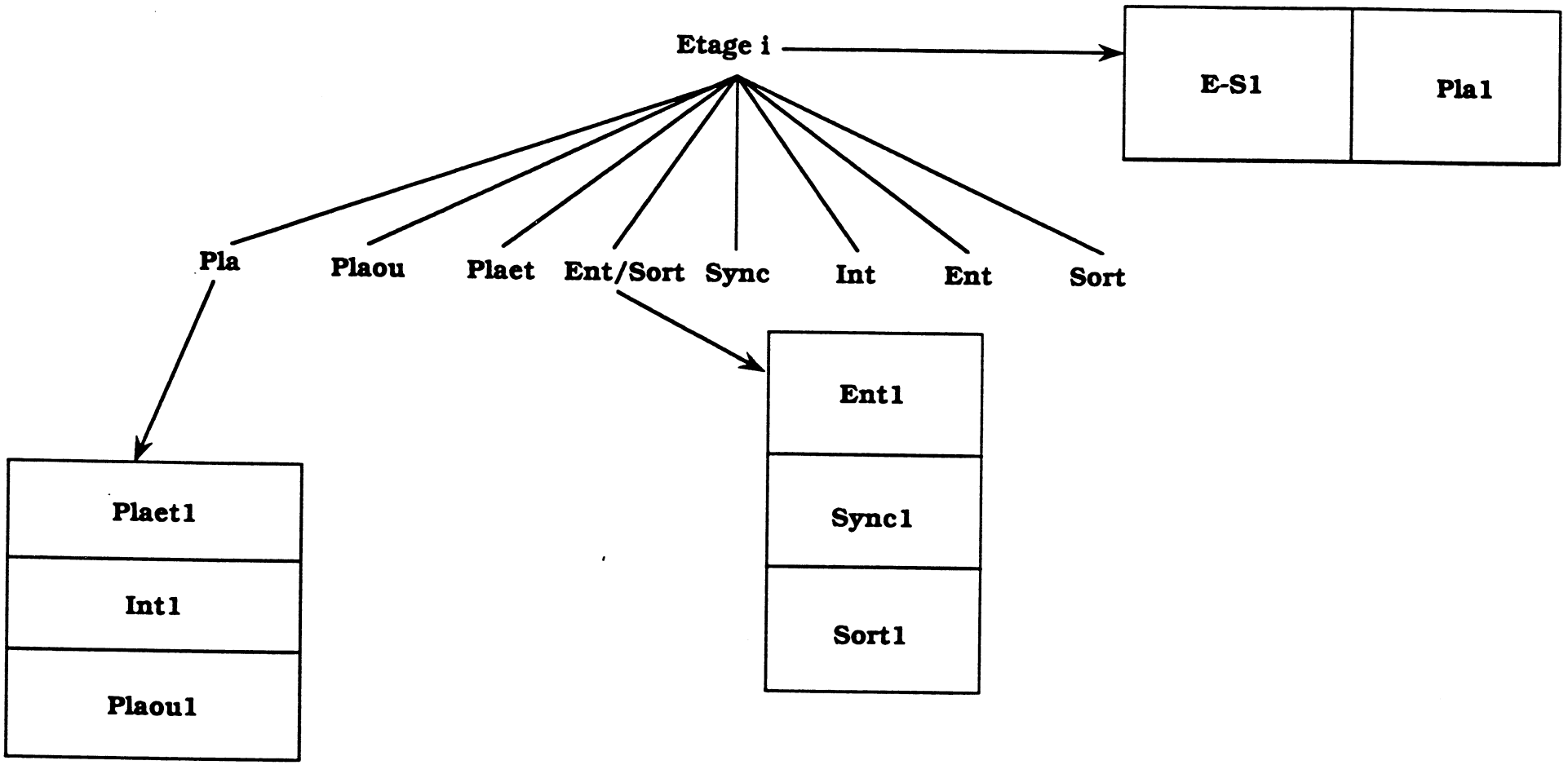


Figure V.1 : Génération d'un étage de la partie contrôle

```
(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)
```

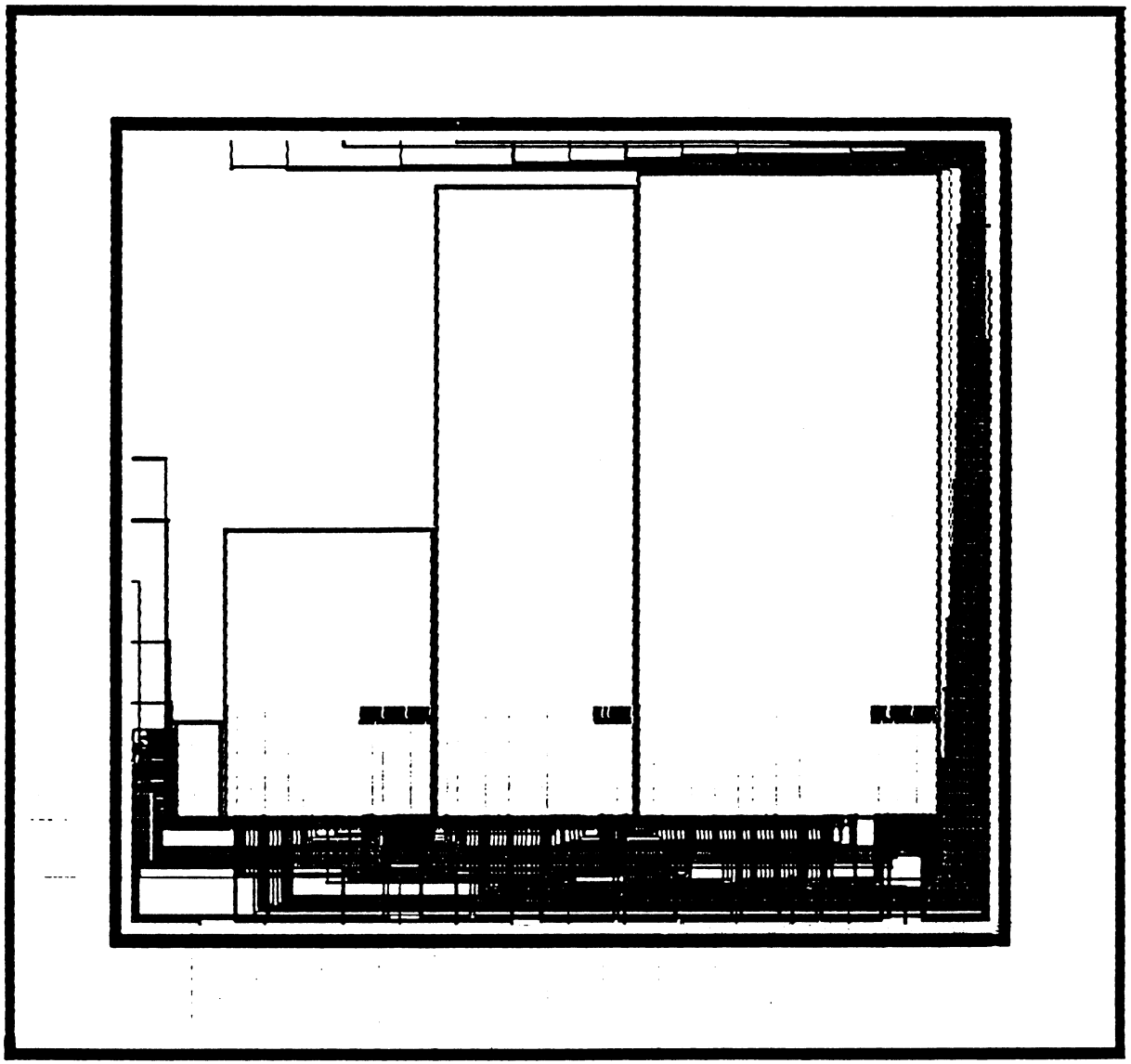


Figure V.2 : Exemple de layout de Partie Contrôle (NAUTILE)

```
;********** 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 contrôle dans son entier il faudra assembler les différents étages entre eux par l'intermédiaire d'un bus.

