Description et simulation mixte analogique-numérique: analyse de VHDL analogique, réalisation d'un simulateur mixte

Dominique Rodriguez

THESE

présentée par

RODRIGUEZ Dominique

Pour obtenir le titre de Docteur

de l'Université Joseph Fourier - Grenoble I

(arrêtés Ministériels du 5 Juillet 1984 et du 30 Mars 1992)

(Spécialité INFORMATIQUE)

Description et Simulation Mixte Analogique-Numérique : Analyse de VHDL-Analogique, Réalisation d'un Simulateur Mixte

---

Date de Soutenance : 15 février 1994

Composition du jury :	Président	ADIBA Michel
	Rapporteurs	ISRAEL Michel
		VACHOUX Alain
	Directrice	BORRIONE Dominique
	Examinateurs	EL TAHAWY Hazem
		ROUQUIER Denis

Thèse préparée au sein des Laboratoires :

ARTEMIS/IMAG et CIT/MOS (CNET Grenoble)

Τυ πενσεσ ετ μοι φε συισ
Οβελιξ
(Αστεριξ λεγιονναιρε)

Remerciements

Je voudrais remercier :

Mme Dominique Borrione, directrice de cette thèse, pour son sérieux et tous ses conseils au niveau de mon travail,

MM. Mohamed Tawfik, Hazem El Tahawy et Larry Moore pour la confiance qu'ils m'ont faite en permettant de mener cette thèse au sein de la siociété Anacad EES,

M. Jacques Lecourvoisier pour m'avoir accuilli dans le département CIT-MOS du CNET Grenoble,

M. Michel Adiba pour avoir accepté la présidence de mon jury de thèse,

MM. Alain Vachoux et Michel Israël pour avoir accepté d'être rapporteurs,

M. Denis Rouquier pour ses conseils au cours de ce travail,

MM. Laurent Planchon et Mart Altmae pour leur aide dans la réalisation de l'interface mixte VHDL-ELDO,

M. Jean-José Mayol pour tout ce qu'il m'a apporté autant sur le plan professionnel que personnel,

M. Pascal Bolcato pour son soutient moral,

Tous mes collègues de travail d'Anacad, du CNET-Grenoble et du laboratoire ARTEMIS,

Evidemment je ne pourrais pas terminer ces remerciements sans citer mes amis et ma famille, et tout particuliérement Véronique et Mathilde pour tout l'amour et la joie qu'elles me procurent.

Introduction

Le travail de cette thèse a trait à la simulation de circuit VLSI et s'intègre donc dans le développement d'outils de CAO pour la micro-électronique.

En phase de conception, la simulation n'est pas la panacée pour la validation d'une réalisation. Voici pourquoi, en ce qui concerne l'électronique numérique, on voit apparaître des outils de synthèse de plus en plus performants et bientôt des outils de preuve formelle. Malheureusement, pour l'électronique analogique de tels outils sont encore à l'état de prototypes dans des universités, ou bien alors ils se limitent à un domaine bien particulier : par exemple la synthèse d'amplificateurs opérationnels. Pour toutes ces raisons, la simulation reste l'unique outil de validation pour les concepteurs.

Pour pouvoir simuler un circuit VLSI dans son ensemble, la simulation en mode mixte est la solution. En effet, elle permet de conserver une grande précision sur une partie du circuit et diminue le temps de simulation du reste. D'autre part, la conception de véritables circuits mixtes (convertisseurs, filtres) est actuellement en plein développement, la simulation en mode mixte est donc particulièrement adaptée pour ceux-ci.

D'autre part, depuis maintenant plus de cinq ans les concepteurs de circuits numériques possèdent un langage de description standard : VHDL. Comme nous le verrons dans le premier chapitre et plus en détail dans le second, VHDL est un langage très puissant qui permet la description du niveau porte logique jusqu'au niveau système.

Face à la demande d'un langage standard pour la description analogique, il a semblé naturel à certains d'étendre VHDL pour ce niveau de description. Nous avons donc pris une part active dans ce travail de normalisation, on peut dire ainsi que Grenoble a été un des moteurs de cette normalisation avec l'Université Joseph Fourier, le CNET et Anacad.

En plus de cette aspect langage, nous avons réalisé un simulateur en mode mixte à partir de VHDL : VHD$_e$LDO. Au delà de la réalisation informatique, nous avons travaillé sur une approche de description qui a permis de valider le bien fondé de certaines propositions relatives à la

normalisation de VHDL-analogique.

Voici, le plan de cette thèse.

Le premier chapitre est un rappel sur les langages de description de matériels. Pour l'électronique numérique, nous présentons les langages industriels les plus utilisés ainsi que le travail de recherche du groupe CONLAN. Pour la partie analogique, à partir d'une présentation de la simulation analogique, nous avons isolé quels sont les besoins d'un langage de description comportemental.

Le chapitre suivant est consacré au langage standard VHDL. Aprés une brève présentation de ce langage, nous faisons le point sur la normalisation de son extension analogique, en décrivant les différentes propositions, illustrées d' exemples.

Dans le troisième chapitre, nous abordons la simulation en mode mixte. Nous comparons les algorithmes de simulation numérique et analogique, et abordons le problème du partitionnement. Les paragraphes suivants sont consacrés aux différents problèmes liés à la réalisation d'un simulateur mixte. Enfin, nous analysons quelques réalisations.

Le dernier chapitre de cette thèse est la présentation de VHD$_e$LDO. Ce travail s'intègre dans une coopération entre Anacad et l'Institut de Micro-électronique de Suède d'une part, et Anacad et le CNET Grenoble d'autre part.

Chapitre I.
Les langages de description de matériel

I. Qu'est-ce qu'un langage de description de matériel

1. Généralités

Pour décrire un système quelconque, que ce soit un circuit électronique ou bien une voiture, il faut utiliser un langage. Bien entendu, on peut utiliser le français ou bien encore pour des raisons internationales l'anglais, mais ceux-ci ne sont pas vraiment appropriés et ils sont beaucoup trop compliqués pour être analysés automatiquement par voie informatique. C'est pourquoi des langages ont été définis : on les appelle les langages de description de matériels ou encore les HDLs (pour Hardware Description Language).

Dans les années soixante on ne parlait pas encore de langages mais plutôt de formats d'entrée. Ce fut, dans la décennie suivante avec tous les progrès réalisés en compilation et dans la structuration des langages informatiques que l'on a commencé à parler de HDL. Ainsi, on pourra remarquer la forte ressemblance entre la structuration de VHDL et celle du langage de programmation Ada.

L'évolution des langages informatiques se fait de telle façon que la partie d'optimisation système, c'est à dire du code ou du stockage des données est laissée au compilateur et non pas au programmeur comme ce fut le cas pendant longtemps. Ce qui fait que ce dernier ne se préoccupe plus que des problèmes d'algorithmique.

En ce qui concerne la description de matériel, l'avantage d'un langage par rapport à un simple format est sa meilleure structuration et aussi une sémantique plus complète. Au travers de cette thèse, on pourra comparer VHDL-analogique, qui est un HDL, avec Spice qui est plutôt un format. Donc, on aura la possibilité de juger avec cet exemple concret, les avantages et inconvénients de chacun. Ce qui gêne le plus les utilisateurs est sans aucun doute la plus grande lourdeur d'écriture

d'un langage, et les connaissances algorithmiques que cela requiert pour les parties comportementales.

Ces HDLs ont connu un grand essor pendant de nombreuses années surtout au niveau de la description des circuits électroniques numériques. Une série de conférences, la CHDL, a consacré de nombreuses sessions aux progrès réalisés dans la définition des concepts et des primitives de ces langages.

2. Les niveaux de description

La description d'un système peut être différente selon le domaine dans lequel on la fait. On trouve trois domaines pour la description : structurel, comportemental et physique. Dans notre cas, on étudiera les domaines structurel et comportemental; le domaine physique étant plus lié à la réalisation finale des circuits.

Chaque HDL permet la description de systèmes à un certain (ou plusieurs) **niveau** d'abstraction : ainsi on n'utilisera pas le même niveau pour décrire un automate ou un transistor. Dans le domaine de la micro-électronique, on trouve les niveaux suivants :

- électrique,
- interrupteur (Switch),
- logique ou porte,
- transfert de registre (RTL),
- algorithmique,
- système.

Les niveaux précédents sont donnés dans l'ordre croissant de niveau d'abstraction, du plus fin au plus général. Bien entendu, rien n'empêche de décrire un système en utilisant plusieurs de ces niveaux selon la finesse de description que l'on désire sur les différentes parties du système, il faut alors soigner l'interface entre les différents niveaux. On parle alors de descriptions **multi-niveaux**, mais nous y reviendrons plus loin.

Cependant, comme le dit D. Borrione [Bor81], il y a un nombre aussi grand que l'on veut de

niveaux d'abstraction.

Les HDLs servent non seulement d'interface entre les utilisateurs et les outils de CAO, mais aussi ils peuvent être utilisés comme langage de spécification entre un client et un concepteur de systèmes. Donc les outils de simulation, de synthèse ou bien encore de preuve formelle utilisent des HDLs.

II. La description numérique

1. Historique

C'est pour les circuits numériques que sont apparus réellement les langages de description de matériel au milieu des années 60. Pour ces circuits, les HDLs sont utilisés en majorité pour la simulation, mais de plus en plus pour la synthèse.

Au départ, les langages étaient juste des formats qui permettaient de décrire des circuits dans le domaine structurel; mais pour pouvoir simuler des systèmes plus gros il a fallu introduire la notion de description comportementale. Celle-ci, dans un premier temps, était faite dans des langages de programmation classique (Fortran, Pascal), mais devant leurs limitations, on les a d'abord étendus, pour ensuite réellement définir des langages dédiés à la description de matériel.

On a assisté ensuite, dans les années 70, à une explosion de HDLs, G. Lipovski a alors comparé cela à la Tour de Babel [Lip77]. C'est pour cette raison, que le département de la défense américain (DoD) a lancé un appel d'offre pour normaliser un HDL. Ce fut VHDL.

Dans ce qui suit, nous allons présenter le résultat d'une étude universitaire sur un concept de langage standard, ensuite nous présenterons rapidement ce qu'a apporté VHDL et enfin nous ferons le tour des standards industriels actuels.

2. L'approche CONLAN

a. Présentation

Le but de ce travail de recherche, mené par plusieurs universités, fut de définir un ensemble de

règles de dérivation et un langage de base, à partir desquels il est possible de définir un HDL dédié à un certain niveau d'abstraction. Le groupe de travail a commencé en 1973 pour répondre à la multiplication anarchique et impressionnante des HDLs. Il y a dix ans, ce travail a abouti à l'introduction de concepts nouveaux et à des résultats théoriques intéressants [Pil80-83-85].

Cependant, le laboratoire ARTEMIS, à partir de cette approche, a développé un projet de simulateur multi-niveaux et multi-modes, permettant de simuler du niveau électrique jusqu'au niveau système. Il s'agit du projet CASCADE [Bor93b]. Ce projet sera présenté plus loin. Tout d'abord, voyons les principales caractéristiques de l'approche CONLAN, et les nouveautés qu'elle a apportées.

Ce travail a permis de faire le point des connaissances sur les HDLs. Il n'était pas question d'en créer un de plus, mais plutôt de permettre d'arriver à un consensus (CONsensus LANguage) et ainsi de définir un langage de base standard BCL (Base ConLan) sur lequel seraient construits de futurs HDLs. L'autre concept était d'avoir un unique langage par niveau de description, intégrant les domaines structurel et comportemental.

Ce travail s'adresse à la fois aux utilisateurs et aux développeurs d'outils. Pour les premiers, le passage d'un HDL à un autre sera réduit en investissement puisque ils auront de nombreux points communs. D'autre part, pour l'utilisateur seul l'apprentissage de BCL est important, les règles de dérivation peuvent être laissées de côté.

Pour les développeurs d'outils de CAO, la création d'un langage dédié à leur application et au niveau sur lequel celle-ci va fonctionner, devient beaucoup plus facile grâce à la connaissance de BCL. De plus, l'outil bénéficie d'une étiquette standard, et ainsi il n'effraiera pas les utilisateurs avec un nouveau langage.

b. Le langage

CONLAN s'est appuyé sur les progrès faits au niveau des langages de programmation, et a hérité, entre autre, d'une structuration importante ainsi que d'un typage fort. D'autre part, la sémantique associée à BCL était définie formellement. Ces aspects l'ont démarqué des autres HDLs qui étaient

beaucoup plus simples.

Ainsi, dans CONLAN, il est possible de définir de nouveaux types, des fonctions, des sous-types avec héritage des fonctions du type père. On peut également citer la notion de surcharge d'une fonction ou d'un opérateur, et la définition de nouveaux opérateurs. Il est également possible de regrouper un ensemble de définitions dans une unité spécifique, qui peut être référencée dans d'autres unités.

Les descriptions des systèmes peuvent être paramètrées au moyen de constantes génériques, qui sont utilisées par exemple pour définir la taille d'un bus ou bien un retard de propagation.

Pour ce qui est de la gestion du temps, le concept de délais infinitésimaux pour représenter les pas de calculs est intégré. Ce concept est important, il permet de gérer les propagations à temps nul en conservant le déterminisme. Enfin, les événements sont générés à partir des états passés du système, et on modifie le présent en fonction du passé, et non le futur en fonction du présent comme en VHDL par exemple.

Evidemment, CONLAN permet de traiter les domaines comportementaux et structurels. D'autre part, toutes les instructions de CONLAN sont concurrentes, ce qui en fait un langage plutôt déclaratif [Bou91a].

c. Les mécanismes de dérivation

En plus, du langage de base, CONLAN possède un ensemble de mécanismes de dérivation qui permettent de définir un nouveau langage. Ce langage peut être dérivé soit à partir de BCL, soit à partir d'un langage dérivé à partir des mécanismes CONLAN. L'esprit de CONLAN est qu'un langage ne soit dédié qu'à un seul niveau d'abstraction.

Le mécanisme principal de la dérivation est la dérivation des types de données. Sur chacun de ces nouveaux types, il faut définir les fonctions et opérateurs nécessaires à leur manipulation. Il est également possible d'exporter des constructions du langage de départ.

Comme nous l'avons déjà mentionné, cette dérivation n'est pas une extension. Ainsi, un langage dérivé ne possédera pas par défaut tous les types, opérateurs et fonctions du langage de départ, mais

uniquement ceux qui ont été exportés.

3. La révolution VHDL

Tandis que CONLAN partait sur le principe d'un langage par niveau d'abstraction, VHDL est un langage qui permet de les traiter tous à la fois. Comme on peut le lire dans [Bor93a], cette approche est considérée par certains comme une erreur, et conduit à une dégradation de la manière de décrire (descriptiveness) et des performances des outils qui l'utilisent.

Il est évident que VHDL est un langage très complexe, les compilateurs le sont aussi, mais les progrès réalisés dans ce domaine ont été importants ces dernières années. De toute façon, comme c'est le cas en synthèse, et pour certains simulateurs dédiés, rien n'oblige un fabricant à accepter tout VHDL, mais il peut se limiter à un sous-ensemble. C'est ainsi, que l'on voit apparaître des simulateurs qui permettent de simuler uniquement les circuits synchrones, et dont les performances sont largement supérieures aux simulateurs VHDL-complet dans ce domaine.

Par contre, au niveau de la puissance de description, on peut dire que VHDL est vraiment supérieur aux autres. Il permet la description du niveau système au niveau portes logiques, avec même des essais de simulation au niveau timing avec des algorithmes spécifiques.

Par contre le gros défaut de ce langage est d'avoir été spécifié pour la simulation. En effet, la sémantique définie dans le manuel de référence est une sémantique de simulation. C'est surtout cela qui gêne le plus les chercheurs qui travaillent dans la preuve formelle et la synthèse.

4. Quelques autres HDLs numériques

Dans cette partie, nous nous intéresserons uniquement aux principaux langages utilisés dans l'industrie. Actuellement il semble que le marché se limite à VHDL, M et Verilog. D'ailleurs ce dernier vient d'être normalisé, et est donc devenu du domaine public. Quant aux quelques autres, ils semblent en voie d'abandon pour VHDL.

Donc, en plus de ces deux autres HDLs, nous nous intéresserons à UDL/I, langage standard développé par le Japon, qui semble intéressant car il contient dans son manuel de référence une

sémantique de synthèse. De plus, il se rapproche de VHDL par le concept de langage standard non attaché à un outil, c'est d'ailleurs un de leurs rares points communs.

Les paragraphes suivants, ont été écrits grâce à [Ber93], dans lequel on trouvera d'intéressantes comparaisons entre ces trois langages et VHDL.

a. UDL/I (Unified Design Language for Integrated circuits)

UDL/I est un langage normalisé, standard et public, qui est sponsorisé par l'industrie japonaise. La phase de normalisation vient de prendre fin, et donc son exploitation va commencer. Certains pensent que ce langage est mort-né face à VHDL ou Verilog. Cependant, la société Fintronics aux Etats Unis commercialise des outils qui utilisent ce langage en entrée. La principale qualité de ce langage, est d'avoir été développé en vue d'utilisation en synthèse, le manuel de référence spécifie d'ailleurs une sémantique pour la synthèse. Ceci l'oppose à VHDL ou à Verilog, qui sont des langages de simulation. Ainsi dans UDL/I chaque primitive représente une réalité matérielle. Cet aspect en fait un langage plus adapté à la description de circuit intégré plutôt qu'à la description système de haut niveau.

L'unité de base de UDL/I est la **design description**. Celle-ci contient plusieurs sous-parties, dont une d'identification ou sont référencées des informations sur l'auteur de la descripton, la date, ce que la description représente, la technologie utilisée pour fabriquer ce circuit ou encore ce à quoi elle va servir : simulation, preuve formelle.

Les connexions peuvent être déclarées, comme des horloges : **clock**, des références électriques : **powers**, des **resets**, des connexions externes :**ext**, **in**, **out** ou **inout**. Ces trois dernières sont bien connues, le mode **powers** sert pour les outils de routage, les **ext** pour utiliser des composants d'un autre langage et les modes **clock** et **reset** n'ont pas de sémantique associée dans le manuel de référence.

La grosse lacune de UDL/I est de ne pas pouvoir paramétrer les **design description**.

UDL/I autorise la description structurelle, comportementale ou par table de vérité, mais dans une même **design description** il n'est possible d'utiliser qu'un style parmi ces trois.

La description structurelle se fait très classiquement en référençant directement d'autres **design**

description. Ensuite, on attache une connexion d'une de celles-ci à une connexion ou un ensemble de connexions d'une ou plusieurs autres. On peut optionnellement indiquer la direction de la propagation des valeurs. Cet attachement est fait au moyen d'une instruction qui ressemble à une affectation, dont l'identificateur de gauche est le nom du **net**.

La description comportementale se fait au moyen d'instructions concurrentes et, pour ce qui est de la description algorithmique séquentielle, il est possible de définir des automates. Ces derniers modélisent du code séquentiel, mais ils sont composés d'états dans lesquels se trouvent des instructions concurrentes. En fait, il n'existe pas d'instructions séquentielles en UDL/I.

Tous les objets UDL/I ont le même type : le type bit quatre états (X, 0, 1, Z). La gestion des conflits se fait au moyen de différentes fonctions de résolution pré-définies.

Il est possible de faire de la rétro-annotation en UDL/I. Chaque unité possède une partie delay dans laquelle on peut spécifier la charge nominale et maximale que supporte une sortie et où on peut aussi calculer le temps de propagation entre les entrées et les sorties, ou encore le temps d'établissement des sorties en fonction de la charge qui suit.

Enfin, il existe une sémantique non-déterministe dans l'affectation simultanée des registres. Ce choix a été fait par rapport au comportement des circuits qui est quelquefois non-déterministe. Mais l'interprétation des résultats est alors difficile.

Le principal point fort de UDL/I est de proposer une sémantique pour la synthèse. Certains diront qu'il s'adresse donc plus à des électroniciens, considérant souvent les HDLs en particulier VHDL comme trop informatique. En effet, chaque instruction de UDL/I possède une représentation physique. Enfin, un avantage pour certains et un inconvénient pour d'autres, UDL/I est un langage simple.

b. Verilog

Ce langage est le HDL utilisé par le simulateur numérique de Cadence. Actuellement, c'est le langage le plus utilisé dans le monde pour la simulation numérique. Cette suprématie est très forte aux Etats-Unis, un peu moins en Europe ou l'utilisation de VHDL connaît un bel essor.

Il a pour principal avantage de posséder d'énormes bibliothèques de composants. Sa lutte avec VHDL s'est terminée par un match nul, il semble que l'on se tourne vers des simulateurs mixtes VHDL-Verilog.

L'unité de base de Verilog est un **module**. Ce **module** peut lui même en référencer d'autres et les interconnecter via des **wires**, on fait alors de la description structurelle. De plus, il peut contenir des parties algorithmiques éxécutées séquentiellement : les **always** et les **initials**. Les **always** sont des boucles sans fin alors que les **initials** ne s'exécutent qu'une seule fois. Ces parties séquentielles comportent des instructions de synchronisation sur des changements d'état d'objets pendant un certain temps, ou bien sur une condition booléenne plus complexe. Pour les **always** la présence d'une instruction de synchronisation est obligatoire, sous peine de bloquer le simulateur avec une boucle sans fin.

A l'intérieur du programme séquentiel on peut donner une liste d'instructions concurrentes. Il s'agit des instructions **fork ... joined**. Il est possible d'écrire des fonctions, celles-ci restent locales à un module. Il est également possible de les référencer d'un autre module, mais il n'existe pas de gestion de paquetages comme en VHDL. D'ailleurs il n'y a pas à proprement parler de gestion de bibliothèque, une description simulable se trouve dans un fichier. Ceci alourdit le cycle de conception.

Comme en CONLAN ou VHDL, il y a une dualité du temps : temps physique et delais infinitésimaux pour les propagations à délai nul.

Verilog n'autorise pas les types utilisateurs. Pour les objets relatifs au matériel, seul un type bit 4 états (X, 0, 1, Z) existe. Pour les variables manipulées dans les **always** ou **initial**, il existe les types réel, entier ou time.

Les connexions des modules ont donc un type fixé, mais elles possèdent d'autres propriétés, en particulier la direction de propagation : **in, inout** ou **out**. De plus, il est possible de choisir entre **net** et **register**. Cela permet de faire une distinction entre connexion électrique et élément de mémoire. D'ailleurs, c'est sur les **nets** que sont résolus les conflits, et l'utilisateur peut choisir le comportement de cette résolution au moyen de la classe de net qu'il va utiliser : **wire, wand, wor,** ....

La possibilité la plus intéressante de Verilog est la description du comportement sous forme de table de vérité. En effet, il existe l'état 'Don't care' qui représente n'importe lequel des quatre autres, et l'état 'Bit value' qui représente les deux valeurs explicites 0 ou 1. Ces deux états rendent l'écriture d'une table concise et rapide.