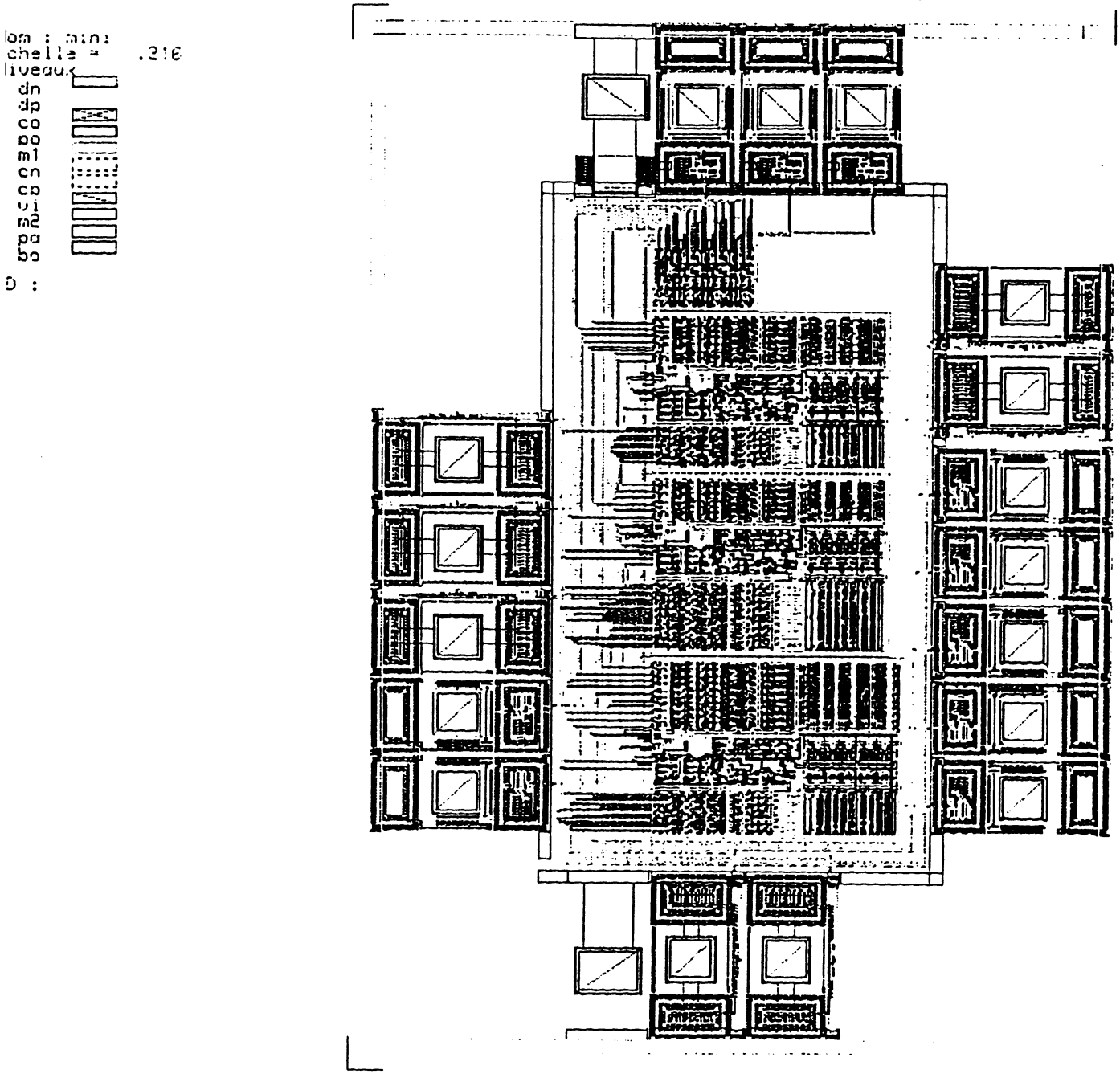


Figure V.3 : Exemple de layout de Partie Contrôle (LUCIE)

VI. Conclusion

Pour conclure cette thèse on peut d'une part tirer un bilan de l'expérience NAUTILE et d'autre part dresser une liste des développements possibles.

VI.1 NAUTILE : le bilan

NAUTILE a permis de développer un certain nombre de nouveaux concepts. On peut citer entre autres :

— Les notions de motifs et surtout de motifs externes, permettant de manipuler des cellules sans se préoccuper de ce qu'elles contiennent, d'utiliser des cellules décrites à l'aide d'autres systèmes et même de générer à partir de cellules NAUTILE, des cellules utilisables par d'autres systèmes.

— NAUTILE a permis aussi d'envisager un système indépendant de la technologie, pouvant, lors d'un changement de technologie, générer une nouvelle description correcte sans une intervention majeure d'un concepteur. Ceci est permis par le principe de la vue de construction qui permet de sauvegarder des structures de données de circuits non figées dans une technologie donnée et par l'introduction de fichiers technologiques.