Il existe un ensemble complet d'opérateurs sur les bits, booléens et arithmétiques. De la même manière, on trouve un ensemble prédéfini de portes logiques standards.

Le point fort du simulateur Verilog, est d'être beaucoup plus rapide que les autres. Cela s'explique :

- par le manque de type utilisateur, la manipulation des objets a pu être ainsi optimisée,
- par l'écriture des tables qui se prête bien au codage machine,
- par son ensemble de primitives pré-définies.

Toutefois, au niveau du langage, il est beaucoup moins complet que VHDL ou CONLAN. Mais, c'est cette différence qui rend la simulation plus rapide.

d. M

M est le langage du simulateur Lsim. Il présente la même architecture que le C, duquel il se rapproche beaucoup. Ainsi la gestion des descriptions se fait comme en C avec des fichiers '.h', au moyen d'inclusion.

L'unité principale en M est le **module**. Il est composé de quatre parties :

- une partie déclarative,
- une partie pour la description structurelle,
- une partie pour l'initialisation,
- une partie de code séquentiel pour la description comportementale.

Un **module** peut recevoir des paramètres comme par exemple la taille du vecteur de bit qu'il manipule. La description structurelle se fait en référençant directement d'autres **modules** et en les connectant les uns aux autres au moyen de la fonction **net**.

L'interface d'un module est composé de **terminal** qui peuvent être d'un type utilisateur, logique,

ou bien d'un type spécial pour faire de la simulation "timing" avec le simulateur ADEPT. Ce n'est pas à proprement parler de la simulation en mode mixte puisque ce simulateur est géré par évènement. Mais M intègre des primitives pour ce genre de description et permet donc une descripton rudimentaire de systèmes analogiques. Les types utilisateurs sont les mêmes que ceux définissables en langage C.

La gestion de la concurrence se fait au moyen de co-processus dans la partie de code séquentiel. Ces co-processus sont de vrais processus Unix, qui communiquent au moyen de variables. Les conflits engendrés par de telles descriptions sont résolus par le système de manière plus ou moins prévisible, d'autant plus que ces conflits ne sont pas détectables à la simulation.

Il est également possible de donner une liste d'instructions à exécuter en parallèle au moyen de l'instruction **fork...joined.**

La bibliothèque de fonctions pré-définies est conséquente, comme celle de portes de bases. D'autre part, M intègre des fonctions pour faire de la rétro-annotation. Ces fonctions servent pour évaluer les temps d'établissement d'une valeur en fonction de la charge qui est branchée derrière.

III. La description analogique

1. L'état actuel

Actuellement, il existe un standard de fait pour la description analogique structurelle : il s'agit du langage défini pour le simulateur Spice [Nag75]. Par extension on le nomme langage Spice.

Il y a bien eu des essais de langages comportementaux, mais ceux-ci sont restés d'une utilisation très marginale. Ainsi, aucun standard industriel n'a émergé durant ces dernières années. Cependant, depuis 1987, la société américaine Analogy propose un tel langage : MAST [Get87, Man92], et en France la société ANACAD avec le CNET le langage FAS [Ana91]. Ils n'ont connu ni l'un ni l'autre l'essor que l'on aurait pu attendre. En effet, ils utilisent des nouveaux concepts qui effraient un peu les concepteurs de circuits intégrés. De plus, pour le cas du langage MAST on peut mettre en avant la faiblesse du simulateur qui lui est associé.

Pourtant, de plus en plus de sociétés essaient de définir leur propre langage : Diablo pour Dazix ou encore Cadence qui va en commercialiser un à la fin de l'année.

Heureusement, depuis un an un groupe de normalisation a vu le jour pour proposer une extension analogique au langage standard VHDL. Dans le chapitre suivant, nous verrons où en est cette normalisation, et comment ce travail s'y intègre. De la même manière, un autre groupe de travail plus ancien a défini les spécifications d'un langage de description analogique orienté micro-onde : MHDL [MHD93].

Dans les sous-paragraphes suivants, nous regarderons différents langages analogiques. On passera rapidement en revue quels sont leur principaux avantages ainsi que leurs limitations.

2. Les besoins de la simulation analogique

Comme on l'a déjà dit auparavant, l'outil le plus utilisé en CAO analogique reste le simulateur : la majorité des descriptions lui est destinée . Il est donc intéressant de voir les principes de bases de la simulation analogique [Ped84].

La première étape est la mise en équation du système. On obtient alors un système d'équations algébro-différentielles non-linéaires à plusieurs inconnues. Pour pouvoir le résoudre par voie informatique, il faut transformer les équations différentielles en équations algébriques au moyen de méthodes d'intégration numériques. On possède alors un système d'équations algébriques non-linéaires qui va être résolu au moyen de l'algorithme de Newton-Raphson par exemple.

Ce sont ces trois points qui vont être présentés dans les paragraphes suivants, on trouvera plus de détails sur chacun et sur la simulation analogique dans [Vac93], surtout au sujet des nouveaux algorithmes tels que ceux basés sur la relaxation.

a. La mise en équation

Plusieurs méthodes existent, actuellement la plus utilisée est la **méthode nodale modifiée**. Cette méthode apparaît donc comme la moins problématique, et la plus efficace. On citera néanmoins la méthode nodale et la méthode du tableau.

La méthode nodale prend pour inconnues les tensions des noeuds et comme équations la somme des courants en un noeud qui est égale à zéro (loi des noeuds de Kirchoff). Cette méthode ne permet de prendre en compte les sources de tensions qu'au moyen d'un pré-traitement matriciel lourd. En effet, pour ces composants on connaît explicitement la différence de potentiel à leurs bornes, mais on est incapable d'exprimer le courant qui les traverse en fonction de ces potentiels.

La méthode nodale modifiée est dérivée de la méthode nodale, et elle permet de prendre facilement en compte les sources de tensions. Pour chacune de celles-ci, on rajoute comme inconnue le courant qui la traverse et comme équation la différence de potentiel à respecter.

La méthode du tableau prend pour inconnues les courants de branches, les tensions des noeuds et les états internes des différents systèmes; elle prend pour équations les lois des noeuds et des mailles et les lois de fonctionnement des composants. Cette méthode conduit à la construction d'un système très important, mais la matrice correspondante est très creuse. Les simulateurs utilisant cette méthode possèdent de bons algorithmes de manipulation de matrice creuse.

b. Les méthodes numériques de calculs des dérivées

La résolution numérique remplace l'intervalle continu du temps de la simulation en un intervalle discret composé d'un certain nombre de points. Selon les algorithmes utilisés l'intervalle de temps entre deux points sera constant ou variable. Cet intervalle est appelé le pas de calcul du simulateur.

A un instant donné de la simulation, le simulateur possède pour chaque variable une connaissance de son passé sous forme d'une série de valeurs à des instants précis. Les dérivées de ces inconnues sont évaluées au moyen de formules linéaires à plusieurs pas. On calcule l'ordre de ces formules en fonction de l'ordre des polynômes pour lesquelles elles donnent un résultat exact.

Les plus simples et les plus efficaces sont Forward-Euler, Backward-Euler et la méthode des trapèzes.

c. Les algorithmes de résolution

i. Newton-Raphson

L'algorithme de Newton-Raphson est une généralisation de l'algorithme de Newton à l'ordre n. L'algorithme de Newton permet de résoudre une équation à une inconnue, au moyen d'une suite itérative. Cette méthode converge sur des intervalles strictement monotones. La formule fait intervenir la dérivée de l'équation par rapport à l'inconnue, à l'ordre n la dérivée se transforme en matrice jacobienne. C'est cette matrice que le simulateur doit inverser.

Cet algorithme s'est vu largement dépassé par la taille sans cesse croissante des circuits intégrés, cette taille approche plusieurs millions de transistors par puce : il a donc fallu trouver de nouveaux algorithmes. C'est là que sont apparues les méthodes dites de relaxation [Whi87]. Ces algorithmes utilisent la latence du circuit. On peut citer par exemple WaveForm-Relaxation (WFR) [Lel82], Iterated Timing Analysis (ITA) [Sal87] et encore One-Step-Relaxation (OSR) [Hen85]. Les simulateurs qui utilisent ces algorithmes, sont appelés simulateurs de troisième génération.

ii. La relaxation

Pour de tels algorithmes, le circuit est partitioné en sous-circuits dont le couplage entre eux est faible. Chaque sous-circuit possède des variables internes qui doivent être résolues, et des variables externes qui représentent les connexions avec d'autres sous-circuits. Les variables internes sont calculées pour un temps donné, si l'activité du sous-circuit est suffisante, ou bien si une des variables externes a été modifiée. L'activité du circuit est un critère qui se mesure avec les dérivées des différentes variables internes. Si celles-ci ont des valeurs faibles, il est au repos, sinon il est en train de travailler.

On distingue deux catégories de décomposition :

- une seule équation par sous-circuit (pointwise),
- plusieurs équations par sous-circuit (blockwise).

Les méthodes de relaxation sont des méthodes itératives de résolution. Cependant, dans la boucle de résolution, il faut résoudre un ensemble d'équations et cela se fait grâce à l'algorithme de

Newton-Raphson par exemple.

Ces algorithmes de relaxation sont particulièrement adaptés à la simulation de circuits numériques. En effet dans de tels circuits il existe réellement des sous-circuits, et ceux-ci travaillent rarement en parallèle, et donc sur l'ensemble du circuit à un instant donné, seule une partie effectue des opérations. D'autre part, les transistors MOS sont les mieux adaptés pour ces algorithmes. En effet, ces transistors possèdent un fort découplage entre la grille et les deux autres connexions : le drain et la source. C'est pour ce cas précis que les simulateurs utilisant une décomposition *pointwise* donnent leurs meilleurs résultats.

D'un autre coté, pour les gros circuits analogiques pour lesquels le couplage est fort entre les différents noeuds du circuit, et encore plus pour les circuits à base de transistors bipolaires, les résultats ne sont pas bons, surtout pour les simulateurs utilisants une décomposition *pointwise*.

d. Au niveau de la description

Pour ce qui est de la description, la grande majorité des simulateurs utilisent comme langage le standard Spice. Ils connaissent donc le comportement d'un ensemble de composants de base. Ce comportement est décrit en interne en langage de programmation C ou Fortran.

Nous nous intéresserons à la description pour un simulateur utilisant la méthode nodale modifiée pour sa mise en équation, et l'algorithme de Newton-Raphson pour la résolution, couplé ou non à un algorithme de relaxation. Ceci représente la majorité des simulateurs du marché.

Pour de tels simulateurs, les descriptions sont des procédures qui permettent de calculer le courant qui entre ou qui sort de chaque branche du composant en fonction des tensions. Pour être plus général, il s'agit de procédures qui permettent de calculer les termes des équations en fonction des inconnues. Ces procédures sont utilisées pour vérifier l'ensemble des équations à résoudre : les sommes des courants de branches qui doivent être nulles et les différences de potentiel des sources de tensions.

D'autre part, il leur faut calculer la matrice jacobienne, qui est composée des dérivées des différentes équations par rapport aux inconnues. Comme les équations sont composées de sommes de

courants et que les inconnues sont les tensions, il faut donc calculer les dérivées des courants par rapport aux tensions.

Le calcul de ces dérivées se fait soit par voie numérique, soit par l'utilisation de formules explicites pour les cas simples. Ainsi, pour une résistance, la contribution au jacobien vaut ±1/R. Par voie numérique, pour évaluer la dérivée d'une branche de courant par rapport à une tension, on calcule son taux d'accroissement pour deux valeurs très proches de la tension. On peut évidemment utiliser une formule plus complexe, mais la pratique montre que le rapport vitesse/précision est meilleur pour la plus simple.

En conclusion, la description peut contenir deux types d'information : une facultative et une obligatoire. L'information obligatoire est la formule qui permet de calculer les courants qui sortent des connexions du composant, et l'information facultative la formule de la dérivée de ces courants par rapport aux tensions dont ils dépendent.

Ainsi, pour les modèles simples, il peut y avoir des formules permettant de calculer les valeurs des dérivées des courants par rapport aux tensions en vue du calcul du jacobien. Dans tous les cas, la description est composée de formules qui permettent de calculer le courant en fonction des tensions. Dans le cas où une telle formule est difficile à extraire de l'équation dont le courant est solution (équation transcendantale), on rajoute ce courant comme nouvelle inconnue et cette équation à l'ensemble à résoudre.

Enfin, chaque description possède trois vues différentes, une pour le comportement en courant continu, une pour le mode transitoire et une pour l'analyse fréquentielle. Les deux premiers peuvent être identiques, mais le troisième nécessite des calculs en nombre complexe.

3. Le langage Spice

Spice permet de faire de la description structurelle à partir d'un ensemble de composants de base dont le comportement est inclus directement dans le simulateur. La syntaxe est vraiment très simple : la ou les premières lettres de la ligne identifient le type de composant: R pour une résistance ou encore C pour un condensateur. Enfin, chaque composant comporte un certain nombre de

paramètres, optionnels ou non, que l'utilisateur doit fournir.

```
R --> résistance
C --> condensateur
L --> inductance
Q --> transistor bipolaire
M -->      "     mos
V --> source de tension
I --> source de courant
```

On peut néanmoins introduire une hiérarchie en définissant des sous-circuits qui pourront eux même être utilisés dans d'autres sous-circuits. De plus, dans l'extension Spice du simulateur Eldo, ces sous-circuits peuvent contenir des paramètres qui seront donnés par l'utilisateur lors de leur instanciation.

En conclusion, ce langage a pour principaux avantages sa simplicité d'accès et surtout la rapidité avec laquelle on peut décrire un circuit très simple. Par contre, pour des gros circuits, la description devient vite lourde, et la réutilisation d'une description pour une autre plus complexe n'est pas immédiate.

Du point de vue informatique, la description est analysée sans génération de code intermédiaire, on passe directement aux structures de données du simulateur. Donc, on ne peut pas faire de compilation séparée. Les bibliothèques que certains proposent sont en fait des morceaux de description (sous-circuit) qui sont ajoutés à la description.

4. Fas

Ce langage a été conjointement développé par le CNET et la société ANACAD. Il est intégré au simulateur électrique Eldo [Hen85] en plus de son entrée Spice. Fas est une extension analogique du HDL numérique Fidel [ElT84], malheureusement aucun couplage n'existe entre les deux langages.

Le langage Fas a été développé pour répondre à la demande de certains concepteurs qui voulaient pouvoir décrire eux-mêmes le comportement de leur composant. Initialement, il a été développé une entrée en langage C directement dans le code source du simulateur. L'utilisateur avait alors accès à un certain nombre de fonctions pour la déclaration et l'utilisation des objets. Cette option existe toujours dans Eldo, elle s'appelle C-Fas. Elle présente les inconvénients de devoir fournir le code

objet du simulateur et de proposer aux utilisateurs une syntaxe peu conviviale. C'est pourquoi, il a été décidé d'étendre le langage Fidel pour permettre de décrire le comportement des composants analogiques.

La description en Fas suit le formalisme de description propre à la méthode nodale modifiée. Ainsi, une description en Fas est une procédure qui permet dans le cas général de calculer le courant de chacune de ses connexions en fonction des tensions de ces dernières. Il est également possible de calculer la tension d'une connexion, il faut alors le déclarer au simulateur en changeant le statut de la connexion pour indiquer qu'il s'agit d'une source de tension. Il est également possible de rajouter des inconnues et donc aussi des équations. Par contre, il n'a pas été envisagé de pouvoir donner directement les formules pour calculer les termes du jacobien.

D'autre part, une fonction permet de tester dans quel mode se trouve le simulateur, et ainsi de l'aiguiller vers la description propre au transitoire, au continu ou au fréquentiel.

Les modèles Fas sont compilés, le compilateur génère un pseudo-code qui sera interprété durant la simulation. Ce code est stocké dans une bibliothèque et utilisé par Eldo si le modèle est instancié dans le circuit. L'inconvénient de cette solution est une relative lenteur de la simulation. A terme, il a été envisagé de générer directement du langage C. Cette solution permettra de gagner un facteur cinq.

Bien entendu, la rapidité de simulation dépendra de la qualité de la description. Ainsi, un calcul explicite du courant sera plus rapide à simuler qu'une équation implicite, solution qui présente l'inconvénient de rajouter une équation et une inconnue.

D'autre part, les modèles peuvent être utilisés en analyse fréquentielle dans certains cas. En effet, les calculs de la description sont faits en nombres complexes, et chaque fonction doit avoir son équivalent dans ce mode, sinon l'évaluation ne peut avoir lieu. Ainsi, si l'utilisateur a utilisé, dans sa description, la fonction qui renvoie la valeur du pas de calcul courant ou s'il a référencé des valeurs précédentes d'états ou de tensions de connexion, la simulation fréquentielle ne peut pas avoir lieu. En effet, ces deux fonctions n'ont pas de représentation fréquentielle.

Par contre, on ne peut pas faire une description dédiée à l'analyse fréquentielle en utilisant

directement la fréquence dans les formules.

Cependant, grâce à une variable système, on peut séparer la description en trois parties : une pour l'analyse continue, une autre pour la transitoire et enfin une dernière pour la fréquentielle.

Le langage Fas permet de faire des descriptions assez précises, ainsi une partie des modèles de base d'Eldo a été écrite en Fas.

Cependant, la syntaxe n'est pas parfaite et son utilisation n'est pas très souple. C'est pour cette raison, qu'un effort interne a été fait pour redéfinir Fas à partir des travaux de normalisation de VHDL-analogique. Le but d'une telle réalisation était d'une part d'améliorer les performances du langage, et d'autre part, de commencer à intégrer les nouveaux concepts de VHDL. Cette réalisation peut être vue comme un pas vers un futur VHDL-analogique.

Les apports de VHDL pour le langage Fas sont la séparation des vues externes (entité) et internes (architecture) d'un système, une définition plus claire des différents objets et concepts du langage et enfin une meilleure lisibilité. Par exemple, dans Fas les déclarations d'équations à résoudre sont faites dans la même partie que le code procédural. Avec le nouveau Fas, il existe deux parties distinctes : une pour les déclarations d'équations et une pour la partie procédurale.

Exemple Fas : un circuit RLC

La description

```
amodel rlc (p, n);  (* vue externe du composant *)

     (* type des connexions *)
     declare pin p, n : electrical;

     (* paramètres du système *)
     declare param r, l, c : real;

     (* variable intermédiaire, pour calculer
        le courant traversant la résistance    *)
     declare variable ir : real;

     (* état du système : le courant de la self inductance
        il s'agit d'un état implicite
        il est considéré comme inconnue
        il faut donner une équation dont il est solution   *)
     declare istate il : real;

analog
     (* application de la loi d'Ohm
```

```
        volt.diff(p,n) renvoie la différence
        de potentiel entre les pins p et n    *)
    make ir = volt.diff(p,n) / r

    (* équation dont il est inconnue
        state.integ référence l'intégrale par rapport au temps
        state.dt référence la dérivée temporelle  *)
    solve(il, state.integ(il)/c + l*state.dt(il)
            - volt.diff(p,n))

    (* on donne le courant sortant de p et de n *)
    make curr.on(p) = -ir - il;
    make curr.on(n) = ir + il;
endanalog
endmodel
```

Pour cette description, on a conservé l'équation du second ordre, et elle est donnée à résoudre directement au simulateur avec la fonction solve. De plus, on a déclaré il comme inconnue : c'est un **istate** (implicite state).

La simulation

La simulation d'un modèle Fas se fait en utilisant le simulateur Eldo, il faut donc réaliser une description pour ce simulateur. On utilise alors le langage structurel Spice.

5. MAST® (Modeling Analog System with Template)

MAST est un langage de description analogique qui couvre les domaines structurel et comportemental. Il permet de couvrir de nombreux champs d'application dont le domaine électrique, thermique ou encore mécanique.

L'élément de base de la description est le **template**, qui représente un composant. La description se fait en trois parties : une d'initialisation, une de calcul des valeurs intermédiaires et une où l'on donne les relations à vérifier entre les divers objets.

Les connexions possèdent deux champs : un représentant le courant (through) et un la tension (across). On retrouve dans ce langage tout ce qui est nécessaire pour décrire le comportement analogique : opérateur de dérivation et d'intégration, toutes les fonctions mathématiques, accès au pas de calcul du simulateur et encore date ou fréquence actuelle de la simulation. Une autre particularité intéressante de MAST est la possibilité de gérer une table d'évènements pour forcer le

simulateur à effectuer un pas de calcul à une date précise pour modifier le comportement d'un élément. Dans le même sens, il existe des primitives pour décrire rapidement des éléments logiques.

Le gros avantage de MAST est de posséder une bibliothèque de plusieurs milliers de modèles. D'autre part, Saber est couplé avec le simulateur numérique Cadat pour faire de la véritable simulation mixte.

6. Demain

Actuellement, il semble que la demande soit forte pour avoir accès à un langage de description analogique pour le domaine comportemental. La normalisation de VHDL-analogique attire de plus en plus de monde. D'ailleurs, pour accélérer le processus IEEE, un groupe d'industriels a lancé une normalisation parallèle pour sortir un standard rapidement. Bien entendu, le langage sera basé sur les travaux de la normalisation VHDL-analogique. Cette initiative s'appelle AVI.

IV Les langages mixtes

1. Les approches non-intégrées

La réalisation d'un simulateur mixte se fait souvent par la réunion de deux simulateurs existants. Chacun de ces simulateurs utilise un langage de description différent pour décrire les circuits qu'ils sont capables de simuler. Comme il serait coûteux de développer un nouvel analyseur sur un langage unique, la réalisation se contente de mélanger les deux descriptions.

Une première solution consiste à décrire dans chaque langage la partie de circuit correspondante. Ensuite, au moyen de constructeurs de chaque langage, il faut déclarer les noeuds sur lesquels vont avoir lieu les échanges d'information. Ces constructeurs peuvent être déjà présents dans le langage, ou bien spécialement définis pour cette réalisation. Ceci peut être gênant pour un langage standardisé. L'étape suivante consiste à lier chaque noeud mixte à son homologue de l'autre niveau. Cela peut se faire soit en utilisant un fichier neutre, soit dans l'un des langages, soit dans les deux. Cette solution est celle choisie par diverses sociétés qui veulent réaliser des "backplanes". Il s'agit

d'un noyau capable de faire communiquer plusieurs simulateurs aussi bien numériques qu'analogiques. Le groupe de travail relatif à ces travaux est le groupe CFI.

La seconde solution consiste à décrire le système dans son ensemble de manière structurelle dans un des deux langages. Certaines feuilles de la description seront en fait des modules de l'autre langage. Il faudra, pour chacune de ces feuilles référencer un fichier dans lequel se trouvera leur description. Cette solution est utilisée dans Fideldo et dans VHD$_e$LDO. Dans le premier cas c'est dans la description Spice que se trouve toute l'information structurelle, alors que dans le second c'est dans la description VHDL.