— NAUTILE a introduit aussi les principes de traitement unifié de différentes vues (topologique, électrique, logique,...) et le principe de la vue de construction à partir de laquelle on peut générer toutes les autres vues à condition, bien sur, que les motifs de base aient été définis au préalable.

— Enfin NAUTILE a offert un environnement au compilateur SYCO , une structure de données pour son étape finale de génération des masques et des outils d'assemblage

— On peut ajouter que l'utilisation de NAUTILE dans un langage interprété de type Lisp permet au concepteur de travailler interactivement, passant sans interruption de la phase de conception à une représentation graphique.

VI.2 Et demain ...

Le prototype du système NAUTILE actuel ne peut être considéré comme complet. Il manque encore un certain nombre de potentialités qui n'ont pas été réalisées. Seul un noyau du système existe actuellement, permettant déjà, cependant, une utilisation du système (par exemple la partie contrôle d'une version TIM3 du 6502 a été réalisée à l'aide de NAUTILE).

On peut citer ici quelques uns des points importants qui pourront bientôt être mis en oeuvre :

Le problème de la gestion de la cohérence n'a pas encore été mis en pratique. Seuls les éléments de base ont été introduits et la structure de données est prête à les recevoir, mais il n'y a encore aucune vérification réellement effectuée. Ces vérifications de cohérence doivent être faites essentiellement à partir des dates de création et des dates de mise à jour, ceci par des vérifications d'antériorité et par des indicateurs de cohérence (ils mentionneront le passage d'un vérificateur, d'un comparateur,...).

L'ensemble des règles d'assemblage n'a pas encore été complètement défini. Il reste à compléter les règles d'assemblage topologique déjà existantes (pour les outils d'assemblage : les distances fil à fil, fil à via,... sont paramétrées), et surtout il faut définir des règles d'assemblage électrique, voire logique. Ces règles seront appliquées et vérifiées lors de chaque appel d'un outil d'assemblage. Ces règles d'assemblage seront décrites en fonction de la technologie. On peut ajouter pour la vérification et l'application des règles d'assemblage que si la structure peut déjà accueillir les ombres et les éléments partagés, ceux-ci ne sont pas encore traités.

Pour l'instant le système NAUTILE permet de lire et d'écrire du "LUBRICK", et de manipuler des cellules LUCIE sans les interpréter. Dans un très proche avenir il devrait permettre de s'interfacer également avec RNL. On envisage également d'interfacer NAUTILE avec d'autres systèmes suivant les disponibilités offertes. De toutes façons, l'ajout d'une nouvelle compatibilité avec un système sera toujours simple à réaliser en utilisant une méthode similaire à celle utilisée pour le modèle LUBRICK ou LUCIE déjà existant.
En effet pour le traitement des cellules importées depuis un autre système, NAUTILE se contente de manipuler le nom du système, le nom du fichier associé, et éventuellement la vision externe de la cellule. La génération d'une cellule dans un système externe se réalisera à l'aide d'un simple parcours hiérarchique de la cellule, (il s'agit en général de la définition d'une nouvelle sémantique d'impression des cellules).

A l'heure actuelle le système NAUTILE dispose d'un ensemble d'outils réduit, composé essentiellement de routeurs. Cependant lors de l'utilisation de NAUTILE pour SYCO , des générateurs de PLAs ont été écrits dans l'environnement NAUTILE. Il faut ajouter aussi que, grâce à la compatibilité de NAUTILE avec d'autres systèmes, il est toujours possible d'utiliser les outils disponibles sur ces systèmes. Il est possible, par exemple, d'utiliser des DRC fonctionnant sur des descriptions LUCIE que le système NAUTILE aura générées.

Le système NAUTILE ne dispose pour l'instant que d'un outil d'affichage graphique, c'est-à-dire qu'il ne peut que dessiner des cellules présentes dans la structure de données. Toute la conception d'un circuit se fait par instructions textuelles. Cependant il est prévu que NAUTILE comporte un éditeur graphique évolué permettant l'édition de cellules et l'appel des différentes fonctionalités (instance, assemblage,

générateur,...). Le langage textuel existant est déjà prévu pour avoir son langage graphique équivalent. On peut citer par exemple les fonctions d'ouverture, fermeture de cellule (acces, fin-acces), d'assemblage,...

On peut conclure en précisant que jusqu'à présent NAUTILE constitue surtout une spécification d'un système automatique et indépendant de la technologie de conception de circuit intégrés (VLSI). Le prototype existant ne constitue pas une fin en lui-même mais doit permettre de montrer les avantages que peut apporter à la conception des circuits, un système tel que NAUTILE.

VII. Annexe I : Glossaire

* **Abouter :** On dit qu'on aboute une cellule à une autre, si on la place exactement contre elle (placer bout à bout).

* **Alias :** Les alias permettent d'avoir des celllules différentes ayant des vues communes (par exemple : deux cellules ayant des vues électriques identiques, mais des vues topologiques différentes).

* **Arbre d'appel :** Arbre formé naturellement par les appels des cellules à des sous-cellules, motifs, . . . Les branches de cet arbre seront les appels, les noeuds seront les cellules et les feuilles seront les motifs.

* **Bâtons (dessins en) :** On parle de dessin en batons pour les cellules représentées sous forme symbolique. Les rectangles dans les différentes couches composant la cellules ne sont pas étendus et sont représentés sous forme de fil sans épaisseur. Ceci simplifie considérablement la tâche du concepteur et, avec des règles de dessin appropriées, lui permet de dessiner correctement sa cellule.

* **Canal :** On appelle canal de routage une surface (qui peut être ouverte) réservée au routage pour l'assemblage de deux ou plusieurs cellules.

* **C.A.O. :** Conception Assistée par Ordinateur

* **Cellule :** Elément de base du système, créé par l'utilisateur, appelant des motifs et ne comportant qu'une seule vue.

* **Collage :** On dit d'une ligne qu'elle est collée à 0 (resp. 1), si elle a constamment la valeur 0 (resp. 1)

* **Contact :** Un contact sert à relier électriquement deux couches technologiques différentes, on parle de via si ces deux couches sont en Métal

* **Couche :** Un circuit intégré est composé de rectangles de différents matériaux "posés" les uns sur les autres. Chacun des matériaux utilisés correspond à une "couche" différente (appelée aussi niveau). On peux citer en exemple les couches suivantes : Aluminium (Alu1 et Alu2), Polysilicium (Poly), Diffusion (Diff).

* **Connexion :** Liaison électrique entre différents points (ensemble de connecteurs) de diverses cellules.

* **Contact :** Liaison physique entre deux niveaux technologiques différents.

* **Dur :** On dit d'une cellule qu'elle est dure, si elle a été entièrement évaluée et ne comporte plus de paramètres, ni d'appels d'outils

* **Equipotentielle :** Ensemble de points (ou connecteurs) reliés ensembles électriquement

* **Fil :** Correspondance topologique d'une connexion électrique.

* **Frontière :** Représentation permettant de délimiter une cellule de son environnement extérieur.

* **Générateur :** Outil algorithmique permettant de générer une famille de cellules (PLA, mémoires, . . .)

* **Greedy Algorithm :** On appelle "greedy algorithm" (algorithme glouton) un algorithme résolvant les problèmes au fur et à mesure qu'ils lui sont posés. Exemple : un algorithme de routage glouton traitera le canal de la droite vers la gauche en faisant intervenir tous les terminaux dans l'ordre dans lequel ils se présentent.

* **Hiérarchie :** Ensemble de cellules composant un arbre d'appel.

* **Icone :** Représentation symbolique d'une cellule permettant d'imager sa fonction logique, électrique, . . .

* **Layout :** Dessin des masques

* **L.D.S. :** Langage de description des systèmes . Il s'agit du langage de description utilisé par le compilateur SYCO

* **Macro :** Suite d'opérations répétitives que l'on affecte à un mnémonique.

* **Motif :** Elément de base dans la hiérarchie comportant les quatres vues (électrique, logique, topologique et temporelle)

* **Mou :** On dit d'une cellule qu'elle est molle, si elle n'est pas entièrement évaluée et comporte encore des paramètres ou des appels d'outils

* **Niveau :** Les niveaux technologiques correspondent aux différentes couches de matériaux (Aluminium, Diffusion, Polysilicium, . . .) utilisées pour la fabrication d'un circuit intégré.

* **Ombre :** Zone réservée pour l'utilisation de certains niveaux technologiques

* **Outil :** Programme spécialisé dans l'exécution d'une tâche particulière : outil de routage, de compaction, . . .

* **Partagé :** Caractère d'un motif pouvant appartenir en même temps à deux cellules différentes.

* **Primitive :** Fonction de base du système, utilisée pour construire d'autres fonctions plus évoluées.

* **Processeur :** Synonyme d'Outil

* **Routage :** Ensemble de rectangles générés habituellement par un outil automatique (routeur) servant à réaliser la connexion entre deux cellules.

* **Soeur :** Une cellule A est dite soeur de B si A et B sont définies sous une cellule commune C.

* **Via :** Un via sert à relier électriquement deux couches technologiques de Métal (voir contact).

* **Vue (facette) :** Il existe quatre vues générales pour un motif : électrique, logique, topologique et temporelle. Le concept de vue permet de différencier l'accès, l'utilisation, ... du motif et par la suite de la cellule, suivant l'aspect de la cellule que l'on veut traiter

* **VLSI :** Very Large Scale Integration (très grande échelle d'intégration).

VIII. Annexe II : Structure de données de NAUTILE

(setq #:sys-package:colon '{Nautile})

STRUCTURE NAUTILE

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

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

:Ens-Var	
Champs	*Type*
Coefx	fix
Coefy	fix
Tranx	fix
Trany	fix
Interdit	(List fix)

(defmake {Ens-Var} :mkvar (Coefx Coefy Tranx Trany Interdit))

:CELL	
Champs	*Type*
Nom	symbol
Filles	(List symbol)
Connects	(List CONNECT)
Etat	ETAT
Corps	CORPS
Boite	Rect
Icone	ICONE

(defmake {CELL} :mkcell (Nom Filles Connects Etat Corps Boite Icone))

{CELL}:CELLULE	
Champs	*Type*
-	-

{CELL}:MOTIF	
Champs	*Type*
-	-

(defmake {CELLULE} :mkcellule (Nom Filles Connects Etat Corps Boite Icone))

(defmake {MOTIF} :mkmotif (Nom Filles Connects Etat Corps Boite Icone))

:CONNECT	
Champs	*Type*
-	-

{CONNECT}:CONN-N	
Champs	*Type*
Nom	string
Dir	string
Niveau	string
Position	Coord
Type	string
Taille	fix
Occupe	cons
Equi	string
N-inst	string

{CONNECT}:CONN-O	
Champs	*Type*
Nom	string
N-inst	string
N-connect	string
Type	string
Equi	string
Occupe	cons
Num	fix

(defmake {CONN-N} :mkconnect (Nom Dir Niveau Position Type Taille Occupe Equi))

(defmake {CONN-O} :mkorig (Nom Occupe Equi N-inst N-connect Num Type))

:ETAT	
Champs	*Type*
Date_Creat	DATE
Date_Modif	DATE
Origine	string
Coherence	List

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

:CORPS	
Champs	*Type*
Lcorps	(List) ;((Celec >)(Ctopo >) . .)