Ainsi, chaque partie du système sera selon le niveau d'abstraction décrit en utilisant l'un ou l'autre langage. Ces solutions sont donc des artifices permettant d'utiliser un outil. En aucun cas la description réalisée ne pourra être utilisée pour des spécifications ou pour un autre outil. Cependant, étant donné qu'actuellement il n'existe pas réellement de standard industriel de langage de description permettant de couvrir ces niveaux d'abstractions, on est obligé de passer par de telles solutions.

D'ailleurs, ceci est peut être la cause de la non utilisation de simulateur multi-mode. En effet, les premiers simulateurs de ce genre sont apparus il y prés de quinze ans [Hil79], et actuellement aucun produit industriel n'est réellement utilisé à grande échelle. Cependant, la demande reste forte.

2. Cascade

Cascade est un projet qui a été mené par le laboratoire Artemis au début des années quatre-vingt [Bor93b]. Son but était de proposer un simulateur couvrant tous les niveaux de description ainsi que les domaines continu et discret. L'autre point fort de ce projet était l'utilisation des principes CONLAN. Ainsi, les langages utilisés pour le simulateur étaient tous basés sur cette approche. Comme nous l'avons vu, CONLAN est un langage qui s'adresse aussi bien aux concepteurs qu'aux développeurs. En effet, CONLAN possède des primitives de dérivation pour définir un langage à partir d'un langage père. Par contre, Cascade n'intègre pas ces mécanismes et n'est destiné qu'aux concepteurs et utilisateurs d'outils de CAO. Par exemple, on ne peut pas définir un nouveau type, mais uniquement des sous-types de types existants.

On peut ainsi dire que Cascade fut le premier langage unique pour décrire des systèmes du niveau électrique au niveau système, sur le domaine continu ou discret.

Cascade intégrait les sous-langages suivant :

- IMAG pour le niveau électrique,
- CASTOR pour le niveau interrupteur,
- POLO pour le niveau portes logiques,
- CASSANDRE pour le niveau transfert de registres,
- LASCAR pour le niveau fonctionnel,
- LASSO pour le niveau système.

Une unité Cascade référence avant tout un langage. Pour chacun des langages, exceptés POLO et CASTOR, il est possible de mélanger domaine structurel et comportemental. Ceci est une grande différence avec les langages de l'époque dans lesquels un seul domaine était supporté.

Le problème de la description multi-niveaux est résolu par la modularité, c'est à dire qu'un niveau donné peut englober des feuilles d'autres niveaux y compris le sien. Les informations sont échangées au travers de structures communes entre les deux langages. Le tableau ci-dessous, emprunté au manuel d'introduction, montre les structures communes qui permettent l'échange et indique quels langages acceptent des feuilles d'un niveau différent. Ainsi, verticalement, on trouve les boîtes englobantes et horizontalement les boîtes englobées.

	Imag_F	Imag_D	Castor	Polo	Lascar	Lasso
Imag_D	variable					
Polo	variable	variable	plot	logic variable	signal, register	
Lascar	variable	variable	plot	logic variable	signal, register, latch	signal, clock
Lasso				logic variable	clock, latch, signal	control, variable, clock

V. Conclusion

Ce chapitre a permis de présenter ce que sont les langages de description de matériels et leur utilisation. D'autre part, nous avons vu les notions importantes en description que sont les niveaux et les domaines. Actuellement, ces langages sont dédiés soit au monde discret (numérique) soit au monde continu (analogique). En effet, mis à part le projet Cascade qui est malheureusement resté universitaire, il n'y a pas eu de développement d'un HDL permettant de mélanger ces deux mondes, bien que des simulateurs mixtes aient existé. A chaque fois, ces simulateurs ont mixé de manière plus ou moins simple deux langages différents. Il semble d'ailleurs que le non développement de ces outils soit justement dû à la difficulté de la description. Donc, la nécessité d'un langage standard pour la description mixte est importante. Voila pourquoi VHDL-Analogique doit être réalisé.

Chapitre II.
VHDL

I. Rappel sur la normalisation de VHDL

Lors de la dernière normalisation de VHDL, qui s'est terminée fin 1992, des utilisateurs ont émis des *requirements* (requêtes, revendications) afin que le langage permette la description analogique. On pourra trouver par exemple [Bou91, LeF91, ElT91, ElT92, Rou91, Rod93a&b]. Devant ces propositions, le groupe chargé de la normalisation (VASG) a décidé de ne pas les prendre en compte pour la présente normalisation, le temps restant étant jugé trop court. Cependant, il a été décidé de créer un sous-standard (Sub-Par 1076.1) qui suivra une normalisation équivalente à VHDL. Cette normalisation devrait aboutir courant 1994. Enfin, les deux langages seraient regroupés lors de la prochaine normalisation de VHDL, c'est à dire pour 1997.

La notion de sous-standard a dû être rigoureusement définie : il fallait savoir en effet si celui-ci inclurait l'actuel VHDL ou s'il ne contiendrait que l'extension analogique. Pour des raisons de cohérence, il semble normal de créer un langage mixte unique, c'est à dire que le sous-standard inclura VHDL, à la norme 1992 bien entendu, et non à celle de 1987.

Dans ce qui suit, plutôt que de rester dans le cas général de la modélisation physique, on va se ramener au cas particulier du monde électrique. Ainsi, les deux grandeurs intensives et extensives seront respectivement le courant et la tension.

De plus, on va présenter le langage grâce aux différentes propositions faites durant la normalisation et aussi d'après des réflexions internes (CNET, Anacad, Université Joseph Fourier et Ecole Polytechnique Fédérale de Lausanne). Le langage ainsi proposé permet de modéliser de nombreuses choses. Néanmoins tous les problèmes sont loin d'être résolus et une étude plus approfondie est nécessaire, surtout pour assurer la cohérence du langage.

II. Présentation de VHDL

Le but de ce paragraphe est de présenter les principales caractéristiques de VHDL'87, surtout celles qui sont utiles à la compréhension de l'extension de VHDL vers l'analogique.

1. Notions Générales et les concepts de VHDL

a. Le langage

VHDL est un langage standardisé par l'IEEE, son statut indique qu'il doit être re-standardisé au moins tous les cinq ans. On est actuellement à la fin de la première standardisation. C'est pour cela que l'on parle de VHDL'87, ancienne norme et VHDL'92, nouvelle norme. Le langage est spécifié dans un document appelé manuel de référence, en anglais Language Reference Manual (LRM) [VHD87]. Il faut souligner que VHDL a hérité de l'expérience du langage Ada, autant sur le point syntaxique que sur le point de la gestion de la bibliothèque.

b. La gestion de bibliothèque

VHDL, tout comme Ada, ignore la notion de fichier, on parle d'unités. Ces unités sont analysées et placées dans la bibliothèque de travail courante. Chaque unité pourra référencer tout ou une partie de sa bibliothèque ou d'une autre. Il est à noter que pour une unité sa bibliothèque est toujours celle de travail.

c. La hiérarchie : les blocs, les processus et les signaux

i. Le bloc

Comme la majorité des HDLs numériques, une description VHDL est une description hiérarchique. L'objet VHDL qui permet de représenter cette hiérarchie est le bloc (**block**). Un bloc possède une interface qui permet de le relier au bloc qui le contient. L'interface du bloc de plus haut niveau est vide. Un bloc contient aussi un ensemble de déclarations. Enfin, le corps du bloc contient un ensemble d'instructions concurrentes. Le bloc est lui même une instruction concurrente de VHDL. On la présentera plus en détail plus loin.

ii. Le processus

Le second objet important de cette hiérarchie est le processus (**process**). Un processus est la feuille terminale de la description hiérarchique. Ce processus contient du code algorithmique qui représente une partie du comportement du système modélisé. Tous les processus sont concurrents les uns par rapport aux autres.

iii. Le signal

Enfin, le dernier objet important de la hiérarchie VHDL est le signal (**signal**). Le signal est une équipotentielle qui représente un ou plusieurs fils dans le monde physique. C'est au moyen de ce signal que les processus communiquent, et que les blocs sont connectés les uns aux autres. Un signal contient une valeur courante et son historique. De plus, à chaque signal est associé un ou plusieurs pilotes (**driver**). Ces pilotes contiennent le futur possible du signal. On ne modifie jamais un signal directement mais on modifie un de ses pilotes associé. La règle de création d'un pilote est la suivante : on crée un pilote par processus qui modifie le signal. Dans le cas où un signal possède plusieurs pilotes, on parle alors de signal multi-sources, il faut que le signal soit résolu. C'est à dire, qu'une fonction permette de calculer une valeur unique à partir de l'ensemble des valeurs de tous les pilotes. Cette fonction s'appelle fonction de résolution. Le type d'un signal n'est pas limité à des types pré-définis, il peut être n'importe quel type utilisateur, du simple tableau jusqu'à l'enregistrement le plus complexe. Par contre un signal ne peut pas être un pointeur ou un fichier.

Lorsque l'on affecte un signal on donne un délai, en effet il est impossible de modifier instantanément la valeur d'un signal. Dans le cas où l'on donne un délai explicitement nul ou si on n'en donne pas, l'exécuteur VHDL introduit un pas temporel infinitésimal appelé delta-délai. Ce sont ces délais qui maintiennent la causalité.

Les processus sont activés par des événements sur les signaux ou tout simplement par une heure de réveil. L'événement sur un signal est une modification de sa valeur courante. La mise en attente sur un signal, peut être complétée par une condition booléenne. Par exemple, un processus est sensible sur l'horloge mais uniquement sur le front montant.

d. L'opposition signal-variable

En VHDL, il existe la notion de variable. Cet objet est uniquement destiné à la description algorithmique. C'est pourquoi en VHDL'87 les variables peuvent être définies uniquement dans les processus et les sous-programmes. Une variable ne peut pas servir à l'échange d'information entre des processus, seul le signal est habilité à faire cela. Cette restriction vient du fait qu'une variable peut être modifiée instantanément, et on connaît les risques du partage de ressources critiques en informatique, et surtout le risque de la perte de la causalité. Comme nous le verrons plus loin, la norme VHDL'92 devrait inclure les variables globales.

e. La description d'un système : l'entité et l'architecture

En VHDL, un système est représenté par une entité (**entity**) et plusieurs architectures (**architecture**). Ce sont deux unités de conception du langage. L'entité représente la vue externe du système et l'architecture une description interne. Une entité peut donc posséder plusieurs architectures qui sont chacune la description du système à un niveau d'abstraction donné.

i. L'entité - LRM 1.1

Cette unité représente la description externe d'un système. On y trouve la définition des connexions du système et de ses paramètres génériques. Il peut également y avoir une liste d'instructions concurrentes pour des vérifications statiques, par exemple de temps d'établissement (setup) et de maintien (hold) pour des circuits numériques. En aucun cas, ces instructions ne peuvent modifier un objet VHDL.

D'autre part, il y a une partie déclarative dont les déclarations seront visibles de toutes les architectures qui lui sont associées. Voici la syntaxe relative à la déclaration d'une entité sous Forme de Backus-Naur :

```
entity_declaration ::=
    entity nom_entity is
        [ generic ( generic_list ) ; ]
        [ port ( port_list ) ; ]
        { entity_declarative_item }
    [ begin
        entity_statement_part ]
    end [ nom_entity ];
```

```
entity_statement_part ::=
     concurrent_assertion_statement
     | passive_concurrent_procedure_call
     | passive_process_statement
```

Un exemple d'entité d'une porte and à n entrées :

```
entity andn is
     generic (n : positive; delai : time);
     port (i : in bit_vector (1 to n); o : out bit);
     -- zone déclarative
begin
     -- possibilité de mettre des instructions concurrentes
     -- passives de controle, utiles pour la spécification
end andn;
```

Déclaration de Generic

Il s'agit d'une liste de constantes qui servent à paramétrer la description du système. Leurs valeurs seront connues au moins à la fin de l'élaboration. On peut leur donner une valeur par défaut. Ces paramètres peuvent être aussi bien des délais de propagation que des tailles de bus qui seront utilisées dans la déclaration des connexions.

Déclaration de Port

Il s'agit d'une liste de connexions possibles avec l'extérieur. Un port est un signal formel (**formal**), il sera connecté (mapping) dans la hiérarchie à un signal ou à un autre port. Ces derniers sont alors appelés signaux effectifs (**actual**).

Un port possède un type et un mode. Le type informe sur ce que peut contenir le signal, et il connaît les mêmes restrictions que pour le type d'un signal. Quant au mode, il indique ses relations avec l'extérieur. Il existe cinq modes :

- **in**, qui indique que le port importe une valeur dans le bloc, ne peut qu'être lu dans celui-ci,
- **out**, indique que le port exporte une valeur du bloc, ne peut qu'être écrit, possibilité de donner une valeur par défaut,
- **inout**, qui indique que le port importe et exporte une valeur, peut être lu et écrit,
- **buffer**, qui indique que le port ne pourra charger qu'une seule source,
- **linkage**, qui est un mode spécial introduit pour des connexions possibles avec d'autres

langages. Son utilisation n'est pas spécifiée.

ii. L'architecture - LRM 1.2

Cette unité permet de décrire l'intérieur du système. Cette description peut aussi bien être structurelle, comportementale, flots de données ou bien les trois à la fois. En effet, une des spécificités de VHDL est de pouvoir mélanger les trois styles de description au sein d'une même architecture. L'architecture est composée de deux parties : une première déclarative dans laquelle on peut définir des objets VHDL, et la seconde où l'on trouve la description qui est une liste d'instructions concurrentes.

Une des caractéristiques de VHDL est d'avoir disjoint la description de la vue externe et interne d'un composant. Ainsi, à une entité peuvent correspondre plusieurs architectures.

Voici deux architectures possibles de l'entité "andn" :

```
architecture comportementale of andn is
    function cherche_zero (s : bit_vector) return bit is
        l : for k in s'range loop
            if s(k) = '0' then return '0'; end if;;
        end loop l;
        return '1';
    begin
    end function cherche_zero;
begin
    o <= cherche_zero(i) after delai*(n-1);
end comportementale;

architecture comportementale_bis of andn is
begin
    p1 : process
        variable v : bit;
    begin
        wait on i;
        v := '1';
        l : for j in i'range loop
            if i(j) = '0' then v:='0'; end if;;
        end loop l;
        o <= v after delai*(n-1);
    end process p1;
end comportementale_bis;
```

L'identificateur de l'architecture peut être quelconque, certains les nomment a, b et c, d'autres préfèrent être explicites et les nomment structurelle, comportementale ou flot_de_ données.

iii. Les blocs

Un couple entité-architecture s'appelle bloc externe en opposition à l'instruction concurrente VHDL qui est appelé bloc interne. D'ailleurs, l'interface d'un bloc interne et celle d'une entité sont très proches : on y trouve les mêmes informations sur les paramètres et les connexions. Cette appellation vient du fait qu'un couple entité-architecture peut être utilisé via un composant (cf. paragraphe suivant) dans un bloc pour une description structurelle. Dans ce cas, le LRM spécifie la création de deux blocs imbriqués lors de l'élaboration (LRM - 9.6).

iv. en VHDL-analogique

Ces notions relatives à la hiérarchie vont être étendues à la description analogique. Ainsi, il n'y aura pas d'entité ou d'architecture spécifique aux systèmes analogiques. Un système sera toujours décrit au moyen d'un couple entité-architecture, et ce sont ses connexions qui permettront de savoir s'il est numérique, analogique ou bien mixte. Cela implique que le bloc sera lui aussi étendu pour la description de systèmes analogiques.

f. La notion de support : les composants - LRM 4.5

```
component_declaration ::=
    component nom_component
        [ generic ( generic_list )]
        [ port ( port_list )]
    end component;
```

La notion de hiérarchie nécessite la possibilité de connecter dans une architecture des sous-systèmes déjà présents en bibliothèque sous forme de couples entité-architectures. En VHDL'87, on ne peut pas directement connecter une entité dans un bloc. Il faut passer par un composant (**component**). Cet objet est une représentation externe du système, qui sera donc proche et souvent identique à l'entité. C'est donc ce composant qui sera instancié dans un bloc. Ensuite, au moyen d'une configuration de composant ou d'une spécification de configuration, on associera l'entité à l'instance de composant. La première se fera localement dans le bloc, et la spécification dans une unité de conception spéciale : l'unité de configuration (**configuration**).

Cette manière de faire est un peu lourde, mais elle permet une très grande puissance dans la description structurelle. D'une part une architecture ne dépend pas directement des autres entités

qu'elle référence. D'autre part, il devient assez facile de changer de couples entité-architecture : il suffit de modifier la spécification de configuration.

Le composant représentant une entité ne doit pas obligatoirement posséder tous ses paramètres génériques. De la même manière, les identificateurs des paramètres génériques ainsi que ceux des ports peuvent différer de ceux de l'architecture. Dans ce cas, il conviendra de les associer (mapping) lors de la configuration, qui devient alors obligatoire.

Par contre, il faut que les ports qui sont associés aient un mode compatible et un type identique ou compatible.

Pour finir, dans le cas ou aucune configuration ne serait donnée, par défaut l'élaboration VHDL configure avec l'entité portant le même nom et avec la dernière architecture compilée.

Voici un exemple d'utilisation d'un composant dans une description structurelle :

```
entity nandn is
    generic (n : positive; delai : time);
    port (i : in bit_vector (1 to n); o : out bit);
end andn;

architecture structurelle of nandn is
    component porte_n_entrees
        generic (retard : time; p : positive);
        port (input : in bit_vector (1 to n); output : out bit);
    end component;
    signal not_o : bit;
begin
    porte_et : porte_n_entrees
        generic map (n, delai)
        port map (i, not_o);
    o <= not not_o;
end structurelle;
```

Cette notion de composant sera étendue à VHDL-analogique, et donc le composant tout comme l'entité et le bloc verra son interface complétée avec des connexions analogiques.

g. La notion de configuration - LRM 1.3

Comme nous l'avons déjà dit plus haut, il existe deux manières de configurer une instance de composant en VHDL : utiliser directement une spécification de configuration, ou bien passer par une unité de configuration.

La spécification de configuration se donne directement dans la zone déclarative du bloc dans laquelle se trouve l'instance de composant. Pour l'exemple précédent, il faut rajouter les lignes suivantes dans la zone déclarative de l'architecture :

```
...
for    porte_et    :    porte_n_entree        use    entity
    work.andn(comportementale)
            generic map (n => p, delai => retard)
            port map (i => input, o => output);
...
```

Cette manière de procéder est plus simple et plus rapide, mais on perd en souplesse de description. Cette description sera liée avec la description de la porte andn.

Par contre, l'utilisation d'une unité de configuration permet de rendre indépendantes les deux descriptions. Cette unité est la plus spécifique à VHDL, elle le rend très puissant et permet, de plus, de faire des descriptions modulaires et ainsi d'utiliser au mieux la compilation séparée. Une unité de configuration est attachée à un couple entité-architecture.

On peut associer une unité de configuration à l'entité-architecture nandn(structurelle) :

```
configuration cfg_nandn of nandn is
    for structurelle
        for porte_et : porte_n_entrees
            use entity andn(comportementale)
                -- association des paramètres générique
                -- entre le component et l'entity
                generic map (n => p, delai => retard)
                -- idem pour les ports
                port map (input, output);
        end for;
    end for;
end cfg_nandn;
```

Au moment de la configuration, on indique la correspondance des paramètres génériques et des ports entre l'entité et le composant si ceux-ci ont des noms différents. Il se peut d'autre part, que le composant n'ait pas de paramètres génériques, dans ce cas il faut donner des valeurs pour ceux de l'entité.

Là encore, cette notion de configuration est maintenue pour la description analogique. On peut donc dire que la description structurelle analogique sera très proche de la description structurelle numérique.

h. Session VHDL

On peut faire une description VHDL pour spécifier un système ou pour utiliser un outil informatique. Dans ce cas on parle d'exécution de la description VHDL. Généralement cette exécution est une simulation ou une synthèse, mais on peut envisager un outil de preuve formelle ou encore de placement routage. Dans le LRM la sémantique associée au langage est une sémantique de simulation.

2. Les unités de conception (design units)

Les unités de conception sont les blocs de base du langage, elles sont au nombre de cinq : entité, architecture, paquetage (**package**), corps de paquetage (**package body**) et configuration (**configuration**). Ce sont ces unités qui sont gérées dans les bibliothèques VHDL.

Une **architecture** est associée à une **entity**, tout comme un **package body** est associée à un **package**. C'est pourquoi, on parle d'unités primaires et secondaires. Les unités secondaires sont associées à une unité primaire et elles ne peuvent pas se trouver en bibliothèque sans celle-ci. Par contre une unité primaire peut être en bibliothèque sans avoir d'unité secondaire associée.

a. Le paquetage (package) - LRM 2.5

Cette unité est un ensemble de définitions d'objets VHDL qui pourront être utilisées par plusieurs autres unités de conception en référençant le paquetage. Pour les fonctions et les procédures, seule leur déclaration se trouve dans cette unité.

Par exemple, la fonction cherche_zero déclarée dans l'architecture de la porte andn ou encore le composant porte_n_entree, auraient pu être inclus dans un paquetage et ainsi être utilisés dans une autre description :

```
package decl_locale is
    -- toutes les déclarations faites ici sont utilisables
    -- par les unités qui référencent le paquetage

    function cherche_zero (s : bit_vector) return bit;

    component porte_n_entrees
        generic (p : positive; retard : time);
        port (input : in bit_vector (1 to n); output : out bit);
    end component;
```

```
    signal not_o : bit;
end decl_locale ;
```

b. Le corps de paquetage (package body) - LRM 2.6

Dans le corps de paquetage on trouve les corps des fonctions et des procédures dont la déclaration se trouve dans le paquetage associé. On peut également y trouver des objets ou des fonctions et procédures non déclarées dans le paquetage. Tous ces objets ne sont donc pas visibles à l'extérieur du corps du paquetage, et leur utilisation est restreinte à celui-ci. En effet seuls les objets qui sont définis dans le paquetage sont exportables.

Voici le corps de paquetage associé au paquetage précédent :

```
package body decl_locale is
    -- Les déclarations faites ici ne sont pas utilisables par
    -- les unités référençant le paquetage
    function cherche_zero (s : bit_vector) return bit is
    begin
        l : for i in s'range loop
            if s(i) = '0' then return '0'; end if;;
        end loop l;
        return '1';
    end function cherche_zero;
end decl_locale;
```

3. Les déclarations et les objets - LRM 4

Chaque unité de conception, exceptée la configuration, possède une partie déclarative dans laquelle on peut trouver toutes les déclarations qui suivent. Chacune de ces parties déclaratives rend visible les déclarations dans une zone spécifique appelée la portée (**scope**).