Lparam	(List) ; ELT ELT

(defmake {CORPS} :mkcorps (Lcorps Lparam))

:ELT	
Champs	*Type*
-	-

{ELT}:MOT	
Champs	*Type*
-	-

{MOT}:EXTERNE	
Champs	*Type*
Nom_fichier	string
Origine	string

(defmake {EXTERNE} :mkmext (Nom_fichier Origine))

{MOT}:INTERNE	
Champs	*Type*
Liste	(List)

(defmake {INTERNE} :mkmint (Liste))

:RECTANGLE	
Champs	*Type*
rect	Rect
niv string	

(defmake {RECTANGLE} :mkrect (rect niv))

{ELT}:LAPP	
Champs	*Type*
Lappel	(List APPEL)

(defmake {LAPP} :mklapp (Lappel))

:APPEL	
Champs	*Type*
-	-

{APPEL}:AP_CELL	
Champs	*Type*
Position	Coord
Nom-Def	symbol
Nom-Inst	string
Ltransf	(List)

(defmake {AP_CELL} :mkap_cell (Position Nom-Def Nom-Inst Ltransf))

{APPEL}:AP_OMBRE	
Champs	*Type*
Position	Coord
Nom-Def	symbol
Nom-Inst	string
Ltransf	(List)

{APPEL}:AP_OUTIL	
Champs	*Type*
Nom	symbol
Lparam	(List)

{AP_OUTIL}:ROUTEUR	
Champs	*Type*
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))

{AP_OUTIL}:CONNEX	
Champs	*Type*
-	-

:TERM	
Champs	*Type*
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)

:ASSEMB	
Champs	*Type*
-	-

(defmake {ASSEMB} ASSEMB ())

:ELECT	
Champs	*Type*
-	-

(defmake {ELECT} ELECT ())

:LOGIC	
Champs	*Type*
-	-

(defmake {LOGIC} LOGIC ())

:TEMPO	
Champs	*Type*
-	-

(defmake {TEMPO} TEMPO ())

:NAUTILE	
Champs	*Type*
-	-

(defmake {NAUTILE} NAUTILE ())

:LUBRICK	
Champs	*Type*
-	-

(defmake {LUBRICK} LUBRICK ())

:LUCIE	
Champs	*Type*
-	-

(defmake {LUCIE} LUCIE ())

IX. Annexe III : Manuel d'utilisation NAUTILE

(New AUtomatic Technology Independant Layout Environnement)

IX.1 Introduction

Le système NAUTILE est un système d'aide à la conception de circuits intégrés écrit en CEYX . Pour l'utiliser il faut donc disposer de l'interpréteur CEYX (basé sur Le_Lisp).

Le système Nautile fait appel à des notions de cellules et de motifs. Le motif est l'élément de base du système, considéré comme étant juste par définition . Chaque motif et chaque cellule a un nom, un ensemble de vues (électriques,topologiques,. . .) , des connecteurs.

Pour plus de détail sur les généralités du système, se reporter au rapport NAUTILE [BCH87]

IX.2 Chargement du système

Exécuter la fonction Naut :

? (Naut) ; Cette fonction charge effectivement Nautile

IX.3 Fonctions de base

Construction d'une nouvelle cellule : Les cellules ont deux noms : le nom donné par l'utilisateur selon la fonction de la cellule et le nom tenant compte de la hiérarchie dans laquelle est placée la cellule (cellules mères et filles).

Dans la suite tous les noms seront des noms utilisateurs sachant que le système gère automatiquement les noms dépendant de la hiérarchie.

IX.3.1 Définition d'une nouvelle cellule

? (defcell <nom de cellule>)
=<Nom réel de cellule>
; ceci correspond au nom dans la hiérarchie à l'aide d'un package décri-
; vant l'ascendance de la cellule

IX.3.2 Definition d'un nouveau motif

? (defmotif <nom de motif>)
=<Nom réel de cellule>
; ceci correspond au nom dans la hiérarchie

IX.3.3 Chargement de cellules depuis un fichier

? (chargeq <système> <fichier> <cellule>)
=<Nom réel de cellule>
;les arguments ne sont pas évalués

? (charge <système> <fichier> <cellule>)
=<Nom réel de cellule>
;les arguments sont tous évalués

Pour plus de commodité il existe aussi les deux fonctions "char-lib" et "char-libq" qui recherchent ou écrivent dans la bibliothèque prédéfinie correspondant au système.

Sauvegarde d'une cellule :

(genere <système> <fichier> <cellule>)
(genereq <système> <fichier> <cellule>)
(gen-lib <système> <fichier> <cellule>)
(gen-libq <système> <fichier> <cellule>)

Pour l'instant on dispose des systèmes suivants :

— NAUTILE

— LUBRICK

— LUCIE

Au départ les bibliothèques sont situées par défaut dans le directory BIBLI, chaque système ayant un sous-directory à son nom. Il est cependant possible de modifier ces directory par la fonction cd:

(cd <système> <directory>)
(il est préférable, mais non indispensable, que <directory> soit
donné par son chemin d'accès complet depuis la racine : /users/. . .)

De même pour savoir quelles cellules sont disponibles dans le directory d'un système :

(ll <système>)

IX.3.4 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 cellule courante.

? (acces <nom de cellule>) ; la cellule devient la nouvelle cellule courante
=<Nom réel de cellule>

A cet instant on peut construire à l'intérieur de la cellule.

Une autre fonction d'accès est disponible : accesa (acces absolu) qui permet d'accéder directement à une cellule par son nom réel (#:Tclass:Nautile:. . .).

IX.3.5 Accès à un corps particulier

Il existe différents sous-corps du corps d'une cellule qui correspondront aux différentes vues existantes pour cette cellule (électrique, topologique, temporelle, . . .). Pour cela il existe différents modes de travail permettant d'accéder aux différents corps existants. La fonction Mode renverra le mode courant :

? (Mode)
=Cgen ; mode courant
? (corps? <mode> <nom de cellule>)
;renvoie un pointeur sur le corps

avec <mode> qui peut valoir : Cgen, Ctopo, Clogic, Celec, Ctempo, . . .

De plus pour savoir si ce corps est vide :

? (sendq vide <corps>)
= ( ) ; si le corps est vide
(<corps> est le résultat de la fonction précédente)

IX.3.6 Fin d'accès à une cellule

? (fin-acces)

ou bien :
? (fin-acces <nom de cellule>)
; ceci ferme toutes les cellules ouvertes depuis la dernière ouverture de
; la cellule citée.

Les fonctions acces et fin-acces retournent la cellule courante.

Exemple :

? (defcell 'Essai)
=#:Tclass:Nautile:environnement:cell:Essai

? (acces 'Essai)
=#:Tclass:Nautile:environnement:cell:Essai

? (defmotif 'Essai2)
=#:Tclass:Nautile:environnement:cell:Essai:Essai2

? (acces 'Essai2)
=#:Tclass:Nautile:environnement:cell:Essai:Essai2

? (fin-acces 'Essai)
=#:Tclass:Nautile:environnement:cell

IX.4 Fonctions de placement

IX.4.1 Fonction de modification de la boite englobante

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

Cette fonction donne à la cellule demandée la boite d'origine dans son repère local (x y) et longueur et largeur Dx et Dy (voir aussi B?)

Exemple :

(boite 'Essai 0 -9 100 140)
=((0 . -9) 100 . 140)

IX.4.2 Fonction de placement absolu d'une cellule dans la cellule courante

? (instance <Nom de cellule> <Nom d'instance> <Position> <Liste de paramètres>)

Avec

<Nom de cellule> = nom de la cellule à placer

<Nom d'instance> = nom choisi pour l'instance ou bien ()

<Position> = position ou placer l'instance

<Liste de paramètres> = liste de transformations à appliquer à l'instance. La liste de transformation pourra comporter des symétries, des répétitions ou des rotations sous la syntaxe suivante :

<transformation> ::= <Rep>/<Sym>/<Rot>

<Rep> ::= (Rep <Nbr de fois> <Dir> <incrément>)

"instance" retourne un objet de type APP correspondant à l'instance créée. Il existe aussi une fonction "instance1" qui ne retourne que le nom de l'instance créée. Si le nom d'instance n'est pas donné, il sera généré automatiquement. De plus les connecteurs de l'instance seront reproduits dans la cellule courante.

Exemple :

```
? (acces 'Essai)
=#:Tclass:Nautile:environnement:cell:Essai
? (instance 'Essai2 'I1 '(10 . 10) '((rep 3 'Nord  0) (sym x 4)))
=<instance>
? (instance1 'Essai2 () '(30 . 30) ())
=Essai2-254
? (fin-acces)
= #:Tclass:Nautile:environnement:cell
```

IX.4.3 Fonction de placement relatif avec routage élémentaire

?(direct <Direction> <Nom de l'instance> <Nom de cellule>)

Avec

<Direction> = direction de l'assemblage

<Nom de l'instance> = nom de l'instance déjà existante qui va servir de repère pour le placement de la nouvelle instance.

<Nom de cellule> = nom de la cellule à placer

Cette fonction retourne le nom de la nouvelle instance créée. De plus un motif de routage sera créé qui contiendra comme vue topologique, les rectangles reliant les connecteurs placés les uns en face des autres.

?(directa <Direction> <Position> <Nom de l'instance> <Nom de cellule>)

Cette fonction est identique à la précédente et autorise le placement absolu.

Exemple :

```
? (acces 'Essai)
=#:Tclass:Nautile:environnement:cell:Essai

? (instance1 'Essai2 'I1 '(30 . 30) ())
=I1

?(direct 'Est 'I1 'Essai2)
=Essai2-255

? (fin-acces)
= #:Tclass:Nautile:environnement:cell
```

IX.4.4 Fonction de placement avec routage

? (dev <Direction> <Nom de l'instance> <Nom de cellule> {<Décalage>})

Avec

<Direction> = direction de l'assemblage

<Nom de l'instance> = nom de l'instance déjà existante qui va servir de repère pour le placement de la nouvelle instance.

<Nom de cellule> = nom de la cellule à placer

<Décalage> = argument optionnel indiquant le décalage avec lequel on va placer la cellule par rapport à l'instance citée

Cette fonction retourne le nom de la nouvelle instance créée. De plus un motif de routage sera créé qui contiendra comme vue topologique, les rectangles reliant les connecteurs .

IX.4.5 Fonction de placement avec routage bi-couche

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

Avec

<Direction> = direction de l'assemblage

<Nom de l'instance> = nom de l'instance déjà existante qui va servir de repère pour le placement de la nouvelle instance.

<Nom de cellule> = nom de la cellule à placer

<Frontière1> = liste de points permettant d'établir une frontière (routage crénelé)

<Frontière2> = liste de points permettant d'établir une deuxième frontière

<Décalage> = argument optionnel indiquant le décalage avec lequel on va placer la cellule par rapport à l'instance citée

<Lconnexion> = Liste de connexions à effectuer

Cette fonction retourne le nom de la nouvelle instance créée. De plus un motif de routage sera créé qui contiendra les rectangles reliant les connecteurs de même nom d'équipotentielle.

IX.5 Fonctions sur les connecteurs et connexions

IX.5.1 Fonction d'ajout d'un connecteur dans une cellule

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

Cette fonction construit et ajoute un connecteur dans la cellule demandée.

IX.5.2 Fonction d'extraction de connecteurs

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

Cette fonction renvoie la liste des connecteurs dans la direction donnée de la cellule demandée. Les connecteurs sont donnés sous la forme d'objets de type CONNECT (voir structure.ll). Il existe aussi la fonction conn-dira qui prend comme argument le

nom réel de la cellule (avec tous les packages dépendants de la hiérarchie).

? (depconn-dir <Direction> <Nom de cellule> <Vecteur>)

Cette fonction est identique à la précédente mais effectue une translation <Vecteur> sur les connecteurs et renvoie une liste ne contenant pas les connecteurs eux-mêmes, mais une copie sous une structure différente : TERM utilisée principalement pour les routages.

IX.6 Fonctions sur les rectangles

IX.6.1 Fonction d'ajout de rectangle dans un motif

? (addrect <nom de cellule> <x> <y> <Dx> <Dy> <niveau>)

Cette fonction ajoute un rectangle de longueur Dx et de largeur Dy , en position (<x> <y>), de niveau <niveau> dans la cellule demandée.

IX.7 Utilitaires d'intérêt général

IX.7.1 Sur les boites

(calc-boi {<Nom de cellule>})
=<boite>

Recalcule la boite englobante d'une cellule. Sans argument cette fonction s'applique à la cellule courante.

IX.7.2 Effacement de cellules, d'instances

(rcell <cell1> . . . <celln>)
=<liste des cellules effacées>

Cette fonction efface littéralement dans la structure les cellules demandées

(rinst <N-inst1> . . . <N-instn>)
=<liste des instances effacées>

Cette fonction agit dans la cellule courante et efface les instances citées. Il est conseillé après l'appel de cette fonction de recalculer la boite englobante de la cellule courante (voir calc-boi)

De même :

(rinsta <N-cell1> . . . <N-celln>)

Identique à la fonction précédente mais qui va effacer dans la cellule courante les appels aux cellules citées (appel à calc-boi conseillé)

IX.7.3 Effacement de fonctions

(oublie <Thème>)
=<liste de fonctions>

Avec <Thème> = Routeur , Assembl ou Graph Cette fonction sera utilisée pour regagner de la place en mémoire si l'utilisateur n' a plus besoin des fonctions sur le thème donné.

IX.7.4 Règlage de la technologie

Un certain nombre de paramètres de la technologie sont règlables dans le fichier techno.ll . Pour plus ample information consulter le manuel d'entretien.