Ainsi, les déclarations faites dans une entité seront visibles dans cette entité et dans toutes les architectures qui lui sont associées. Celles faites dans une architecture seront visibles uniquement dans celle-ci.

Dans un paquetage, les déclarations pourront être visibles des unités de conceptions qui référencent ce paquetage. Par contre, dans un corps de paquetage, elles ne seront visibles que dans celui-ci.

On trouve également une zone déclarative dans les sous-programmes et dans deux instructions :

les blocs et les processus.

a. Type - LRM 4.1

En VHDL, il est possible de définir n'importe quel type utilisateur, du plus simple au plus abstrait. Ainsi, il est possible d'avoir des enregistrements, des pointeurs ou encore des tableaux.

VHDL est un langage fortement typé, il est impossible d'affecter ou de connecter ensemble deux objets si leurs types sont différents bien qu'ils représentent la même chose. Il existe cependant un cas ou cela est possible, si l'un est sous-type de l'autre. La syntaxe ne présente aucune difficulté, elle reste classique.

Cependant, il existe la notion de type physique. Ces types sont des types entiers définis à partir d'une sous-unités de base :

```
type distance is range -1e18 to 1e18
    units
        -- unité de base
        nm;                          -- nano mètre
        um = 1000 nm;
        mm = 1000 um;
        metre = 1000 mm;
        mil = 25400 nm;              -- unités anglaises
        inch = 1000 mil;
        ft = 12 inch;
        yard = 3 ft:
        mile = 5280 ft;
    end  units;

-- il est maintenant possible de faire :
signal dist : distance := 3 meter - 5ft + mm * 12;
```

Il existe un type physique prédéfini qui est le type permettant de représenter les durée, le type **time**. L'unité de base de ce type est la femto-seconde.

Cette notion de type physique est très utile pour un HDL analogique où l'on manipule des formules dimensionnées. Malheureusement, la notion VHDL ne permet pas de faire cette vérification.

b. Sous-type - LRM 4.2

```
subtype_declaration ::=
    subtype nom_subtype is [ nom_res_fonction ]
        nom_type [ constraint ];
```

Un sous-type est un type défini à partir d'un type père, dont il hérite des fonctions de

manipulation. De plus, outre ces notions d'héritage, le sous-type est à la base d'un mécanisme subtil, celui de la gestion des conflits pour les signaux possédants plusieurs sources. Ainsi, à un sous-type peut être associée une fonction appelée fonction de résolution qui permettra de résoudre ces conflits. Cette gestion est donc complétement ouverte à l'utilisateur.

c. Constante - LRM 4.3.1.1

C'est un objet classique des langages de programmation, elle contient une valeur déterminée durant toute l'exécution du code VHDL. Cette valeur est connue au maximum à la fin de l'initialisation.

```
constant pi : real := 3.1415926;
```

d. Variable - LRM 4.3.1.3

La variable est typiquement un objet utilisé pour l'algorithmique. C'est pourquoi on peut la déclarer uniquement dans une zone déclarative de code séquentiel : sous-programme ou processus.

Dans un processus, sa valeur est rémanente d'un réveil à l'autre. Par contre, un sous-programme ne conserve pas la valeur de ses variables d'un appel à l'autre, car là il s'agit bien d'appel différent. Une exception existe, car on peut trouver une instruction de mise en attente dans une procédure, on est alors dans un cas équivalent au processus, et alors le contenu de la variable est conservé.

Il est à noter que pour VHDL'92 il sera possible de définir des variables globales. Au delà de la polémique, on peut dire que celles-ci remettent en cause le déterminisme du langage.

```
variable a, b, c : integer;
```

e. Les déclarations et les corps de sous-programmes - LRM 2.1 & 2.2

En VHDL, les sous-programmes ont le même rôle que dans un langage de programmation. La déclaration consiste juste en l'écriture de l'en-tête du sous-programme avec le nom, le mode, la classe et le type de ses paramètres ainsi que le type de retour pour une fonction.

Le corps du sous-programme doit se trouver dans le corps de paquetage associé si la déclaration est faite dans un paquetage, ou bien juste après pour les autres zones déclaratives.

En fait, cette déclaration est utilisée dans les paquetages et pour permettre la récursivité croisée.

Pour les fonctions, les paramètres sont passés uniquement par valeur (**in**), alors que pour les procédures ils peuvent être soit passés par valeur (**in**), soit passés par référence sans droit de lecture (**out**) ou avec (**inout**).

D'autre part, les paramètres peuvent conserver leur classe, ainsi un signal peut être considéré soit comme un signal soit comme une constante à l'intérieur d'un sous-programme. Cependant, dans le cas de la fonction les paramètres sont tous en mode **in**, et la classe variable est interdite.

Un autre concept important est la surcharge des sous-programmes. Comme en ADA, il est possible de donner le même nom à un sous-programme si la liste de paramètres (appelé signature) diffère : soit par le type d'un paramètre, soit par le nombre de paramètres. Il est également possible de définir des opérateurs arithmétiques et logiques sur des nouveaux types, ou bien de re-définir ceux existant.

f. Attribut - LRM 4.4 & 5.1

Un attribut est un objet qui est attaché à un autre objet ou à une unité de conception. Il existe un ensemble pré-défini d'attributs sur les types, les signaux et les tableaux. Ces attributs sont soit des fonctions prenant un ou plusieurs arguments, soit des signaux.

Comme nous l'avons vu, il est impossible de créer des signaux à l'exécution. Cela implique l'interdiction d'utiliser les attributs signaux dans un sous-programme. En effet, lors de l'appel de la fonction il faudrait créer un signal.

L'utilisateur peut définir ses propres attributs. Ceux-ci peuvent être associés aux objets suivants : entité, architecture, configuration, procédure, fonction, paquetage, type, sous-type, constante, signal, variable, composant et label. La valeur de l'attribut est statique, c'est à dire connu à la fin de l'élaboration et non modifiable en cours d'exécution.

Ce mécanisme est utile pour la spécification. Ces attributs ne sont pas utilisés en simulation. Par contre, ils peuvent servir pour un outil de rétro-annotation pour spécifier les valeurs des capacités des ports d'une entité. En général, ils sont utilisés par les outils de traitements formels.

L'utilisation des attributs se fait en suffixant au nom de l'objet concerné une apostrophe (tick en anglais) suivie du nom de l'attribut.

g. Alias - LRM 4.3.4

Ce mécanisme permet de renommer un objet. Il peut être utilisé pour nommer les différents bits d'un vecteur : ainsi on pourra nommer le bit de débordement, de zéro, de parité et de demi-débordement d'un accumulateur d'un processeur.

```
signal a : bit_vector (1 to 8);

alias Z : bit is a(6);
alias H : bit is a(3);
alias C : bit is a(4);
```

h. Tableau Récapitulatif

Le tableau suivant indique dans quelles zones déclaratives peut se trouver chaque déclaration.

	Entité	Architecture	Paquetage	Corps de Paq.	Configuration	Bloc	Processus	sous-prog
Décl. de sous-programme	oui	oui	oui	oui		oui	oui	oui
Corps de sous-programme	oui	oui		oui		oui	oui	oui
Décl. de type, sous-type et constante	oui	oui	oui	oui		oui	oui	oui
Décl. de variable							oui	oui
Décl. d'alias, de fichier et d'attribut	oui	oui	oui	oui		oui	oui	oui
Décl. de composant		oui	oui			oui		
Décl. de signal	oui	oui	oui			oui		
Spéc. de déconnexion	oui	oui		oui		oui		

4. Les instructions concurrentes - LRM 9

Dans le manuel de référence, à chaque instruction concurrente est associé un équivalent sous forme de processus. En fait, une description est un ensemble de processus qui s'exécutent en concurrence les uns par rapport aux autres, et qui communiquent au moyen de signaux. Certains

parlent d'une flotte de processus noyée dans un océan de signaux.

a. Le bloc - LRM 9.1

Nous avons déja présenté le bloc, voici sa syntaxe :

```
block_statement ::=
    block_label : block [ (guarded_expression) ]
        [ generic (generic_list)
        [ generic map (association_list); ]]
        [ port (port_list)
        [ port map (association_list); ]]

        { block_declarative_item }

    begin
        { concurrent_statement }
    end block [ block_label];
```

Dans la partie déclarative, on peut définir des clauses port ou générique qui permettent de le connecter au bloc qui le contient. Dans ce cas il faut immédiatement les associer. Les ports seront associés à des signaux ou des ports du bloc supérieur. Cette caractéristique n'est pas employée en description, elle permet de donner un équivalent à une instance de composant configuré avec un couple entité-architecture (LRM 9.6).

Dans un bloc il est possible de référencer tous les objets définis dans les hiérarchies supérieures, mais le contraire est faux.

Enfin, il est possible de définir des blocs gardés qui regroupent un ensemble d'affectations de signaux en attente sur le même . L'exemple typique est le cas de traitement synchrone où les instructions attendent un front d'horloge pour s'exécuter.

```
synch : block (clock = '1' and not clock'stable)
begin
    o <= guarded cherche_zero(i) after delai;
end block synch;
```

En modélisation, le bloc est utilisé pour regrouper des déclarations, structurer une architecture complexe et pour définir une condition de garde.

b. Le processus - LRM 9.2

Voici la syntaxe du processus comme instruction concurrente :

```
process_satement ::=
    process_label : process [ (sensitivity_list) ]
        { process_declarative_item }
    begin
        { sequential_statement }
    end process [ process_label ];
```

L'exécution d'un processus est en fait une boucle sans fin. C'est pour cette raison qu'il ne faut pas oublier de mettre une instruction de mise en attente sur un événement ou de donner une liste de sensibilité. Cette liste est composée de signaux, dés qu'un événement survient sur l'un de ceux-ci, le processus est exécuté jusqu'à sa fin et il se remet en attente sur sa liste de sensibilité.

En ce qui concerne les déclarations, il est interdit de déclarer des signaux, mais on peut déclarer des variables.

Enfin, l'exécution d'un processus se fait sur un même delta-délai, le changement de temps ne se fait qu'après une instruction de mise en attente.

c. Les instructions de génération

Il existe deux instructions de génération : l'instruction conditionnelle et l'instruction de boucle. Ces deux instructions génèrent des blocs. La première sur l'évaluation d'une condition en génère un, et la seconde en génère une liste dont le nombre est donné par les indices de la boucle.

L'instruction de génération de boucle est utile pour modéliser les structures répétitives, structures fréquentes en électronique numérique. Par exemple on peut reprendre la description de la porte and à n entrées :

```
architecture structurelle of andn is
    component porte_and
        generic (delai : time
        port     (entree_1, entree_2 : in bit;
                  sortie : out bit);
    end  component;

    signal b : bit_vector (2 to n-1);

begin
    g : for j in 1 to n-1 generate
        iprem : if j=1 generate
            first : porte_and
                generic map (delai)
                port map (i(1), i(2), b(2));
        end  generate;
```

```
        imilieu : if j>1 and j<n-1 generate
             et : porte_and
                  generic map (delai)
                  port map (b(j), i(j+1), b(j+1));
        end  generate;

        ider : if j=n-1 generate
             last : porte_and
                  generic map (delai)
                  port map (b(n-1), i(n), sortie);
        end  generate;
    end  generate g;
end structurelle;
```

Chacun des composants de la boucle de génération est configurable indépendamment. Par exemple, voici une configuration possible, qui n'est valable cependant que pour n supérieur à 6 :

```
library techno1, techno2;
configuration cfg_andn_structurelle is
    for structurelle
        for g(2 to 5)
            for imilieu
                for et : porte_and
                    use entity techno2.and(a)
                end for;
            end for;
        end for;
        for g(6 to n-2)
            for imilieu
                for et : porte_and
                    use entity techno2.and(a)
                end for;
            end for;
        end for;
        for g(1)    -- cas particulier des if … generate
            for iprem, ider
                for et : porte_and
                    use entity techno2.and(a)
                end for;
            end for;
        end for;
        for g(n-1)  -- cas particulier des if … generate
            for iprem, ider
                for et : porte_and
                    use entity techno2.and(a)
                end for;
            end for;
        end for;
    end for;
end cfg_andn_structurelle;
```

d. L'affectation de signal

Cette instruction est équivalente à un processus dont la liste de sensibilité est composée de tous les signaux apparaissant en partie droite de l'affectation. S'il n'y en a pas, alors l'instruction n'est

exécutée qu'une fois. Cette instruction correspond à une connexion physique entre les deux parties.

Cette instruction est très complète, elle permet d'affecter des formes d'onde complexes grâce à ses deux structures l'une conditionnelle et l'autre de sélection de choix. Ces différentes instructions sont utilisées pour donner des stimuli, ou encore pour décrire le comportement d'un composant sous forme d'équations logiques.

Enfin, il existe deux modes d'affectation pour les signaux : le mode transport ou le mode inertiel qui est le mode par défaut. Le premier prend en compte toutes les transactions, alors que le second réalise un filtre passe bas pour éliminer les impulsions dont la durée est inférieure au temps de propagation.

Il existe un cas pour lequel cette instruction n'est pas une connexion physique, il s'agit de l'affectation gardée. Elle ne peut être utilisée que dans un bloc gardé. Dans ce cas, le signal est connecté à l'expression pendant tout le temps où la condition est vraie plus pour la durée spécifiée par la spécification de déconnexion après que cette condition soit devenue fausse. S'il n'y a pas de clause de déconnexion ou si le temps donné est nul, la déconnexion a lieu un delta-délai après.

```
synch : block (clock = '1' and not clock'stable)
    disconnect o : bit after 10 ns;
begin
    o <= guarded cherche_zero(i) after delai;
end block synch;
```

e. L'appel de procédure concurrent

Cette instruction est équivalente à un processus dont la liste de sensibilité est composée de tous les signaux qui sont en lecture (en mode **in** ou **inout**) à l'appel, et dont le corps est un appel à cette procédure. Il n'est donc pas obligatoire que la procédure contienne une instruction de mise en attente sur un événement. Cependant, s'il n'y en a pas, il va y avoir une élaboration de la procédure à chaque événement sur la liste de sensibilité, et on perd les valeurs des variables locales à cette procédure.

Cette méthode de description est utile pour rapidement valider une méthode, ou pour générer plusieurs fois le même processus qui travaille sur des signaux et paramètres différents. Cependant on obtient une sorte de description structurelle à un niveau de hiérarchie, qui est figée et très peu

maniable.

Reprenons l'architecture de la porte and à n entrées :

```
architecture comportementale_ter of andn is
    procedure andn_comp (i : in bit_vector; o : out bit) is
        variable v : bit;
    begin
        v := '1';
        l : for j in i'range loop
            if i(j) = '0' then v:='0'; end if;;
        end loop l;
        o <= v after delai*n;
    end andn_comp;
begin
    p1 : andn_comp (i, o);          -- equivalent à
                                    -- process
                                    -- begin
                                    --     andn_comp (i, o);
                                    --     wait on i;
                                    -- end process;
end comportementale_ter;
```

Dans ce cas, à chaque événement sur le signal i, la procédure va être appelée et élaborée, ce qui consiste en la création de son environnement. Pour éviter ceci, on peut mettre l'instruction wait dans la procédure et définir celle-ci comme une boucle sans fin :

```
    procedure andn_comp (i : in bit_vector; o : out bit) is
        variable v : bit;
    begin
        endless : loop
            v := '1';
            l : for j in i'range loop
                if i(j) = '0' then v:='0'; end if;;
            end loop l;
            o <= v after delai*n;
            wait on i;
        end loop endless;
    end andn_comp;
```

Dans ce cas, on ne ressort plus de la procédure.

Ce concept d'appel de procédure concurrent a été étendu pour VHDL-analogique. Ainsi un tel appel sera un raccourci d'instance de composant analogique comme il est un raccourci d'instance pour un composant numérique dans VHDL.

f. Instanciation de composant

C'est l'instruction qui permet de faire de la description structurelle. Pour cela il faut juste donner la valeur des paramètres génériques et associer à chaque port du composant soit un signal soit un port

de l'entité qui est associée à l'architecture dans laquelle on se trouve.

5. Les instructions séquentielles

Ces instructions permettent de décrire de manière algorithmique le comportement de tout ou une partie d'un système. On les trouve au sein des processus ou bien des sous-programmes (fonction et procédure). Le langage VHDL offre toutes les instructions classiques d'un langage de programmation plus l'instruction de mise en attente sur un événement qui bloque le processus et l'affectation de signal.

Ce sont les seules que nous présenterons.

a. L'affectation de signal

Comme on l'a présenté précédemment, dans un processus on ne peut pas modifier la valeur courante d'un signal. On ne peut que proposer un ensemble de valeurs futures en écrivant dans le pilote que ce signal possède dans ce processus.

Si on a besoin de stocker des calculs intermédiaires, il faut utiliser la variable. En effet, contrairement au signal, celle-ci est modifiable instantanément dans un processus ou un sous-programme.

b. L'instruction de mise en attente d'un processus

```
wait_statement ::=
    wait [ on signal_name {, signal_name }
         [ until condition ]
         [ for time_expression ];
```

Cette instruction permet de se mettre en attente sur un événement sur un signal (**on**), ou sur une condition booléenne (**until**) ou pour au moins une durée (**for**). Dans le cas où aucune clause n'apparaît, le processus se bloque jusqu'à la fin de l'exécution.

Cette instruction peut être utilisée dans une procédure, mais pas dans une fonction ni dans une procédure se trouvant dans la liste d'appel d'une fonction. De la même manière, on ne peut pas l'utiliser dans un processus qui possède une liste de sensibilité ni dans une procédure appelée par un

tel processus.

6. L'élaboration, l'initialisation et la simulation

La seule sémantique proposée pour VHDL dans son LRM est une sémantique de simulation. Bien entendu, rien n'interdit d'en définir une autre, mais elle ne sera pas standard.

Une description simulable d'après le LRM pour VHDL'87 est un couple entité-architecture qui ne possède pas d'interface : ni de génériques ni de ports, ou bien une unité de configuration sur un tel couple. Cette contrainte est supprimée pour VHDL'92 et tout couple entité-architecture sera simulable.

L'élaboration VHDL consiste en un parcours de la description et en la création de la hiérarchie en terme de blocs, processus et signaux. Donc chaque instance de composant est remplacée par ses blocs équivalents (cf. LRM-9.6.1) et tous les signaux sont créés, ainsi que les pilotes associés. Une fois l'élaboration finie, on passe à la phase d'initialisation.

Voici les deux principales phases de l'initialisation :

- la valeur des signaux est calculée grâce à la propagation des valeurs effectives (**effective**) et motrices (**driving**), en faisant appel à la fonction de résolution si le signal est résolu,
- en l'exécution de tous les processus jusqu'à leur mise en attente.

Maintenant la simulation peut commencer.

III. Aperçu du futur langage : VHDL-Analogique

Après avoir présenté VHDL, nous allons voir point par point quelles modifications sont nécessaires pour pouvoir modéliser des systèmes analogiques. La normalisation en est actuellement à la phase de conception du langage (language design), les groupes de travail écrivent des LES (Language Extension Specification). Ce sont ces documents que l'on référencera dans les paragraphes suivants.

Toutefois, le premier paragraphe fera le point sur les besoins de la description analogique.

1. Les besoins de la description analogique

Dans le cas général, une description d'un système peut servir aussi bien pour de la documentation ou encore de la spécification que pour des outils informatiques tel qu'un simulateur. En électronique analogique, les outils de CAO se limitent essentiellement aux simulateurs électriques, aux outils de placements routage et aux outils d'optimisation. Il existe bien des outils de synthèse, mais leur utilisation reste marginale et leur champs d'application se restreint à une architecture donnée, amplificateur opérationnel, filtre ou encore comparateur. D'autre part, il paraît utopique de croire à un outil de synthèse de haut niveau prenant directement en entrée des équations tellement le champs de cette discipline est vaste.

Dans tous les cas, il semble qu'une description d'un système analogique doit pouvoir se faire de deux manières principales, avec la possibilité de les mélanger. Tout d'abord, la manière la plus intuitive est de pouvoir donner la liste d'équations qui régissent ce système. Cette méthode est très bonne pour la documentation et est utilisable par les simulateurs. La seconde solution, plus orientée pour les simulateurs, est la description procédurale où le système est décrit avec une sorte de procédure dans laquelle on mène des calculs explicites pour calculer des états en fonction de variables. Cette méthode, utilisée en interne dans les simulateurs, reste la plus efficace pour ceux-ci.

En ce qui concerne la description en elle même, un système analogique peut être décrit de plusieurs manières différentes selon que l'on donne son comportement en temporel, en fréquence ou en courant continu. Il faut donc pouvoir associer un de ces domaines d'interprétation à une description. D'autre part, les descriptions en temporel et en courant continu utilisent des opérateurs et des valeurs réelles, alors que la description en fréquence utilise des opérateurs et des valeurs complexes. Il faut donc pouvoir gérer cette dualité.

La description analogique fait appel à des équations possédant une dimension et à des valeurs possédant une unité, il semble donc normal de pouvoir introduire ces notions pour des raisons de documentation et de lisibilité des formules.

2. Trois nouveaux types - LES A

Pour pouvoir simuler correctement, un simulateur analogique a besoin d'une certaine précision. C'est pourquoi un nouveau type réel a été introduit pour VHDL-Analogique. Ce type se nomme **float** et il devra posséder la précision suffisante pour permettre la simulation analogique. C'est-à-dire au moins soixante quatre bits, mais cela reste dépendant de l'implémentation. Il n'y aura pas de lien explicite entre le type VHDL **real** et le nouveau type **float**. Il faudra passer par une instruction de conversion standard. En pratique, les objets analogiques ne pourront pas être de type **real**, cela devra être une erreur. Par contre, côté numérique, on pourra avoir des objets (variable, constante) de type **float**.

Le second type introduit est le type **complex** qui est un enregistrement dont les deux champs sont de type **float**.

Le troisième et dernier type proposé est plus ambigu, il s'agit du type **analog**. Ce type présente la particularité de posséder deux représentations différentes qui dépendent du mode de description. En mode courant continu et transitoire il est équivalent au type **float**, mais en mode fréquentiel, il est équivalent au type **complex**. Ce dernier type est utile pour décrire le comportement en fréquence d'un système.

3. Les connexions - LES G & H

L'actuelle description structurelle de VHDL est suffisante pour permettre la description au niveau électrique. Mais les connexions analogiques et numériques sont fondamentalement différentes. Ainsi, lorsque des ports sont connectés ensemble, la valeur du signal qui leur est relatif est unique, bien que les pilotes puissent avoir des valeurs différentes. Dans ce cas particulier, une fonction de résolution définie par l'utilisateur se charge de trouver la valeur du signal. Par contre, les connexions analogiques impliquent une vérification des lois de Kirchoff (ou équivalentes). En d'autre termes, la somme en un noeud des courants des différentes branches doit être nulle et il y a unicité de la tension.