IX.8 Visualisation des cellules

On dispose pour l'instant de deux processeurs d'affichage : l'un pour Tektro et l'autre pour Colorix 90. Pour représenter une cellule sur l'écran il faut sélectionner le processeur d'affichage courant. Ceci se passe au moment du chargement où il est demandé à l'utilisateur le type de terminal dont il dispose

- C90 = Colorix 90
- Tektro = Tektronix couleur (ex: 4111 4125 . . .)
- Autre = terminal normal sans possibilités graphiques évoluées

IX.8.1 Fonctions graphiques de base

? (Ginit)
=t
Initialisation du terminal graphique

? (Geff)
=t
Effacement de l'écran

? (coefx)
=<coef>

Retourne l'échelle de la dernière représentation graphique

? (interdit <liste>) =<liste de niveaux>

Cette fonction permet de sélectionner les niveaux dans la hiérarchie pour lesquels on ne veut pas afficher les noms des cellules. Sans argument cette fonction retourne la liste courante des niveaux non-affichés.

IX.8.2 Représentation sur écran graphique d'une cellule (ou d'une hiérarchie)

? (D? <nom de cellule> <position> {<echelle>})
=<Nom réel de cellule>
la position est donnée en coordonnées circuit sous la forme : (x . y).

? (P? <x> <y>)
=<X=??? Y=???>

Cette fonction reçoit des coordonnées écran et retourne les coordonnées circuit correspondantes. Cette fonction n'est disponible pour l'instant que sur Colorix (utilisable avec la souris)

IX.8.3 Représentation textuelle d'une cellule

? (A? <nom de cellule>)
=<cellule>
; ceci utilise des sémantiques d'impression spécialisées ({CELL}:prin . . .)

? (long-impr <ind>)
=<ind>
Avec <ind>=t ou () .

Cette fonction permet de passer du format d'impression long (t) au format d'impression court (())

? (B? <nom de cellule>)
=<boite>
; Retourne la boite englobante de la cellule

?(C? <dir>)
=<Liste de connecteur>
; Retourne la liste des connecteurs de la cellule courante dans la direction ; <dir>

?(E?)
=<environnement>
; Retourne l'environnement courant

IX.9 Exemples

```
(de essai ()
(defmotif 'ess)
  (acces 'ess)
    (boite 'ess 0 0 20 40)
    (addconn 'ess 'Est 1 4 2 3 () () )
    (addrect 'ess 30 50 46 41 4)
  (fin-acces))

(de essai2 ()
  (defcell 'ess1)
  (defcell 'ess2)
  (chargeq LUBRICK ampli A)
  (chargeq LUBRICK inv I)
    (acces 'ess1)
      (instance 'A 'E-1 '(0 . 0) ())
      (direct 'Est
                (directa 'Est '(5 . 6) 'E-1 I)
                'I)
    (fin-acces 'ess1))
```

Bibliographie

[Anc83] F. Anceau, **STYX: Descriptions squelettisèes normèes**, [Note interne,TIM3], Grenoble, Juin 1983.

[Anc84] F. Anceau, **Gènèration versus immersion et paramètrisation**, Tech. Rep. , TIM3, Grenoble, Mai 1984.

[AnJ84] F. Anceau et A. A. Jerraya, **Organisation hièrarchique de cellules**, TIM3, Grenoble, Juin 1984.

[Anc86] F. Anceau, **The architecture of micro-processors**, Addison-Wesley publishing company, Bull S.A. Paris, 1986.

[BaH80] J. Batali et A. Hartheimer, **The Design Procedure Language Manual**, Tech. Rep. 598, Massachussetts Institute of Technology, Sep. 1980.

[BCH87] P. Bondono, J. P. Caisso, A. Hornik et A. A. Jerraya, **Nautile: Dèfinition prèliminaire d'un système d'aide à la conception automatique de V. L. S. I.**, TIM3-INPG Laboratoire IMAG, Grenoble, Juin 1987.

[Bry84] R. E. Bryant, **Race detection in MOS circuits by ternary simulation**, California Institute of Technology, 1984.

[Bur86] M. R. Burich, **Design of module generators and silicon compilers**, Scuola Superiore G. Reiss Romoli: NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, Silicon Design Labs, Inc., L'Aquila, Italy, Juil. 1986.

[Bur83] M. Burnstein, **Hierarchical channel router**, D.A.C, IEEE (ed.), , 1983.

[Cai86] J. Caisso, **Simulation multivaluèe - Simulation Switch**, Rapport de Recherche No 597, TIM3/IMAG, Grenoble, Mars 1986.

[Cai87] J. Caisso, **Contribution a la vèrification des circuits intègrès dans un environnement multivaluè** , [Thèse de doctorat,INPG], Grenoble, Dec. 1987.

[CDD86] J. Chailloux, M. Devin, F. Dupont, J. Hullot, B. Serpette et J. Vuillemin, **LE_LISP Version 15.2 Le Manuel de rèfèrence**, INRIA, Rocquencourt, Mai 1986.

[Clo84] G. W. Clow, **A global routing algorithm for general cells**, DAC, IEEE (ed.), , 1984, 45-51.

[CGM87] G. P. Costantino, G. Ghisio, R. Manione, A. Patrucco et N. Rodaro, **A graphic editor for procedural cell design**, Scuola Superiore G. Reiss Romoli : Advanced summer course on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, Juil. 1987.

[Deu76] D. Deutsch, **A dogleg channel router**, D.A.C, IEEE (ed.), , 1976.

[Dri86] F. O. Drisceoil, **RNL - Lubrick Interface**, TIM3, Grenoble, Sep. 1986.

[EDI85] EDIF, **Electronic design interchange format, Version 1 0 0**, , 1985.

[FoC88] G. Fourneris et J. Chambon, **STYX: An object oriented, Lisp procedural based environment for open graphic design**, to be presented to DAC, IEEE (ed.), Thomson CSF, , 1988.

[GeJ87] J. P. Geronimi et A. A. Jerraya, **Architecture Interne des parties contrôles gènèrèes par SYCO**, IMAG/TIM3, Grenoble, Oct. 1987.

[Ger87] J. P. Geronimi, **Gènèrateur de partie controle -GPC-**, TIM3/INPG, Grenoble, Dec. 1987.

[GoR83] A. Goldberg et D. Robson, **SMALLTALK-80, The language and its implementation**, Addison-Wesley, Palo Alto, 1983.

[Gui84] O. Guillaumin, **Maquette d'un èditeur de circuits intègrès - Logiciel graphique sur COLORIX 90**, [Rapport de DEA,INRIA], Rocquencourt, 1984.