C'est pour cette raison qu'il a été proposé de rajouter une nouvelle sorte d'interface : les pins (**pin**).

Les objets connectés sur ces pins seront les noeuds (**node**). C'est en ces noeuds que sera vérifiée la loi des noeuds de Kirchoff. On note bien le parallèle entre les deux domaines : noeud avec signal et pin avec port.

Les pins et les noeuds devront avoir un type qui permette la représentation du couple courant-tension. Ce point n'est pas encore bien défini au niveau de la standardisation. Il est clair que la tension sera une valeur, par contre le courant d'un noeud n'existe pas. En effet, en un noeud, on résout la loi de Kirchoff disant que la somme des courants de branches est nulle. Une solution envisagée est de dire que le champ courant est la liste des courants de branche. Un autre point n'est pas encore très clair. En effet, l'objet pin est le formel correspondant à l'objet noeud. En VHDL cela signifie que ces deux objets ont les mêmes propriétés (un port est un signal). Cependant, il peut être intéressant d'introduire une différence. La pin étant alors une branche de courant, il devient ainsi possible de référencer son courant.

Dans un paragraphe ultérieur, nous verrons la notion de contribution en courant d'une instruction, d'un bloc ou d'une instance de composant analogique. Cette contribution correspond à un courant de branche.

4. Les domaines d'interprétation - LES L

Actuellement, l'exécution d'une description VHDL se fait en temporel. Par contre, l'exécution d'une description analogique peut conduire à une exécution en courant continu, en temporel, en fréquence ou encore mixte fréquentielle-transitoire. D'autre part, comme nous l'avons vu, un système analogique peut être décrit de manière différente pour chaque domaine d'application : temporel, courant continu ou en fréquence. Une description d'un système analogique pourra couvrir un, deux ou bien les trois domaines d'interprétation. Par contre, au moment de la simulation, le simulateur en utilisera certaines que devra posséder obligatoirement la description du composant. Ainsi pour une simulation en fréquence, il faut une description en courant continu et une description en fréquence.

Il est évident qu'une description mixte numérique-analogique ne peut conduire qu'à une exécution temporelle. Donc, par défaut la description analogique faite en VHDL-Analogique sera

dédiée à l'analyse transitoire. Cependant, il sera possible de décrire le comportement d'un système pour d'autres analyses (en fréquence, mixte transitoire-en fréquence), mais aucune sémantique ne sera définie dans le LRM.

5. La description comportementale - LES I & K

Au départ, il y avait deux propositions, plus complémentaires que concurrentes. La première proposait de décrire le comportement analogique grâce à un ensemble d'équations différentielles non ordonnées dont le nombre dépendrait du nombre de connexions du composant, du nombre de ses noeuds internes et du nombre de ses états (cf. plus loin). Quant à l'autre, il s'agissait d'une description séquentielle qui permettrait de calculer soit les courants des différentes connexions en fonction des tensions, soit le contraire. Ainsi, les champs d'une pin ou d'un noeud n'ont pas tous les deux le même statut : l'un doit être en lecture et l'autre en écriture au sein d'une même description séquentielle.

Cette dernière solution est proche de ce qui est intégré, entre autres dans les simulateurs électriques fonctionnant avec la **méthode nodale modifiée**, pour laquelle il est préférable de calculer les courants en fonction des tensions. Cependant, pour certains composants, il est difficile de trouver une telle formule explicite et donc dans ce cas la première solution est utilisée, c'est-à-dire donner directement les équations à résoudre.

En fait, on se dirige vers une instruction composée de deux parties: une procédurale et une autre composée d'équations. La partie procédurale et la partie comprenant les équations pourront être divisées en plusieurs zones spécifiques au domaine d'interprétation. De plus dans la partie procédurale, il pourra y avoir une quatrième partie pour l'initialisation.

```
equation-set-statement ::=
    equation-set_label : equation-set
        { equation-set_declarative_item }
    { procedural [ for domain_name_list ] =>
        { analog-sequential-statements } }
    { equation [ for domain_name_list ] (liste_d_inconnues)
        { equation_statement } }
    end equation-set [ equation-set_label ] ;

equation_statement ::=
    equation_label : literal_expression == literal_expression ;
    | if condition then
```

```
        { equation_statement }
      [[elsif
        { equation_statement } ]
       else
        { equation_statement } ]
       end if;

domain_name ::= init | dc | tran | ac
```

Les équations sont deux expressions numériques séparées par le symbole == d'égalité, et qui possèdent une étiquette non facultative. Le nombre d'équations devra être le même que le nombre d'inconnues présentes dans la liste d'inconnues. Pourront être pris comme inconnues : le champ tension ou courant d'un node ou d'une pin, peut-être les variables, ou bien un objet qui sera présenté plus loin : la **quantity** ou son formel le **coupling**. Dans le cas d'une variable prise comme inconnue, on ne pourra pas accéder à sa dérivée ou à son intégrale, elles devront être solutions d'équations implicites. Dans cette partie, on pourra utiliser des instructions if-then-elsif-else, à condition de faire apparaître dans chacune des branches le même nombre d'équations avec les mêmes étiquettes. D'autre part, les différents blocs d'équations associés à différents domaines d'interprétation devront avoir la même liste d'inconnues.

La partie procédurale, quant à elle, contiendra toutes les instructions séquentielles VHDL exceptée l'affectation de signal et la mise en attente. En effet, une partie analogique ne pourra pas directement interférer dans une partie digitale. Par contre, on aura accès en lecture à tous les signaux visibles.

6. La notion de paramètres variables - LES D

En analogique, il arrive fréquemment qu'une grandeur d'un système contrôle la valeur d'un paramètre d'un autre composant. On peut citer en exemple la source de tension commandée par une différence de potentiel. Pour permettre une telle représentation, l'actuelle clause générique ne suffit pas. En effet, les paramètres définis dans celle-ci sont constants au cours d'une simulation, alors que dans le cas présent, le paramètre varie au cours de la simulation. Une nouvelle clause est donc nécessaire; elle a été proposée et s'appelle **coupling**.

Avec la clause pin et coupling en plus la déclaration d'entité aura la forme suivante :

```
entity_declaration ::=
```

```
    entity nom_entity is
        [ generic ( generic_list ) ; ]
        [ coupling ( coupling_list ) ; ]
        [ port ( port_list ) ; ]
        [ pin ( pin_list ) ; ]
        { entity_declarative_item }
    [ begin
        entity_statement_part ]
    end [ nom_entity ];

entity vcvs is
    generic (gain : float := 5.0);
    coupling (v0 : float [voltage]);
    pin (p, n : electrical);
end;
```

Ces couplings sont considérés comme des objets quantités (**quantity**) (cf plus loin) dont ils sont les objets formels, avec toutes les propriétés qui leurs sont attachées. D'autre part, ils sont connectés au travers d'une clause **coupling map**, de même que des génériques ou des ports. Ces objets peuvent être attachés soit à un autre coupling, soit à une quantité (tension d'un noeud, courant d'une branche, quantité).

En fait, au vu de la description, on peut associer un mode implicite à chaque coupling (in, out ou inout), tout comme les ports. Cependant, il ne peut pas y avoir plusieurs sources pour un coupling, mais il en faut nécessairement une. Toutes ces vérifications devront être faites par l'analyseur.

7. La vérification des dimensions - LES O

Les équations manipulées en physique possèdent une dimension qui est donnée par une combinaison linéaire de cinq dimensions de base :

- la longueur [L] se mesurant en mètre,
- le temps [T] en seconde,
- la masse [M] en kilogramme,
- la température [K] en kelvin,
- le courant [A] en ampère.

Pour de plus amples détails, on peut regarder un précis (handbook) de physique : par exemple [Con67] dans l'annexe A. En utilisant ces notations, voici une courte liste des principales unités utilisées en électricité et de leur dimension :

- le coulomb [AT]
- le volt [$ML^2A^{-1}T^{-3}$]
- la résistance [$ML^2A^{-2}T^{-3}$]
- la capacité [$ML^{-2}A^2T^4$]
- l'inductance [$ML^2A^{-2}T^{-2}$]
- le watt [ML^2T^{-3}]

Lorsqu'on effectue une série de manipulations d'équations en physique, il peut être utile de vérifier la dimension des termes pour éviter les étourderies. Il semble donc normal qu'un langage de modélisation analogique manipulant des valeurs dimensionnées puisse vérifier la cohérence des expressions utilisées. Les unités physiques telles qu'elles existent actuellement dans VHDL ne permettent pas d'effectuer cette vérification, aussi faut-il envisager une redéfinition de celles-ci, ou tout simplement ajouter une nouvelle notion au langage.

a. Une solution

La solution la plus simple consiste à définir un nouveau type auquel est attachée une unité. Chaque objet de ce type aurait donc une unité. De plus, un constructeur d'unité permettrait d'en définir de nouvelles à partir des unités existantes. Un paquetage standard devra contenir la définition des cinq unités de base.

```
variable i : analog unit is ampere;
variable v : analog unit is volt;
-- ou
variable v : analog unit is kilogram*meter**2/coulomb/second**2;


unit newton is kilogram*meter**2/second;
unit poundal is 0,13825 newton;   -- force nécessaire pour donner une
                                  -- accélération de un inch/sec/sec
                                  -- à un pound !!
unit inch is 2.5400051 centi meter;
unit foot is 12.0 inch;
```

Cette solution présente l'avantage d'être relativement simple et de proposer un nouveau type pour les grandeurs analogiques. Cependant, elle mélange deux notions en une seule : dimension et unité.

b. L'autre solution

C'est pourquoi il a été envisagé une seconde solution plus complète. Ainsi, chaque objet dont le type est continu : **float, complex** ou **analog**, pourra se voir associer une dimension. Ainsi ses valeurs auront une unité compatible avec celle-ci.

```
variable v : real [voltage] := 2.5 volt;
```

Il faut donc deux constructeurs : un pour les dimensions et un pour les unités.

```
dimension force is mass*length**2/duration;
unit newton dimension force;
unit poundal is 0,13825 newton;   -- force nécessaire pour donner une
                                  -- accélération de un inch/sec/sec
                                  -- à un pound !!
unit inch is 2.5400051e-2 meter;
unit foot is 12.0 inch;
```

Là aussi, il faudra avoir la définition des cinq dimensions de base avec leur cinq unités respectives dans un paquetage standard. Pour les exemples nous choisirons cette dernière solution.

Dans tous les cas, les objets auront des dimensions ou unités statiques. Cela implique que les vérifications pourront être faites entièrement à l'analyse et cela n'entraînera pas de perte de temps à l'exécution.

8. Une nouvelle classe d'objet : les états (quantity) - LES B & C & E

L'étude de systèmes analogiques introduit la notion d'états du système, par exemple la charge d'un condensateur ou encore le courant d'une inductance. Ceux-ci entrent en jeu lors de la résolution d'équations différentielles. De plus, pour résoudre une équation différentielle il faut connaître un certain nombre de conditions initiales, ce nombre dépendant de l'ordre de l'équation qui est aussi celui du système. Ainsi un condensateur ou une inductance sont des systèmes d'ordre un et possèdent un seul état chacun.

Pour décrire un système d'ordre n de manière explicite, on a besoin de stocker ses états. Ce sont de nouveaux objets qui devront avoir les caractéristiques suivantes :

- possibilité de conserver l'historique,

- possibilité de lui donner un futur (forme d'onde), existence d'un unique "pilote",
- possibilité d'être considérés comme inconnues par le simulateur s'ils n'apparaissent pas à gauche dans une affectation et s'ils se trouvent dans une équation,
- valeur initiale par défaut égale à 0,

L'accès aux anciennes valeurs se fera grâce à un nouvel attribut, par exemple **'old**. Cet attribut pourra avoir un paramètre qui indiquera à quel pas de calcul antérieur on veut connaître la valeur. Ainsi **'old**(1) fait référence à la valeur précédente, et **'old**(N) à la valeur d'il y a N pas de calcul. De plus, ce paramètre subira les mêmes contraintes que ceux de certains attributs existants, c'est à dire il devra être localement statique; et s'il est omis cela sera équivalent à **'old**(1). Cet attribut est à rajouter à la fonction **tstep**(N) qui permet de connaître le temps écoulé depuis le $N^{ième}$ pas de calculs antérieurs. Cette fonction sera présentée plus loin. Quant à la valeur de retour, elle sera du même type que l'objet. De plus, cet attribut ne sera pas une quantité mais une fonction. On ne pourra donc pas faire **S'old'old**.

La possibilité de proposer le futur d'une quantité est proche de l'affectation d'une forme d'onde à un signal en VHDL actuel. Dans ce cas la forme d'onde associée devra être continue. Cette solution permettra de faciliter la modélisation de réponses des sources commandées. Par exemple pour l'interface numérique-analogique il est facile de coder la réponse de la source commandée sous la forme d'une fonction linéaire par morceau (PWL). Actuellement, la notion de multi-sources comme pour un signal numérique, et de ce fait l'utilisation de fonction de résolution n'a pas été envisagée pour la quantité . Il semble, d'ailleurs, que cela ne présente pas un grand intérêt.

Voici quelques exemples d'écriture de formes d'onde. Le temps donné après le mot clé **in** est ici une durée relative au temps de la simulation. On a alors une syntaxe proche de l'affectation de signal.

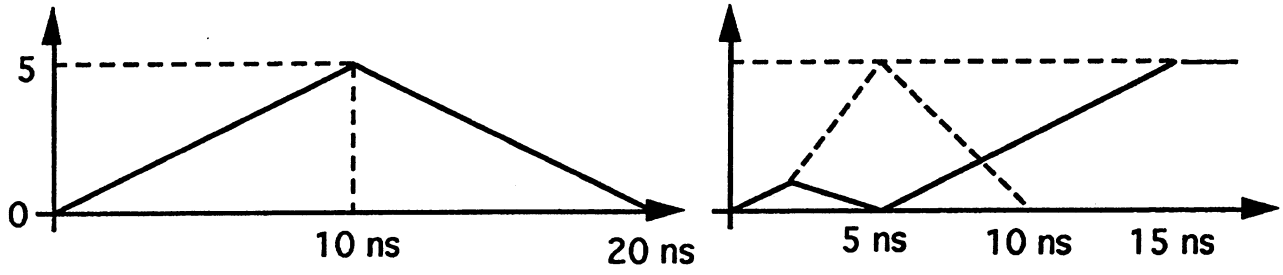
```
q <= q1 in t1, q2 in t2, …, qn in tn;
```

Voyons maintenant quel pourrait être le comportement de cette instruction, tout d'abord une affectation de deux états q1 et q2 dans une même partie procédurale mais en deux endroits différents :

```
q1 <= 0.0, 5.0 in 10.0e-9;
```

```
...
-- complétée dans le même pas de calcul par
q1 <= 0.0 in 20.0e-9;
...
q2 <= 0.0, 1.0 in 2.0e-9, 5.0 in 10.0e-9, 0.0 in 10.0e-9;
...
-- modifiée dans le même pas de calcul par
q2 <= 0.0 in 5.0e-9, 5.0 in 10.0e-9;
```

Pour q1, la seconde affectation porte sur un temps supérieur au temps maximum déja contenu par son pilote : donc on ajoute cette forme d'onde à la précédente. Par contre pour q2, la seconde forme d'onde vient écraser une partie de la précédente car les temps se chevauchent. On obtient alors les figures suivantes, où le trait pointillé représente la forme d'onde écrasée :

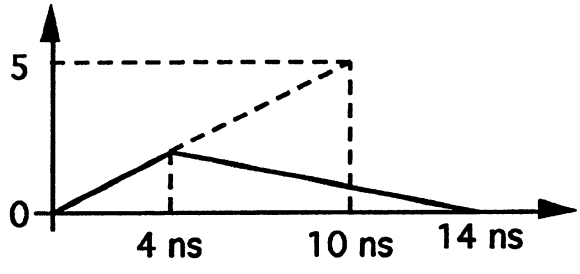


à gauche q1 et à droite q2

Dans l'exemple suivant, l'état q est affecté à deux pas de calcul différents, donc la seconde affectation écrase la forme d'onde déja présente :

```
q <= 0.0, 5.0 in 10.0e-9;

-- modifiée dans un pas de calcul suivant 4ns plus tard en
q <= 0.0 in 10.0e-9;
```

On obtient alors le schéma suivant :



Enfin, comme on l'a vu précédemment, les deux champs d'une pin ou encore d'un noeud interne sont des états. En effet, il faut pouvoir accéder à leur passé et aussi pouvoir leur affecter une forme d'onde.

9. Les opérateurs

Il est inconcevable de pouvoir décrire le comportement d'un modèle analogique sans avoir accès à la dérivée par rapport au temps de ses états (quantités). Evidemment lors de la simulation la valeur de la dérivée sera approximée par une formule choisie par le simulateur grâce aux différentes valeurs passées et aux valeurs du pas de calcul.

Il faut quand même noter, que lors d'une analyse en fréquence, la dérivée par rapport au temps est calculée exactement. En effet cela correspond à une division par la variable complexe p ou plus simplement par $j\omega$.

D'autre part, l'opérateur d'intégration par rapport au temps pourra dans certains cas être utile, par exemple si on veut exprimer la valeur du courant traversant une self-inductance en fonction de la différence de potentiel à ses bornes. Les paramètres nécessaires à cet opérateur sont :

- la date de départ de l'intégration (on suppose que la date finale est le temps du pas de calcul courant),
- la valeur initiale de la valeur intégrée,
- la quantité intégrée.

Ces opérateurs pourront être utilisés indifféremment dans un bloc procédural ou un bloc d'équations. L'intégration ou la dérivation ne pourra se faire que sur un objet quantity. Ceci est lié à une spécificité du langage, en effet les paramètres des fonctions appartiennent à une classe d'objets, et s'ils sont de la classe quantité , il faut leur passer une quantité .

Ces deux opérateurs sont des fonctions VHDL particulières, en effet leur paramètre d'entrée ainsi que leur valeur de retour ne sont pas de type fixe bien qu'ils appartiennent à la classe des unités physiques. Néanmoins, il existe une relation entre le type d'entrée et celui de sortie. D'ailleurs il serait intéressant de laisser l'utilisateur avoir accès à l'écriture de telles fonctions, cela lui permettrait de définir sa propre fonction dérivée si celle qui est proposée par le simulateur ne lui convient pas ou s'il veut assurer la conservation des résultats en changeant de simulateur. L'écriture pourrait être de cette forme :

```
-- déclaration de la fonction
function my_d_dt(quantity x : in real generic [my_dimension])
    return real [my_dimension/duration];

...
-- corps de la fonction
function my_d_dt(quantity x : in real generic [my_dimension])
    return real [my_dimension/duration] is
    variable result : real [my_unit/second];
begin
    result := (x - x'old) / tstep;
    return (result);
end my_d_dt;
```

Encore une fois le travail demandé au compilateur n'est pas facile mais reste envisageable. C'est à l'appel de cette fonction que le type générique sera affecté, cependant la vérification de la formule reste statique.

10. Les interactions mixtes analogiques-numériques

a. Généralités

Dans une description il sera impossible de connecter un signal à une pin ou encore un noeud à un port. Les descriptions mixtes utiliseront soit des interfaces fictives (sans réalité physique), soit un véritable convertisseur Analogique-Numérique ou Numérique-Analogique; ces deux types d'entité possèderont des pins et des ports.

Les descriptions qui utiliseront les interfaces du premier type seront des descriptions de circuits numériques dont une partie (critique) est décrite au niveau électrique. Par contre les autres seront de vrais circuits mixtes qui contiendront des convertisseurs.

Aucun composant d'interface ne sera prédéfini dans le langage, tout devra être défini par l'utilisateur, ou dans des paquetages et bibliothèques standardisées. Ceci conserve l'esprit VHDL, où aucun composant prédéfini n'existe, même pas les portes logiques.

Une entité pourra avoir une architecture associée qui comporte des blocs digitaux et d'autres analogiques, même si au niveau de ses connexions seuls apparaissent des ports ou des pins. Les blocs digitaux pourront lire les valeurs analogiques visibles, en conservant l'esprit des règles de visibilité actuelles, c'est-à-dire les courants ou la tension des noeuds ou encore ceux des pins des

composants, mais ils ne pourront pas les modifier. Il en est de même pour les blocs analogiques vis-à-vis des blocs et des objets digitaux.

Actuellement, face à la lourdeur relative d'intégrer des composants d'interface pour l'interface mixte, il a été envisagé de pouvoir connecter directement des noeuds avec des ports et des signaux avec des pins. Cependant, cette alternative pose un problème au niveau du langage VHDL. En effet, comment clairement spécifier la connexion entre deux objets fondamentalement différents. C'est pour cela d'ailleurs, que ce mécanisme nous semble réfutable.

D'autre part, cela entraîne la définition d'un mécanisme implicite, puisque ces interfaces existeront bien pendant la simulation. Enfin, notons que rien n'empêche l'utilisation d'une saisie de schéma dans laquelle ces interfaces seront omises, mais qui apparaîtront dans le code VHDL généré.

b. Interactions Analogique-Numérique - LES M

Dans ce sens, des nodes, des pins ou encore des quantities vont être référencés dans un processus. Le concept de l'événement analogique est attaché à la notion de seuil. Un événement a lieu si la valeur analogique dépasse un seuil. Ce dépassement peut se faire soit en montant, soit en descendant.

Enfin, le dernier problème est la date de cet événement. En effet, le temps en analogique est continu et devra certainement être arrondi au temps de base supérieur pour être donné au numérique. Que se passera-t-il si deux événements antagonistes ont lieu à la même date numérique?

Le premier point consiste à trouver une syntaxe claire pour spécifier l'attente d'un événement analogique. Une première solution est d'utiliser ce qui existe :

```
wait until q > seuil;
```

Avec cette solution aucune sémantique n'impose le calcul de la date exacte du franchissement du seuil. D'autre part, quand cette condition doit-elle être évaluée et donc a-t-on le droit d'écrire :

```
wait on q;
```

Pour résoudre ce problème, il a été convenu de pouvoir associer à chaque objet quantity une ou plusieurs fonctions de type booléen. L'instruction précédente serait équivalente à :

```
function f1 (quantity q : float) return boolean;
function f2 (quantity q : float) return boolean;

quantity q : f1 f2 float;
...
wait on q until f1(q) or f2(q);
```

Cependant, avec cette solution, il reste toujours le problème de la date exacte du franchissement du seuil : à quelle date la condition est-elle passée de fausse à vraie? Pour l'instant, aucune solution n'a été trouvée.

Pour résoudre ce problème, il faut introduire une nouvelle sémantique pour l'instruction wait. Peut-être tout simplement :

```
wait threshold q > seuil or f1(q);
```

Bien entendu, on pourrait conserver la notion de fonctions associées à une quantity. Dans ce cas l'équivalence serait :

```
wait on q;
=>
wait threshold f1(q) or f2(q);
```

Une dernière remarque sur le cas d'une attente sur une quantity sans fonction associée. Dans ce cas, on considère que l'événement a lieu dès que q change, c'est à dire quasiment à chaque pas du simulateur analogique.

c. Interactions Numérique-Analogique

Dans ce sens la notion est plus simple. En effet, si une instruction equation-set accède à un signal, cela introduit un pas de calcul obligatoire pour le simulateur analogique lors d'un événement sur ce signal, il y aura eu alors un événement mixte.

11. Remarque

La description d'un circuit en VHDL analogique devra être utilisable par n'importe quel simulateur analogique, ou du moins une description devra pouvoir être écrite pour n'importe quel

simulateur. En effet, la simulation analogique utilise différentes mises en équations du problème. Par exemple la méthode nodale prend pour inconnues les tensions, et essaie de les trouver en appliquant la loi nodale : annuler la somme des courants en un noeud. Pour une telle mise en équation, un modèle procédural devra calculer la valeur des courants de ses connexions en fonction des tensions; c'est ce qui est fait en interne pour l'ensemble des composants de base.

La question est donc de savoir si un simulateur respectant la norme VHDL analogique devra accepter n'importe quelle description ou seulement certaines. Il est vraissemblable que c'est la première affirmation qui est la bonne. On en déduit donc que les simulateurs acceptant la norme VHDL devront faire un traitement interne pour rendre compatibles les différentes types de descriptions possibles avec leur(s) schéma(s) de résolution.

12. La définition d'un domaine - LES F

Un HDL analogique doit pouvoir décrire tous les (ou plutôt de nombreux) domaines de la physique. Ainsi, il est possible de décrire un système mécanique ou thermique en utilisant MAST. D'autre part les simulateurs analogiques n'ont pas de mal à simuler ces domaines.

Un système analogique est en contact avec un autre en des points qui se retrouvent dans la clause pin de VHDL analogique. Les pins sont donc des objets qui dépendent d'un domaine, mais qui dans tous les cas possèdent deux champs qui sont une grandeur extensive (across) et une grandeur intensive (through).

Pour différentes raisons, il semble raisonnable de créer une vérification plus stricte du type utilisé dans la clause pin. C'est pour cela que dans les spécifications du futur FAS, il est possible de définir un domaine. Un domaine est le type d'une valeur analogique. Pour le définir il faudra donner explicitement les deux champs respectifs (across et through) ainsi que leur unité. Voici une syntaxe possible :

```
type electrical is domain
    across v : physic unit is volt;
    through i : physic unit is ampere;
end domain;

type mechanical is domain
```

```
    across x : physic unit is meter;
    through f : physic unit is newton;
end domain;
```

Certes il est possible de faire sans. Mais cela permet d'introduire une vérification supplémentaire et assurerait l'utilisation d'une structure à deux champs dans la clause pin. D'autre part cela permet de différencier les deux champs de la structure.

Une solution existe dans la nouvelle norme VHDL'92, elle consiste à mettre des attributs à deux champs d'une structure. Cette dernière n'est pas aussi souple que l'introduction des mots clés domain, through et across, mais justement elle évite l'ajout de mots clés supplémentaires.

13. L'initialisation et le cycle de simulation - LES N

Le manuel de référence de VHDL-analogique spécifiera un cycle d'initialisation ainsi qu'un cycle de simulation. Bien entendu, rien n'empêchera l'implémentation de méthodes de synchronisation plus efficaces à condition toutefois que les résultats soient identiques à ceux de la méthode spécifiée dans le LRM.

C'est pour cette raison que les algorithmes spécifiés dans le LRM devront être le plus général possible. Dans tout les cas, il faut éviter la sur-spécification.

14. La notion de contribution en courant

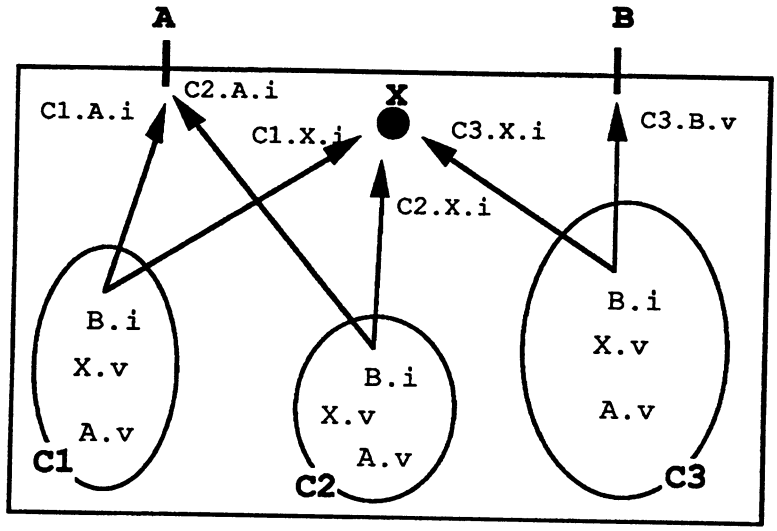
Actuellement, dans VHDL on peut trouver au sein d'une même architecture plusieurs blocs et plusieurs processus. Un bloc est une encapsulation de déclarations communes et locales à un groupe de processus. De plus, toute une hiérarchie de blocs peut être réalisée. Néanmoins tout ceci est parfaitement régi par le LRM.

De même qu'une architecture peut contenir plusieurs processus, elle pourra contenir plusieurs instructions **equation-set**.

Si plusieurs instructions equation-set se côtoient, il faut introduire la notion de branche de courant d'une pin ou d'un noeud. On ajoute une branche de courant si un bloc procédural affecte une valeur de courant à un noeud ou une pin. Mais, pour de telles descriptions, d'autres contraintes sont

nécessaires. Ainsi, la tension d'un noeud ou d'une pin étant unique, une seule instruction equation-set devra donc calculer sa valeur. Enfin, il faudra respecter la contrainte introduite pour les pins et les noeuds dans les intructions equation-set : si on lit le courant (respectivement la tension) on ne peut l'affecter, mais on peut affecter la tension (respectivement le courant) sans toutefois la lire.

Voilà un exemple d'un couple entité-architecture possible :



Ceci est la description d'un modèle à deux pins `A` et `B`, qui possède en outre un noeud interne `X`. La description est composée de trois instructions equation-set : `C1`, `C2` et `C3`. On voit que le courant sortant de `A` provient de deux endroits : `C1` et `C2`, donc deux branches de courant doivent être créées implicitement. La tension de `B`, quant à elle, est calculée dans `C3`. Et enfin le noeud `X` reçoit des contributions en courant de la part des trois parties et donc il possède trois branches de courant. La loi de Kirchoff appliquée à ces branches implique que la somme des trois courants soit nulle. Par contre, en A la somme des branches de courant n'est pas forcément nulle car A peut être connectée dans un bloc supérieur à un noeud sur lequel d'autres branches de courant existeraient.

La recherche des branches de courant d'un noeud ou d'une pin est similaire à celle des pilotes d'un signal ou d'un port (de mode out).

Les notations utilisées dans le schéma pour les contributions en courant sont juste indicatives. Par contre, certains pensent qu'il serait intéressant de pouvoir référencer telle ou telle valeur de courant de branche, il faudra donc trouver une notation. Comme les courants sont rangés dans une liste dans le node, on pourrait écrire `nom-du-node.label.i`.

15. Les sous-programmes- LES LL

La notion de sous-programme permet de structurer du code algorithmique et ainsi d'en augmenter la lisibilité. Cependant, comme nous l'avons vu, il y a aussi la notion d'appel de procédure concurrent qui permet de rapidement instancier des processus ayant la même structure avec des paramètres et sur des signaux différents.

Cette dernière notion est intéressante pour les phases de développement, pour conserver un certain parallèle entre le numérique et l'analogique, certains ont pensé introduire l'appel de procédure analogique concurrent. Cette fonctionalité a entraîné des bouleversements dans la structure de la procédure.

Dans un premier temps, un sous-programme possédera une caractéristique implicite : neutre, numérique ou analogique.

Une procédure sera numérique, si un de ses paramètres est un signal en mode out ou inout, ou s'il y a utilisation de l'instruction wait ou s'il appelle une procédure numérique. Dans ce cas, il ne pourra être utilisé que dans un processus, et son appel concurrent générera un processus.

Une procédure sera analogique si un de ses paramètres est une quantity en mode out ou inout ou s'il appelle une procédure analogique. Dans ce cas, sa structure pourra être la même que celle d'une instruction equation-set , avec une partie réservée à des équations. Si ces équations ne dépendent pas du temps, alors la procédure pourra être appelée dans le code séquentiel ou concurrent, sinon uniquement dans le monde concurrent.

Bien entendu, une procédure analogique ne pourra en aucun cas appeler une procédure numérique, de la même manière, une procédure numérique ne pourra pas appeler une procédure analogique. D'autre part, une procédure ne pourrra pas avoir un paramètre de classe signal en mode inout ou out, et à la fois un autre de classe quantité en mode inout ou out.

Une procédure ni analogique ni numérique est appelée neutre. Une fonction sera toujours de mode neutre.

16. Questions en suspens

a. Les sources

Comme nous l'avons vu précédemment, pour le formalisme de la mise en équation de la méthode nodale modifiée, il est important d'isoler les sources de tension.

Dans une description comportementale il n'est pas évident de les reconnaître. Aussi dans le langage FAS les connexions possèdent un statut, qui est par défaut source de courant et qui peut être modifié en source de tension.

D'autre part, il est impossible pour une connexion de changer de statut au cours d'une simulation. Car la mise en équation est statique, et il est impossible de changer le nombre d'inconnues en cours d'exécution.

En fait il faut être sûr de pouvoir, à partir de la description comportementale, extraire l'information suffisante pour isoler les pins sources de tension et interdire le changement de statut. Encore une fois, cette vérification semble du domaine du possible, quoiqu'elle puisse être assez compliquée, puisque l'information existe dans le code source. Dans le cas contraire il faut introduire une déclaration de statut comme dans FAS.

b. De l'utilité de fonctions systèmes

Dans VHDL il est possible d'accéder au temps courant de simulation : c'est la fonction **now** qui retourne le temps en type **time**. Cette notion de fonction pré-définie, qui est équivalente à une variable globale, est utilisée dans le langage MAST pour fournir le temps de simulation, ou encore la durée des pas de calcul. Il est de plus possible de donner une heure de rappel à laquelle le simulateur devra obligatoirement s'arrêter.

L'utilisation des quantités permet d'accéder à l'historique de la simulation, mais si on ne possède pas la date des pas de calcul cela ne sert à rien. C'est pour cette raison que la fonction tstep doit être présente. Comme pour l'attribut **'old**, c'est une fonction dont le paramètre est un entier naturel qui indique l'antériorité du pas. Ainsi **tstep**(n) retourne le temps écoulé depuis n pas de calcul. Le type de

retour est une unité physique en secondes. On peut donc l'utiliser dans les expressions analogiques contrairement à la fonction **now**.

Enfin, il est utile de pouvoir fixer une heure de réveil pour être sûr que le simulateur effectue un pas à une date précise pour simuler un événement : début ou fin d'un créneau par exemple. On peut introduire la fonction **timeout** dont le paramètre est une durée de type time. Une autre solution est de se servir de la synchronisation mixte, en faisant lire du côté analogique un signal dont les changements corrrespondent aux dates de cassure.

Une solution alternative serait de pouvoir affecter une forme d'onde (waveform) aux états ou aux valeurs des courants ou des tensions.

IV. Exemples

1. Présentation

Le langage VHDL-analogique n'étant pas encore standardisé, la syntaxe proposée ici n'est que provisoire. D'ailleurs dans certains exemples plusieurs syntaxes différentes seront présentées.

2. Exemple 1 : les interfaces mixtes

Les deux interfaces présentées sont des composants sans réalité physique. Elles ne sont que des boîtes de traduction d'un niveau à un autre.

```
entity interface_ad is
    generic (t01, t10 : time;
             s01, s10 : float [voltage]);
    port     (d : out bit);
    pin      (a, ref : electrical);
end interface_ad;

architecture a of interface_ad is
begin
    p1 : process
    begin
        wait until ((a.v - ref.v) > s01);
        d <= '1' after t01;
        wait until ((a.v - ref.v) < s10);
        d <= '0' after t10;
    end process p1;
```

```
    a1 : equation-set
    begin
        a.i := 0.0 [ampere];     -- impédance d'entrée infinie
        ref.i := 0.0 [ampere];
    end equation-set a1;
end a;
```

Pour cette interface, on peut noter le bloc analogique présent pour indiquer que l'impédance est infinie. Il serait intéressant de pouvoir omettre une telle partie et de considérer par défaut ce comportement. D'autre part, les délais sont dans le type **time** et non en unité physique, car ils sont utilisés comme paramètre des **after**. L'utilisation de la valeur analogique A.v dans le processus numérique implique que dès que A.v est modifié (d'un pourcentage non négligeable) alors un événement est généré pour évaluer la condition. Il s'agit d'une interaction mixte.

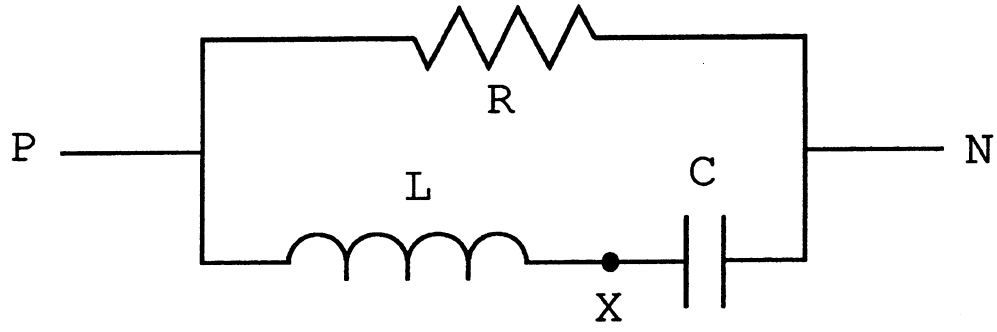
```
entity interface_da is
    generic (t01, t10 : float [duration];
             v1, v0 : float [voltage]);
    port     (d : in bit);
    pin      (a, ref : electrical);
end interface_da;

architecture a of interface_da is

begin
    a1 : equation-set
    procedural
        if (not(d'stable) or (now = 0 ns)) then
            -- un événement sur d ou initialisation
            if (d = '1') then
                a.v <= v1 in t01;
            else
                a.v <= v0 in t10;
            end if;
        end if;
    end equation-set a1;
end a;
```

Celle-ci est légèrement plus complexe, surtout au niveau de l'interaction mixte qui est plus fine et moins explicite. L'architecture est composée d'une seule instruction **equation-set**. Le fait de lire le signal D dans la partie analogique, implique que lorsqu'un événement va survenir sur D, le simulateur analogique va devoir faire un pas de calcul à cette date.

L'utilisation de l'affectation par forme d'onde rend l'écriture du front analogique très facile et intuitive.

3. Exemple 2 : un circuit RLC

Dans cette partie, nous allons modéliser de différentes manières une résistance en parallèle avec un condensateur et une inductance. Cet exemple va mettre en évidence l'utilisation des deux manières différentes qu'il existe entre autre, de décrire un comportement électrique. Toutefois, ce sous-circuit se décrit facilement de manière structurelle. C'est pour cette raison qu'une description structurelle sera donnée en premier, cela permettra aussi de voir que la description de structures électriques est en fait très proche de ce qui existe actuellement en VHDL.



La déclaration d'entité va être commune à toutes les différentes architectures proposées ensuite. C'est pour cela qu'elle est présentée maintenant :

```
entity rlc is
    generic ( r : float [resistance] := 100.0 ohm;
              l : float [inductance] := 1.0e-3 henry;
              c : float [capacitance] := 1.0e-9 farad);
    pin(p, n : electrical);
end rlc;
```

a. Description structurelle

```
library basic_analog_dev;
architecture structural of rlc is

    component inductor is
        generic (l : float [inductance]);
        pin (p, n : electrical);
    end  component;

    component resistor is
```

```
        generic (r : float [resistance])
        pin (p, n : electrical);
    end  component;

    component capacitor is
        generic (c : float [capacitance])
        pin (p, n : electrical);
    end  component;

    for r1 : resistor
        use entity basic_analog_dev.resistor(functional);
    for l1 : inductor
        use entity basic_analog_dev.inductor(functional);
    for c1 : capacitor
        use entity basic_analog_dev.capacitor(functional);

    node x : electrical;

begin

    r1 : resistor
        generic map (r => r)
        pin map (p => p, n => n);

    l1 : inductor
        generic map (l => l)
        pin map (p => p, n => x);

    c1 : capacitor
        generic map (c => c)
        pin map (p => x, n => n);

end structural;
```

Pour quelqu'un qui connait VHDL, il n'y a aucun problème; la forme syntaxique est conservée. On peut noter la déclaration du noeud intermédiaire, qui sert dans la connectique comme servirait un signal en numérique.

```
    node x : electrical;
```

Ce noeud n'est visible que dans le corps de l'architecture. Le simulateur le traitera cependant comme les autres noeuds, c'est-à-dire qu'il entrera dans la méthode de mise en équation. Par exemple pour un simulateur utilisant la loi nodale, une nouvelle ligne sera incluse dans le jacobien, et au cours de la simulation la somme des courants en X sera nulle. Pour le cas présent le simulateur cherchera les valeurs de potentiel qui rendent égaux les courants traversant la capacité et l'inductance.

b. Description comportementale

```
architecture behavioral of rlc is

    node x : electrical;

begin

    equation-set
        variable ir, ic : float [current];
        -- il a une valeur d'initialisation
        -- pour sa condition initiale
        quantity il : float [current] := 0.0 [ampere];
    procedural
        -- loi d'ohm pour le courant traversant le résistor
        ir := (p.v-n.v)/r;

        -- ic = c du/dt
        ic := c*(d_dt(x.v)-d_dt(n.v));

        -- approximation de l'intégrale il = ∫ u/l dt
        il := ((p.v-x.v)+(p.v'old-x.v'old))*tstep(1) / 2.0 / l +
              il'old;

        -- pour ce cas simple, il = ic
        x.i := il - ic;
        p.i := -ir - il;
        n.i := ir + ic;
    end  equation-set

end behavioral;
```

Le but d'une telle description est de calculer explicitement la valeur des courants arrivant à chaque noeud. Donc, elle sera optimale pour un simulateur utilisant la loi nodale.

Détaillons les points importants de cette description.

Pour le calcul du courant traversant la capacité, il faut noter l'utilisation de l'opérateur dérivée. Par contre pour celui traversant l'inductance, on a préféré donner explicitement l'approximation de l'intégrale. En fait on pourrait inverser cela, c'est-à-dire calculer explicitement la dérivée et utiliser l'opérateur d'intégration. Dans ce cas, il aurait fallu une nouvelle **quantity** pour stocker la différence de potentiel. On aurait alors la description équivalente suivante :

```
        quantity diff : float [voltage];
    ...
        ic := c * ((p.v-x.v)-(p.v'old-x.v'old))/tstep(1);
        diff := p.v - n.v;
        il := integ(diff, 0)/l;
```

Enfin, on accède au courant d'une pin ou d'un noeud en utilisant la notation pointée comme pour une structure. Peut-être pourrait on trouver une nouvelle écriture pour éviter les deux dernière lignes ou encore le calcul de la différence de potentiel. On peut imaginer les notations suivantes :

```
...
ir := (p,n).v/r;
...
(p->n).i := ir + ic;
...
```

c. Avec des équations

```
architecture behavioral_2 of rlc is

    node x : electrical;

begin

    equation-set
        quantity il : float [current];
    equation (il, x.v, n.v, p.v)
        e1 : (p.v-n.v)/r + n.i + il == 0.0 [ampere]
        e2 : n.i + p.i == 0.0 [ampere];
        e3 : (p.v - x.v) - l*d_dt(il) == 0.0 [volt];
        e4 : il - c*(d_dt(x.v)-d_dt(n.v)) == 0.0 [ampere];
    end  equation-set;

end behavioral_2;
```

Le nombre d'équations à fournir est égal au nombre de pins augmenté du nombre d'états et de noeuds.

d. En utilisant les deux

Dans cet exemple, le courant traversant la résistance se calcule aisément avec la loi d'ohm; par contre le courant de l'autre branche est plus délicat à calculer de manière explicite. C'est pour cela qu'il serait bon de pouvoir décrire une partie de ce modèle avec des équations et l'autre avec une partie procédurale. De plus, en utilisant cette méthode on n'est plus obligé d'introduire un nouveau noeud.

```
architecture behavioral_3 of rlc is
```

```
begin

    equation-set
        variable ir : float [current];
        quantity il : float [current];
    procedural
        -- loi d'ohm pour le courant traversant le résistor
        ir := (p.v-n.v)/r;

        p.i := -ir - il;
        n.i := ir + il;
    equation (il)
        el : l*d_dt(il) + integ(il,il'old,time-tstep(1),time)/c -
             p.v + n.v == 0.0 [volt];
    end  equation-set

end behavioral_3;
```

La variable à résoudre est l'état IL. On peut quand même accéder à sa valeur dans la partie procédurale.