[Hig83] D. Hightower, **The Lee router revisited**, I.C.C.D, IEEE (ed.), , 1983.

[Hor86] A. Hornik, **Stratègies de routage pour circuits intègrès**, [rapport de DEA Informatique, IMAG-TIM3, Grenoble], Juin 1986.

[Hsu83] C. P. Hsu, **General river algorithm**, D.A.C, IEEE (ed.), , 1983.

[Hul84] J. Hullot, **Programmer en CEYX**, INRIA, Rocquencourt, Oct. 1984.

[JRR85] A. Jerraya, E. Rosier, F. R. Rougeaux et B. Courtois, **A hierarchical symbolic design layout tool: STYX**, VLSI, , 1985.

[JVJ86] A. A. Jerraya, P. Varinot, R. Jamier et B. Courtois, **Principles of the SYCO compiler**, DAC, IEEE (ed.), Las Vegas (USA), Juin 1986.

[JBG88] A. A. Jerraya, P. Bondono, J. P. Geronimi, A. Hornik et B. Courtois, **Nautile: A physical design environment based on an object oriented data manager**, Design Automation Workshop, IEEE, , Jan. 1988.

[Joo86] R. Joobbani, **An artificial intelligence approach to VLSI routing**, Kluwer Academic Publishers, Hingham, 1986.

[Kar83] K. Karplus, **CHISEL: An extension to the programming language C for VLSI layout**, [Phd dissertation,Stanford], Stanford, Fev. 1983.

[LaW86] H. S. Law et G. Wood, **A Mixed Approach to Module Generator Design**, Scuola Superiore G. Reiss Romoli: NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, Silicon Design Labs, Inc., L'Aquila, Italy, Juil. 1986.

[Lee61] C. Y. Lee, **An algorithm for path connection and its application**, IRE, T. computer (ed.), , 1961.

[Lip83] J. Lipman, "" **Lecture on VIP - User's Guide for"SuperVip"**, , 1983.

[Mas86] C. Masson, **SYCOMORE, Projet GEODE: Protocole d'interface des modules de routage**, Rapport de Recherche No , BULL S.A. Direction Technique Groupe, Les Clayes ss Bois, Juin 1986.

[Moo86] P. P. Moore, **Oct.: Database Programmer's Manual**, University of California, Berkeley, Jan. 1986.

[New86] A. R. Newton, **Symbolic layout and procedural design**, Scuola Superiore G. Reiss Romoli : NATO Advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, Juil. 1986.

[Pai85] J. Paillotin, **Le système LUCIE, Version du 1er Juillet 1985**, TIM3-INPG Laboratoire IMAG, Juil. 1985.

[RiF82] R. L. Rivest et C. M. Fiduccia, **A 'greedy' channel router**, D.A.C, IEEE (ed.), , 1982.

[RBS67] J. P. Roth, W. G. Bouricius et P. R. Schneider, **Programmed algorithms to compute test to detect and distinguish between failures in logic circuits**, IEEE Trans. Electron. Comput. EC-16 (10), (1967), 567-580, IEEE.

[Rou87] F. R. Rougeaux, **Outils de C.A.O. et conception structurèe de systèmes intègrès sur silicium**, [The'se de doctorat,I.N.P.G.], Grenoble, Fev. 1987.

[SDL86] SDL, **L-tutorial, L-language, SDL Generator libraries**, , Mai 1986.

[Sch83] J. P. Schoellkopf, **LUBRICK : A silicon assembler and its application to data-path design for FISC**, in VLSI 83, E. S. P. B.V (ed.), North Holland, 1983, 435-445.

[Sch85] J. Schoellkopf, **SILICIEL: Contribution à l'architecture des circuits intègrès et à la compilation du silicium**, [Thèse d'ètat, INPG USMG], Grenoble, Avr. 1985.

[Seg85] T. P. Segovia, **PAOLA: Un système d'optimisation topologique de PLA**, [Thèse de 3ème cycle,TIM3-INPG], Grenoble, Oct. 1985.

[Sou83] J. Soukup, **Routing on one layer - an algorithm and its implementation**, DAC, IEEE (ed.), , 1983, 126-134.

[SpL80] R. F. Sproull et R. F. Lyon, **The Caltech Intermediate Form for LSI Layout Description**, , 1980.

[Tri84] S. Trimberger., **VTIcompose - A Powerful Graphical Chip Assembly Tool**, DAC, IEEE, , 1984, 697-698.

[Ull84] J. D. Ullman, **Computational Aspects of VLSI**, Computer Science Press, 1984.

[WoD86] W. H. Wolf et A. E. Dunlop, **Symbolic Layout and Compaction**, Scuola Superiore G. Reiss Romoli: Advanced summer course on logic synthesis and silicon compilation for VLSI design, AT&T Bell Laboratories, L'Aquila, Italy, Juil. 1986.

[Xio86] J. G. Xiong, **Algorithms for global routing.**, DAC, IEEE (ed.), , 1986, 824-830.

[Yos84] T. Yoshimura, **An efficient channel router**, D.A.C, IEEE (ed.), , 1984.

TABLE DES MATIERES

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de Messieurs

. C. LANDRAULT
. C. MASSON

Monsieur HORNIK Armand

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Informatique "

Fait à Grenoble, le 26 mai 1989

Pour le Président de l'I.N.P.G.
et par délégation,
le Vice-Président
P. VENNEREAU

Resume :

Cette thèse constitue une contribution à l'élaboration d'un nouveau système de conception de circuits intégrés, NAUTILE. Elle comporte une étude des différents système existants et à partir de leur synthèse établit la définition d'un nouveau système. Celui-ci doit réaliser un environnement complet de conception de circuits V.L.S.I. permettant d'être facilement interfaçable avec différents systèmes déjà existants, d'être indépendant de la technologie et de gérer différentes représentations (dessin des masques, schéma électrique, schéma logique) d'un même circuit en assurant la cohérence entre elles.
Enfin cette thèse donne une description du prototype réalisé du système NAUTILE, consistant en une structure de données orientée objet, en les primitives de gestion de la structure, ainsi qu'en un certain nombre d'outils (routeurs, générateurs divers) ayant été mis en oeuvre.

Mots-cles : Environnement de conception, C.A.O., V.L.S.I., Représentation orientée objet, Représentation multiples, Routage, Lisp.