4. Exemple 3 : un transistor MOS

Commençons immédiatement par le code, à l'entité sont attachées deux architectures, une plus complète que l'autre.

```
entity nmos is
    generic (vt : float [voltage];
             w, l : float [length];
             mu0 : float [length**2/voltage/second]
             cox : float [capacitance/length**2];

            -- paramètres facultatifs : valeur par défaut
             cgs : float [capacitance] := 1.0 pico farad;
             cgd : float [capacitance] := 0.5 pico farad;
             va  : float [voltage] := 15.0 volt)
        -- on ne prend pas en compte le substrat (bulk)
    pin      (g, d, s : electrical);
end nmos;

architecture simple of nmos is
    -- description très simple ou d et s ne sont pas permutables
begin
    equation-set
        variable vgs, vgd, vds : float [voltage];
        constant mu0coxw_l : float [capacitance/duration/voltage]
                 := mu0*cox*w/l;
    procedural
        g.i := 0 [ampere];  -- impedance de la grille infinie
        vgd := (g,d).v;
        vgs := (g,s).v;
        vds := (d,s).v;

        if (vgs < vt) then
```

```
            (d->s).i := 0 [ampere];
        elsif (vds < vgs - vt) then
            (d->s).i := mu0coxw_l*((vgs-vt)-vds/2)*vds*(1+vds/va);
        else
            (d->s).i := mu0coxw_l*(vgs-vt)**2*(1+vds/va)/2;
        end if;
    end equation-set;
end simple;
```

Cette première architecture est vraiment très simple, le transistor ne possède pas de couplage entre la grille, la source et le drain. De plus les équations utilisées sont des formules du premier ordre. Ce modèle peut être utilisé pour des circuits numériques, mais pour des montages analogiques il ne présente pas assez de détails. Enfin, la source et le drain ne sont pas permutables.

```
architecture moins_simple of nmos is

    component capacitor is
        generic (c : float [capacitance])
        pin (p1, n1 : electrical);
    end component;

begin
    -- capacité grille-drain
    cgd : capacitor
          generic map (cgd)
          pin map (p1 => g, n1 => d);
    -- capacité grille-source
    cgs : capacitor
          generic map (cgs)
          pin map (p1 => g, n1 => s);

    equation-set
        variable vgs, vgd, vds : float [voltage];
        constant mu0coxw_l : float [capacitance/duration/voltage]
                 mu0*cox*w/l;
    procedural
        g.i := 0 [ampere];  -- impedance de la grille infinie
        vgd := (g,d).v;
        vgs := (g,s).v;
        vds := (d,s).v;

        if (vgs < vt) then
            (d->s).i := 0 [ampere];
        elsif (vds < vgs - vt) then
            (d->s).i := mu0coxw_l*((vgs-vt)-vds/2)*vds*(1+vds/va);
        else
            (d->s).i := mu0coxw_l*(vgs-vt)**2*(1+vds/va)/2;
        end if;
    end procedural;
end moins_simple;
```

Cet exemple nous montre la manière de mélanger la description structurelle avec la description

comportementale.

Dans la partie comportementale, le courant de la grille est mis à zéro, il s'agit en fait de la contribution de cette partie au courant de grille. Ainsi, à cause des deux capacités de grille le courant n'est pas nul. Il en est de même pour le courant drain-source qui n'est pas égal à la formule donnée dans la partie procédurale.

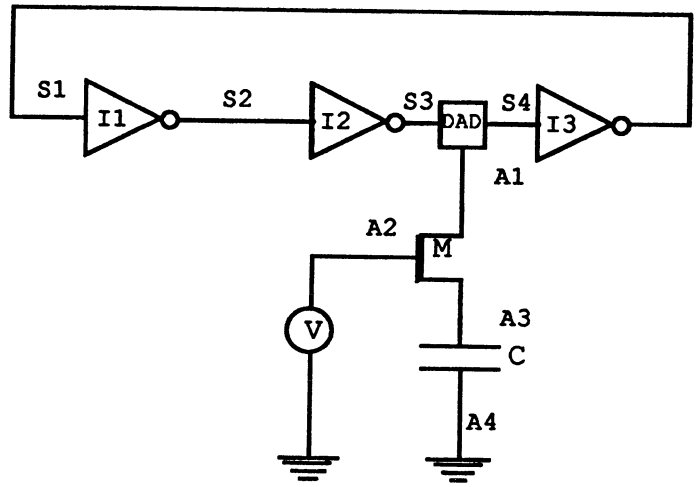
Et voila maintenant à quoi est équivalent l'instantiation de la capacité grille-drain. Cela est équivalent à ce que propose le LRM 87 page 9-11 pour le numérique. Le même bloc sera inséré pour l'autre capacité. La description de la capacité se trouve dans l'exemple suivant.

```
...
    cgd : block
        generic (c : float [capacitance]);
        generic map (c => cgd);
        pin (p1, n1 : electrical);
        pin map (p1 => g, n1 => d);
    begin
        capacite : block
            generic (c : float [capacitance]);
            generic map (c => c);
            pin (p, n : electrical);
            pin map (p => p1, n1 => n);
        begin
            equation-set
            procedural
                (p->n).i := c*d_dt((p,n).v);
            end equation-set;
        end block capacite;
    end block cgd;
...
```

Dans ce bloc équivalent on voit apparaître la contribution en courant de la capacité, qui sera rajouté à la liste des courants de branches des pins g et d.

5. Exemple 4 : l'oscillateur à fréquence contrôlée

Il s'agit de l'exemple qui a été présenté dans le paragraphe sur l'interface mixte. On va proposer une description structurelle dont les composants seront configurés dans une unité de configuration.




```
entity oscillateur is
end oscillateur;

architecture a of oscillateur is

    component mos
        pin (g, d, s : electrical);
    end component;

    component capacite
        generic (c : float [capacitance]);
        pin (p, n : electrical);
    end component;

    component interface
        port     (entree : in bit;
                  sortie : out bit);
        pin      (a, ref : electrical);
    end component;

    component inverseur
        generic (delai : time);
        port     (entree : in bit;
                  sortie : out bit);
    end component;

    component source
        port     (p, n : electrical);
    end component;

    signal s1, s2, s3, s4 : bit;
    node a1, a2, a3, a4 : electrical;

begin

    i1 : inverseur
        port map (s1, s2);

    i2 : inverseur
        port map (s2, s3);

    i3 : inverseur
        port map (s4, s1);
```

```
    dad : interface
        port map (entree => s3, sortie => s4)
        pin map  (a => a1, ref => a4);

    m : mos
        pin map (g => a2, d => a1, s => a3);

    v : source
        pin map (p => a2, n => a4);

    c : capacite
        generic map (10e-12 farad)
        pin map (a3, a4);
end a;
```

Maintenant, il faut décrire les feuilles de base. La description VHDL permet de condenser l'interface en un seul composant à trois connexions, plutôt qu'en deux.

```
entity interface is
    generic (i1, i0 : float [current];
             r1, r0 : float [resistance];
             c1, c0 : float [capacitance];
             t01, t10 : time;
             s01, s10 : float [voltage])
    port     (e : in bit;
              s : out bit);
    pin      (a, ref : electrical);
end;

architecture a of interface is

begin
    p1 : process
    begin
        wait until ((a, ref).v > s01);
        s <= '1' after t01;
        wait until ((a, ref).v < s10);
        s <= '0' after t10;
    end process p1;

    a1 : equation-set
        variable v : physic unit is volt;
    begin
        v := (a, ref).v;
        if (e = '1') then
            (a->ref).i := i1 + r1/v + c1*d_dt(v);
        else
            (a->ref).i := i0 + r0/v + c0*d_dt(v);
        end if;
    end equation-set a1;
end a;
```

L'interface modélise du côté analogique la charge d'entrée de l'inverseur. Le temps d'établissement varie donc en fonction de la charge analogique, cela est nécessaire pour donner des

résultats de simulation corrects. Dans la partie procédurale analogique le courant calculé peut subir des discontinuités, dans ce cas le simulateur doit tenir compte d'une dérivée maximum pour les valeurs calculées. Cette valeur de la dérivée sera peut être intégrée au langage sous le nom de fonction de pente. Ces fonctions pourront être très utiles et simplifieront l'écriture des modèles dont la tension ou le courant passe d'un état bas à un état haut.

Voici, la description de l'inverseur (très classique) :

```
entity inverseur is
    generic (delai : time);
    port     (e : in bit;
              s : out bit);
end;

architecture comportementale of inverseur is

begin
    s <= not(e) after delai;
end comportementale;
```

Et maintenant les descriptions analogiques, d'abord la description de la capacité suivie de celle de la source qui sert de stimuli à la simulation :

```
entity capacite is
    generic (c : float [capacitance]);
    pin      (p, n : electrical);
end;

architecture procedurale of capacite is

begin
    equation-set
    procedural
        (p->n).i := c*d_dt((p,n).v);
    end  procedural;
end procedurale;

------
entity source is
    generic (v : array (positive range <>) of float [voltage];
             t : array (positive range <>) of float [duration]);
    pin      (p, n : electrical);
begin
    assert (v'length <> t'length)
        report "attention, pas le même nombre de v et de t"
        severity error;
end;

architecture a of source is
```

```
begin

    equation-set
        variable n : positive := 1;
        variable h : float [duration] := 0.0 [second];
        variable slew : float [voltage/duration];
    procedural for init =>
        for i in v'range loop
            (p,n).v <= v(i) in t(i);
        end loop;
    end  equation-set;

end a;
```

Grâce à l'affectation d'une forme d'onde continue la description de la source linéaire par morceau est très simple, il suffit à l'initialisation de remplir le pilote des **nodes.**

Enfin, voilà l'unité de configuration qui va configurer les composants et donner une valeur aux paramètres génériques :

```
configuration a of oscillateur is
for a
    for i1, i2, i3 : inverseur
        use entity work.inverseur(comportementale)
            generic map (delai <= 10 ns);
    end  for

    for dad: interface
        use entity work.interface(a)
        generic  map  (i1 => 5.0 milli ampere, i0 => 0.0 ampere,
                       r1 => 5 kilo ohm, r0 => 5.0 kilo ohm,
                       c1 => 10.0 pico farad, c0 => 9.0 pico farad,
                       t01 => 1 ns, t10 => 1 ns,
                       s01 => 3.0 volt, s10 => 2.0 volt);
    end  for;

    for v : source
        use entity work.source(a)
        generic  map  ((0.0 volt, 0.0 volt, 5.0 volt),
                       (0.0 second,
                        0.3 micro second,
                        0.31 micro second));
    end  for;

    for c : capacite
        use entity capacite(procedurale);
    end for;

    for m : mos
        use entity tmos(moins_simple)
        generic  map  (w => 10.0 micro meter;
                       l => 10.0 micro meter;
                       mu0 => 5.8 centi meter**2/volt/second;
                       cox => 4.31 micro farad/meter**2)
        for cgd, cgs : capacitor
```

```
            use entity capacite(procedurale);
        end for;
    end for;
end for

end a;
```

L'ensemble de description complet est simulable. On peut faire une remarque sur la syntaxe : l'affectation des constantes génériques dimensionnées doit pouvoir se faire sans préciser l'unité. En effet celle-ci est donnée dans la définition, et l'écriture devient vite lourde.

V. Conclusion

Le langage présenté dans ce chapitre est une réunion des différentes requêtes de la normalisation, des versions courantes des LES, de propositions personnelles, et d'autres qui sont issues d'un travail conjoint entre le CNET, Anacad et le laboratoire ARTEMIS. Tous trois participent activement à la normalisation du langage VHDL-analogique. Les exemples présentés ont permis de s'assurer que les primitives proposées permettent de décrire un bon nombre de circuits.

De plus, on a pu mettre en valeur des points qui restent à définir. En premier lieu, l'affectation de forme d'onde continue pour un signal, un courant ou une différence de potentiel; on peut par exemple créer un ensemble de procédures de base : PWL, sinusoïde (amortie ou non), exponentielle, pulsation périodique, procédures qui permettent de donner un futur à un état, ou aux champs d'une pin. D'autre part, l'utilisation de la vérification des dimensions des expressions physiques n'est pas aussi lourde à écrire qu'il y parait, et cela donne même des descriptions plus cohérentes. Ce mécanisme mérite donc d'être formalisé en le complétant par la définition de la dimension et non plus uniquement du type.

Les deux sortes de blocs de description analogique : procédural et par relation, semblent nécessaires, bien que la description procédurale se destine plus à des simulateurs utilisant la méthode nodale modifiée. D'autre part, s'il est possible d'écrire des relations dans la partie procédurale, le bloc relation serait peut être inutile. De plus, il ne faut pas oublier que la majorité des simulateurs du marché utilise la méthode nodale modifiée et l'utilisation exclusive des relations

limiterait leurs performances. Enfin, décrire un composant en calculant explicitement un courant ou une différence de potentiel parait très naturel, et tend à se rapprocher de la description fonctionnelle numérique.

Pour finir, le point le plus important est l'interface mixte. Comme on a pu le voir, les interactions mixtes ne sont pas clairement définies et décrire les composants d'interface n'est pas une chose facile. Cela est en train de se faire, et des cycles de simulation et d'initialisation vont être proposés au groupe de standardisation. Actuellement, on possède quatre moyens d'échange entre les deux mondes :

- l'attente sur un seuil, côté numérique,
- la lecture d'une tension ou d'un courant, côté numérique,
- la lecture d'un signal, côté analogique.

Le langage n'est pas encore défini, un important travail reste à faire. Les propositions contenues dans ce chapitre ont et vont servir à alimenter les réflexions du groupe de normalisation.

Chapitre III.
Simulation mixte analogique-numérique

I. Introduction

La vérification de la conception d'un circuit VLSI est très importante face aux coûts de plus en plus élevés du processus de fabrication. D'autre part, comme les outils de preuves formelles ne sont actuellement que des prototypes développés par des universités ou des centres de recherche, la simulation reste l'outil le plus utilisé pour valider cette réalisation.

De plus, le nombre de transistors par circuit a subi lui aussi une importante progression : ce nombre est passé de quelques milliers au million en moins d'une décennie. Cette augmentation rend la tâche des simulateurs de plus en plus ardue. En effet, un simulateur qui est capable de simuler dix mille transistors (ou équivalent-porte) en un temps correct, c'est à dire en moins d'un jour, ne pourra pas en simuler dix fois plus. Car, dans ce cas le temps de simulation prendrait plusieurs jours si la complexité de l'algorithme est linéaire et très longtemps sinon. Face à ce problème, l'augmentation de la vitesse de calcul des machines n'est pas suffisante, et c'est pour cela que les algorithmes des simulateurs doivent être sans cesse améliorés. On peut aussi noter l'apparition de périphériques dédiés à la simulation qui aident les stations de travail : ce sont les accélérateurs matériels. Enfin, pour donner un ordre de grandeur, les meilleurs simulateurs électriques actuels peuvent simuler plusieurs milliers de transistors, ce qui n'est pas suffisant pour simuler un circuit VLSI dans son ensemble.

Face à ces problèmes, la simulation en mode mixte analogique numérique est une bonne solution. En effet, cette solution permet de conjuguer la précision de la simulation analogique et la rapidité de la simulation numérique. Bien sûr, ceci est un compromis, car le concepteur n'a pas le même degré d'information sur tout le circuit, c'est pour cette raison qu'il faut décomposer correctement le circuit et isoler les parties qui nécessitent une simulation fine.

Donc la première utilité de la simulation en mode mixte est qu'elle permet de simuler un circuit VLSI dans son intégralité (ou presque), et ainsi d'en valider le comportement. Mais, ce n'est pas la seule.

Ainsi, les applications modernes de l'électronique telles que les télécommunications, la télévision haute définition ou encore le traitement du signal, sont de plus en plus tournées vers les circuits dont une partie est composée de fonctions analogiques et l'autre de fonctions numériques. Les raisons principales de cette orientation sont d'une part qu'à fonction égale la réalisation analogique nécessite beaucoup moins de transistors et donc moins de place sur le silicium que son équivalent numérique, et d'autre part que l'électronique analogique a fait de gros progrès ces dernières années, et ainsi la différence de qualité entre certaines fonctions analogiques et leur équivalent numérique est devenue négligeable pour bien des applications. Donc, pour ces circuits la simulation mixte est nécessaire pour valider le bon fonctionnement de l'ensemble du circuit surtout au niveau des échanges entre les deux niveaux.

C'est pour ces deux principales raisons que la simulation en mode mixte connaît un tel engouement actuellement.

Après cette rapide introduction, voici le plan de ce chapitre. Dans un premier temps nous aborderons des généralités afin de présenter les notions suffisantes pour cerner les différents problèmes : ainsi il sera question de simulation analogique et numérique. Ensuite, suivra une courte partie sur la décomposition des circuits, c'est à dire comment isoler la ou les parties qui doivent être simulées finement. Les trois parties suivantes concerneront les problèmes de réalisation concrète d'un simulateur mixte. Seront abordés, dans l'ordre : la synchronisation des deux algorithmes de simulation, ensuite les échanges du point de vue de la description et enfin tout le problème de l'initialisation. La dernière partie de ce chapitre sera une présentation rapide de différents simulateurs mixtes existants et de leurs principales différences.

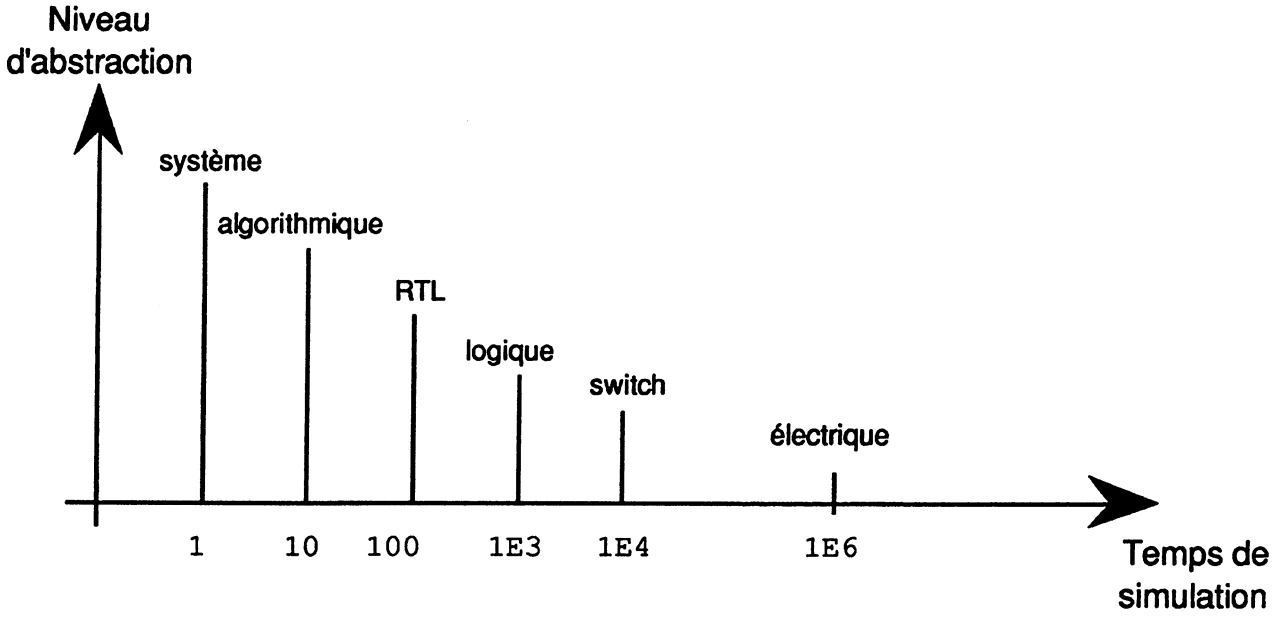
II. Généralités

1. Un mot à propos de la simulation et des HDLs

Il faut admettre que ce sont les outils de simulation qui sont les plus utilisés dans la chaîne de CAO. C'est donc pour eux qu'a été développée la majorité des HDLs. D'ailleurs, on mélange souvent le langage et l'outil de simulation qui se trouve derrière.

Pour ce qui est de la simulation, il existe deux **modes** : la simulation discrète et la simulation continue, ou bien encore numérique et analogique. Une de leurs principales différences est le rapport de la précision des résultats sur le temps de simulation.

Le graphique suivant positionne le niveau d'abstraction en fonction du temps de simulation, donné ici en coût relatif, pour un circuit donné :



1. Les modes de simulation

a. La simulation discrète

Ce mode de simulation est utilisé pour les systèmes dont les variables d'état internes peuvent être exprimées avec un nombre fini de valeurs, et dont le nombre d'**événements** durant la simulation est aussi en nombre fini. Un événement est un changement d'état d'un système.

Il existe deux méthodes de gestion de ce type de simulation. La première évalue les sorties de tous les systèmes à chaque pas de calcul, on parle de séquencement statique. Cette méthode est particulièrement bien adaptée aux circuits synchrones. La seconde n'évalue que les sorties des composants dont une entrée subit un événement. Ce type de simulation est dit piloté par événement (event driven).

Quant au temps de la simulation, il est en théorie continu. Cependant, l'implémentation informatique échantillonne le temps, et dans ce cas, il est stocké comme un nombre entier d'une fraction de la seconde (nano, pico ou femto), ceci venant du fait qu'il est plus rapide de manipuler un entier qu'un nombre en virgule flottante.

La réalisation d'un simulateur numérique ne présente pas de réelles difficultés. Par contre, les mécanismes de gestion des événements doivent être optimisés sinon pour de grosses réalisations, les temps de simulation peuvent devenir importants.

b. La simulation continue

Cette simulation est très proche de la réalité, elle utilise des lois physiques (la loi d'Ohm, les lois de Kirchoff, ...), elle donne des résultats très précis, mais les temps de simulation sont très élevés. De plus, les différentes valeurs, les états des systèmes et le temps, sont codés sous forme de nombres en virgule flottante.

La simulation continue possède plusieurs modes d'analyse :

- le mode transitoire
- le mode continu
- le mode fréquentiel
- le mode mixte fréquentiel-transitoire
- le mode de balance harmonique.

Pour le mode transitoire, comme nous l'avons déjà présenté dans le premier chapitre, le simulateur doit résoudre un ensemble d'équations algébriques différentielles non linéaires. Grâce à un schéma d'intégration (Euler, trapèze ou Gear) les dérivées sont estimées, et il reste alors un

système d'équations algébriques non linéaires qui se résout par exemple avec l'algorithme de Newton-Raphson. L'avancement dans le temps est en général géré par le simulateur. Dans ce cas il n'est pas fixe et dépend de l'activité du système, c'est à dire plus précisément des valeurs des dérivées des variables d'état. Cependant, pour certains cas particuliers, l'utilisateur peut décider de fixer ou borner ce pas temporel.

Ces algorithmes permettent aussi de traiter de nombreux problèmes physiques (mécanique des fluides, optique, thermodynamique, ...) par analogie avec le monde électrique.

L'algorithme de base (Newton-Raphson) résout à chaque pas de calcul le circuit en entier. Des améliorations ont été introduites pour tenir compte de la latence du circuit qui est importante pour la simulation de circuits numériques MOS, ou encore le découplage entre la grille et le drain, et entre la grille et la source des transistors MOS. Il s'agit des méthodes de relaxation [Whi87].

Cependant, il existe des simulateurs électriques conduits par événements (event driven) [Che84, Sak85, Cot90, Pat90] qui utilisent la latence du circuit, le faible couplage et la directionalité de certains composants électriques (MOS); mais en général ils ne permettent pas la simulation de circuits purement analogiques fortement rebouclés (filtres, amplificateurs opérationnels, transistors bipolaires). On parle alors de simulation *timing*. Les algorithmes utilisés sont ceux de relaxation.

Dans d'autres cas [Cot90], il s'agit de la simulation de fonctions de transfert en S : plusieurs algorithmes ont été donnés pour permettre leur intégration dans de tels simulateurs [Cha92]. Les fonctions de transfert en S décrivent le comportement d'un système analogique en fonction de la fréquence. Cela permet de faire des descriptions aisées des filtres. Ceci est en fait un cas particulier de la simulation analogique dans laquelle des influences directionnelles peuvent être définies.

3. La simulation en mode mixte et la description multi-niveaux

Une description multi-niveaux est une description d'un système sur plusieurs niveaux de description, alors qu'une simulation en mode mixte [Sak81, Sal90] est une simulation d'un système en utilisant les deux modes de simulation : continu et discret.

La simulation d'une description multi-niveaux en utilisant le même mode de simulation n'est pas très difficile. Ce n'est pas le cas de la simulation en mode mixte, parce que les coeurs de simulateurs sont différents et il faut donc gérer leur synchronisation et traduire les différentes valeurs d'un niveau discret à un niveau continu.

Par exemple, il est possible de faire de la description multi-niveaux avec VHDL. On peut faire avec ce dernier de la description au niveau logique, registre, algorithmique ou encore système. Pour une telle simulation, il y a juste à gérer les échanges de données et les délais de propagation entre les différents niveaux. Par contre, VHDL ne possède pas la sémantique suffisante pour faire de la simulation en mode mixte. Cependant il est possible de coupler un simulateur VHDL avec un simulateur électrique [Alt91].

Comme nous l'avons vu, un exemple d'approche générale fut celle du projet Cascade [Hum84]. Le but de ce projet était de développer un langage multi-niveaux associé à un simulateur multi-modes. Chaque niveau de description possède son propre HDL, mais tous les HDLs composant Cascade ont été définis sur une base syntaxique commune. Entre autres, le niveau électrique est décrit avec les langages Imag (D et F) qui permettent de faire de la description structurelle et comportementale.

III. La décomposition

Le but d'une simulation en mode mixte est d'accroître la vitesse par rapport à une simulation continue et obtenir des résultats plus précis par rapport à une simulation discrète. Pour obtenir le meilleur d'une simulation mixte il faut isoler la partie qui nécessite une bonne précision et qui sera simulée par le simulateur électrique.

Bien entendu, on parle de partie critique uniquement pour les circuits numériques ou pour les parties numériques de vrais circuits mixtes (convertisseurs).

Cette partie est souvent la partie en cours de mise au point, ou bien celle qui utilise les performances maximales de la technologie. Pour le second cas, la tâche d'isolement de cette partie est faite à la main par l'utilisateur et elle nécessite beaucoup d'expérience. Quant à son automatisation, il s'agit d'un problème ardu qui est encore un sujet ouvert de recherche.

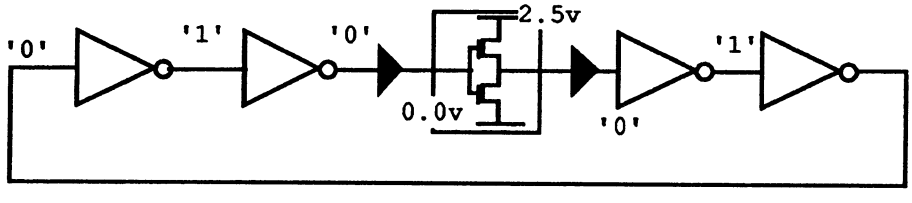
Un autre sujet intéressant est le changement de niveau de description voire même de mode de simulation, en cours de simulation selon que telle ou telle partie du système simulé entre dans un état critique. Le simulateur Lsim/Adept [Odr86] permet de changer de niveau de description : du niveau interrupteur au niveau logique. Cela n'est pas possible dans l'autre sens. On peut parler de changement de mode, mais Adept est en fait un simulateur timing. Toutefois, pour changer de mode ou de niveau, il faudrait que l'utilisateur décrive chaque partie de son système en différents niveaux.

1. Changement de mode de simulation en cours de simulation

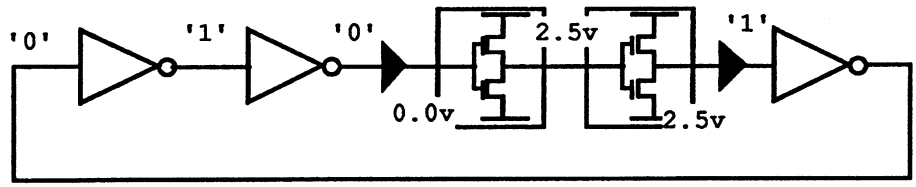
Le changement de mode en cours de simulation de tout ou partie d'une description est un problème assez complexe. En effet, cela implique une modification dynamique de l'ensemble d'équations à résoudre du simulateur continu. Actuellement cela n'est pas possible dans les simulateurs actuels, car tous travaillent avec un nombre d'inconnues fixe.

Essayons d'isoler l'ensemble des modifications à apporter pour permettre une telle chose. Au moment où le mode de simulation va changer sur une partie du système en cours de simulation, il faut que le simulateur continu calcule l'état de cette partie dans un niveau continu. Ce calcul peut être aidé grâce à une traduction des valeurs des états discrets de la partie, mais ce n'est pas suffisant dans la plupart des cas. En effet, le niveau en mode continu va être plus détaillé et ainsi contenir plus d'inconnues que d'états discrets. Donc il faudra que le simulateur continu fasse une résolution de tout le système à cette date pour trouver un état courant cohérent. Bien entendu, en aucun cas les valeurs des inconnues des autres parties de la simulation continue ne doivent changer, par contre il pourra arriver que les valeurs traduites soient modifiées. Ceci résulte du fait que la traduction est assez grossière dans ce sens.

Voyons maintenant un petit exemple simple, un anneau de cinq inverseurs dont un est analogique et son successeur change de description selon son état :



La sortie de l'inverseur simulé au niveau électrique est en train de passer de 0 volt à 5 volt. A 2.5 volt, l'interface mixte traduit le signal numérique équivalent en '1', à ce moment là, l'inverseur suivant juge qu'il entre en phase critique et donc change de niveau de description, on obtient alors le schéma suivant :



Donc, dans un premier temps on peut noter qu'il faut déplacer l'interface mixte. Ensuite, il faut calculer la valeur des tensions du *nouvel* inverseur. En traduisant les valeurs numériques, on obtient 5 volt à l'entrée et à la sortie. Ceci est faux car la valeur de l'entrée est 2.5 volt, car c'est la sortie de l'inverseur précédent. Donc, il faut modifier la tension d'entrée. Après cela le simulateur peut vérifier si la solution trouvée est satisfaisante en résolvant le nouvel ensemble d'équations. Dans ce cas précis, il va modifier la sortie de l'inverseur en 2.5 volt. La simulation peut alors reprendre. Le gros problème est donc de trouver des valeurs cohérentes pour le simulateur continu.

On devine parfaitement l'intérêt d'une telle solution. En effet, si une partie du circuit entre en phase critique uniquement sous certaines conditions, que ce passage en zone critique ne survient que dix pour cent du temps, il est dommage pour une simulation de simuler cette partie au niveau électrique alors que le modèle discret suffirait. Si une telle solution voyait le jour, les concepteurs auraient un outil équivalent au débuggeur des informaticiens, le pas-à-pas après un point d'arrêt deviendrait une simulation fine après une entrée en zone critique.

Cependant, il faut voir si ce changement de mode de simulation ne nuit pas à la simulation dans son ensemble et si les résultats ne sont pas perturbés. Dans ce cas, cette solution, certes attrayante du point de vue réalisation, n'aurait que peu d'utilité.

Une solution plus intéressante et facilement réalisable avec un langage comme VHDL, serait de changer de niveau de description en cours de simulation. Cela est très facile grâce à une instruction de choix au sein de la description comportementale. Dans ce cas, pour la partie continue, il faut maintenir un nombre d'inconnues constant entre les niveaux et surtout assurer la continuité entre

les différents modèles.

2. Reconnaissance de modèle numérique dans une description analogique

Ici, il s'agit de faire passer une description analogique dans un pré-processeur pour reconnaître des fonctions numériques. Celles-ci seront alors remplacées par des descriptions au niveau numérique.

Cette solution présente l'avantage de reprendre les descriptions existantes et ainsi de faire bénéficier les concepteurs de la simulation mixte.

3. Isolement de la partie critique à simuler

Voici le problème le plus ardu. En effet, il est très difficile dans un circuit d'isoler la partie dont le fonctionnement est le plus important dans le circuit. En effet, chaque partie est importante et est nécessaire au bon fonctionnement de l'ensemble.

Pour isoler une telle partie, il faut définir quelles sont les contraintes que chaque partie du circuit doit respecter, ces contraintes peuvent être des contraintes de temps de propagation ou encore de puissance de sortie pour supporter la charge de l'étage suivant. L'étape suivante serait de classer quelles sont les contraintes les plus importantes. Enfin, un outil pourrait, à l'aide d'analyses de Monte-Carlo de chacune des parties, classer toutes les parties du circuit en fonction de leur sensibilité aux variations technologiques : taille des transistors ou encore valeurs des capacités et des résistances. On obtiendrait alors un classement des parties et on pourrait trouver la partie dite critique en fonction des critères choisis.

4. Conclusion

Cette notion de partie critique était importante il y a une dizaine d'années. A cette époque les simulateurs mixtes servaient à simuler des circuits numériques entiers, seule une cellule ou une partie était simulée au niveau électrique. Maintenant, les simulateurs mixtes servent à simuler de vrais circuits mixtes, des convertisseurs NA ou AN, circuits qui sont et seront de plus en plus nombreux. Pour de tels circuits, la décomposition n'est pas difficile à faire.

D'autre part, la maîtrise sur l'électronique numérique est telle que de plus en plus les circuits numériques sont synthétisés automatiquement. De plus, les vérifications ne se font plus au niveau du circuit, mais au niveau de la cellule. Chacune de ces cellules est finement analysée au moyen de simulation au niveau électrique. Le résultat de cette analyse permet de rétro-annoter la description numérique.

IV. La synchronisation

Un simulateur mixte est composé de deux noyaux de simulation dont les fonctionnements sont fondamentalement différents. Chacun de ces coeurs de simulation a besoin de connaître la valeur d'objets de l'autre : valeur d'un signal numérique d'un coté, tension d'un noeud de l'autre.

On définit l'événement mixte comme étant la modification d'une de ces valeurs communes aux deux coeurs de simulation. Quand un tel événement est généré dans un des deux coeurs, celui-ci doit le signaler à l'autre, et ce dernier doit en tenir compte à la bonne date. Ils doivent également se synchroniser sur cette date, et cette synchronisation peut ralentir énormément la simulation selon les cas et selon le choix de l'algorithme de synchronisation.

Dans un premier temps, nous reviendrons sur les principes généraux de ces deux types de simulation et surtout sur les aspects qui permettent la synchronisation. Ensuite, nous verrons différentes méthodes de synchronisation en essayant de voir leurs avantages et leurs inconvénients.

1. La simulation discrète et continue

Comme nous l'avons vu précédemment, le mécanisme général de la simulation discrète est la gestion et la propagation des événements. On parle souvent de simulation conduite par événements (event driven). Chaque système peut calculer ses sorties en fonction de ses entrées, sans itérations, mais il ne peut pas modifier ses sorties dans un délai nul. Il est vrai qu'en VHDL il est possible de faire cela, mais en interne, le simulateur introduit la notion de pas infinitésimaux appelés délais delta. Cette particularité permet de mener à bien toutes les simulations. Enfin, lorsque le simulateur finit un pas de calcul, il connaît la date de son prochain événement.

Ci-après, se trouve le cycle simplifié de simulation VHDL.

1. Si aucun **pilote de signal** n'est **actif** à la date courante alors le temps avance jusqu'à la date à laquelle un pilote devient actif ou jusqu'à ce qu'un **processus** reprenne. La simulation s'arrête lorsque le temps de simulation devient égal à TMAX.

2. Chaque signal actif dans le modèle est mis à jour (Cela peut générer des **événements** sur d'autres signaux)

3. Pour chaque processus P, si P est **sensible** sur un signal S et qu'un événement a eu lieu sur S pendant ce cycle de simulation ou si P était en attente pour une durée donnée et que celle-ci est finie alors P reprend.

4. Tout les processus qui ont repris s'exécutent jusqu'à leur point d'arrêt.

De son coté la simulation continue doit résoudre un ensemble d'équations algébro-différentielles non linéaires qui doivent être vérifiées à chaque instant. Le simulateur avance pas à pas, mais à priori il ne connaît pas la date de son prochain pas de calcul, il n'existe pas de simulateur efficace qui fonctionne en pas constant. Il peut seulement connaître un majorant de celle-ci. Cependant, il possède une particularité intéressante : après avoir convergé à une date donnée, c'est à dire qu'il a trouvé des valeurs pour les inconnues qui vérifient le système d'équations à une précision suffisante, il possède en mémoire au moins deux états complets, ce qui lui permet de pouvoir revenir en arrière d'un pas de calcul. Ce retour se fait simplement pas le rejet du nouveau pas de calcul, de la même manière que lors d'une non convergence. Cette particularité peut servir pour la synchronisation avec un simulateur discret.

1. Le simulateur estime le prochain pas de calcul h.

2. Il essaie de résoudre le système au temps t+h.

3. S'il ne converge pas pour ce point, alors il faut revoir le pas à la baisse et on repart en 2, sinon il accepte le pas de calcul.

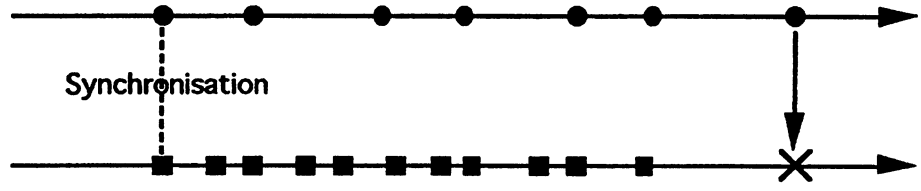
4. Enfin il avance dans le temps. (Fin du cycle)

2. La synchronisation avec retour-arrière (back-track)

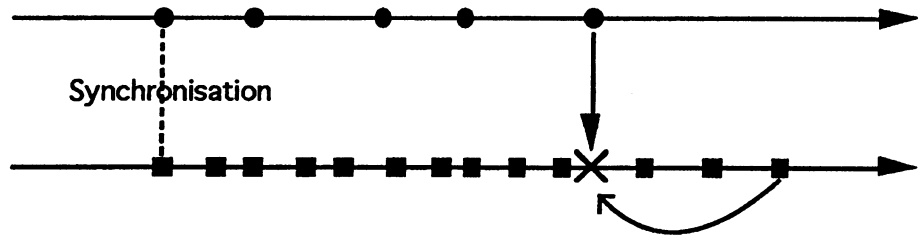
Pour commencer, voici la solution la plus simple du point de vue de la réalisation d'un simulateur mixte à partir de deux simulateurs différents. Elle est cependant quelquefois très coûteuse.

Chaque simulateur avance de son coté sans se soucier de l'autre jusqu'à ce qu'un événement

mixte se produise ou qu'une requête de l'autre arrive. Evidemment, celui qui a détecté l'événement génère une requête pour l'autre. Le schéma suivant illustre bien les deux différents cas possibles.



Premier cas : le simulateur qui reçoit le message doit atteindre la bonne date pour le prendre en considération.



Second cas : le simulateur qui reçoit le message est au-delà de la requête, il doit revenir en arrière pour se synchroniser à la bonne date.

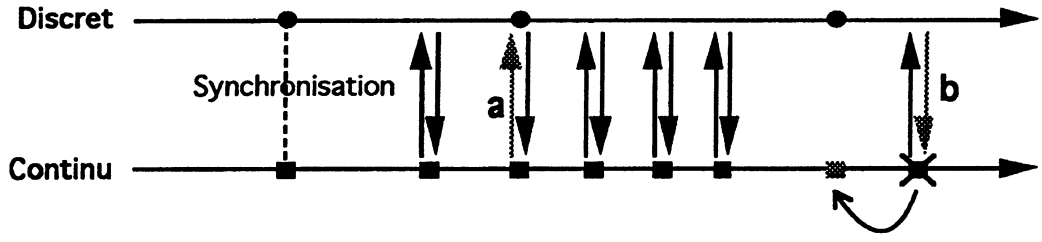
Sur ces schémas, il n'y a pas de distinction entre les simulateurs. Ce n'est pas important, car leur rôle est symétrique. Une légère différence existe cependant pour le second cas : en effet, pour revenir en arrière, le simulateur continu doit calculer l'état du système exactement à la bonne date alors que le simulateur discret doit prendre l'état au pas qui précède cette date.

Cette solution donne de bons résultats pour des systèmes faiblement couplés et qui demandent peu de synchronisation. De plus, elle peut facilement tourner sur des machines ayant deux processeurs ou encore sur des réseaux. Par contre, chaque simulateur doit conserver en mémoire l'historique de la simulation, au moins jusqu'à la date de la dernière synchronisation, et ceci peut poser des problèmes de taille mémoire et engendrer de nombreux calculs inutiles. Il donne cependant de bon résultats lorsque les deux simulations avancent à la même vitesse, ce qui arrive rarement en pratique.

3. L'approche du saut de grenouille (leap frog)

Dans cette solution, le simulateur continu avance d'un pas et avant de l'accepter il demande une

confirmation du simulateur discret. Ce dernier regarde alors son échéancier, calcule ou non quelques pas mais en aucun cas ne dépasse la date atteinte par le simulateur continu. Enfin, si aucun événement mixte n'est généré alors il rend le contrôle en donnant une réponse positive. Au contraire, si un événement mixte est généré alors il renvoie la date de cet événement mixte avec une réponse négative. Alors le simulateur continu rejette le pas calculé et se recale sur la date qui lui est fournie. En général, ce retour en arrière est effectué grâce à une interpolation polynomiale.



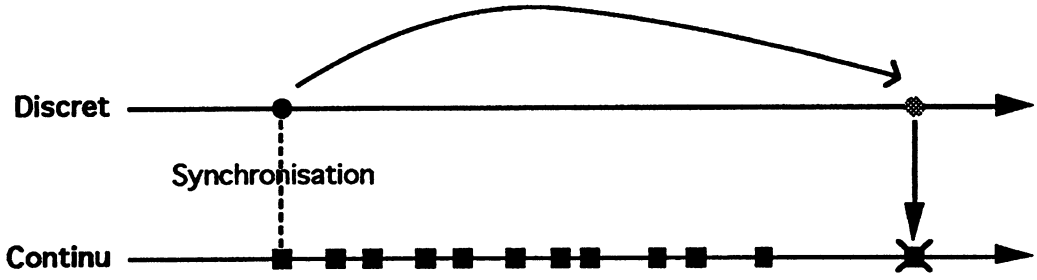
En a : le simulateur continu génère un événement mixte, le simulateur discret le prend en considération.

en b : le simulateur discret rejette le pas du simulateur continu car il y a eu un événement discret avant, et donc le simulateur continu doit revenir en arrière d'un pas.

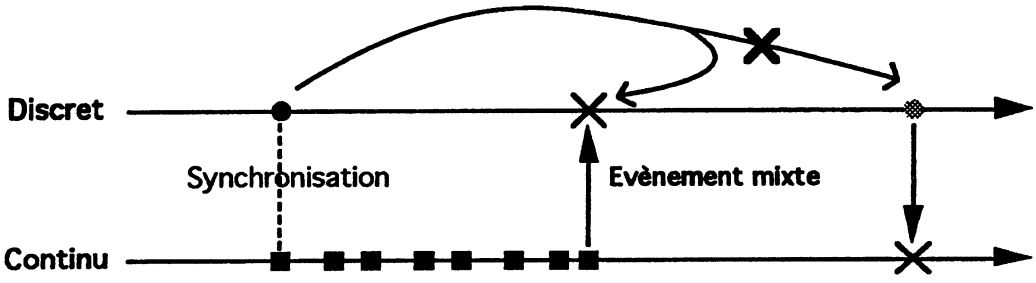
Cette approche donne de bons résultats si la réponse du simulateur discret est rapide, surtout dans le cas où il n'a aucun événement à traiter.

4. L'approche du pas bloqué (lock step)

Le simulateur discret donne au simulateur continu la date du prochain événement qu'il va évaluer. Le simulateur continu va avancer jusqu'à cette date, sans la dépasser, ou jusqu'à ce qu'il génère un événement mixte. Dans le premier cas, il rend la main, le simulateur discret calcule son pas et le cycle est fini. Dans le second, cas le simulateur discret doit tenir compte de l'événement mixte et effectuer un pas supplémentaire, et le cycle est fini.

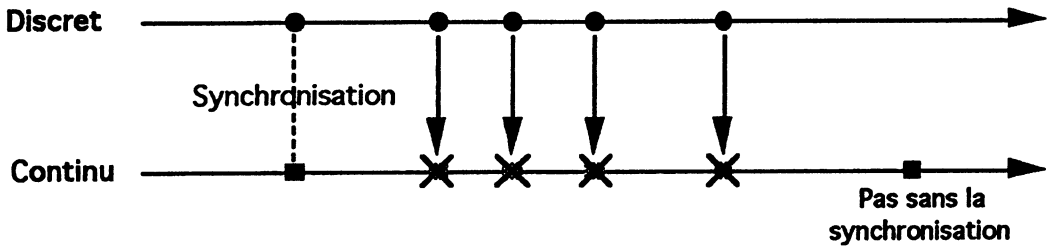


premier cas : le simulateur discret donne la date de son prochain événement et le simulateur continu l'atteint sans problème.



second cas : ici, un événement mixte est généré avant la date du prochain événement du simulateur discret et ce dernier doit en tenir compte.

Cependant, il y a un cas où cette solution donne de mauvais résultats : c'est quand le pas de calcul du simulateur continu est supérieur à celui du simulateur discret, car on oblige le premier à faire des pas supplémentaires et ceux-ci sont coûteux. Heureusement, ce cas est rare. La figure suivante illustre ce cas.



Cas inefficace : Le simulateur continu a des pas de calcul plus grands que ceux du simulateur discret, mais il doit les réduire et faire des calculs supplémentaires à cause de la synchronisation.

On peut améliorer l'algorithme, si on accepte que le simulateur continu dépasse la date du prochain événement mais sans accepter le dernier pas. Alors, on se trouve dans le même cas que pour le saut de grenouille, c'est à dire que le simulateur discret effectue ses calculs au maximum jusqu'à la date d'arrêt du simulateur continu, et en cas d'événement mixte, le simulateur continu doit revenir en arrière.

5. Remarque importante

Le simulateur analogique ne peut pas pour des raisons de continuité prendre immédiatement en compte les modifications du simulateur numérique. Ainsi à un temps donné, les valeurs des interfaces lues par le simulateur analogique sont celles du temps immédiatement précédent.

Par contre, le simulateur analogique a besoin de savoir si il y a eu une modification (événements) sur une interface afin de connaître le début de la montée d'une tension. En effet, une valeur analogique, qui est une inconnue, commandée par une valeur discrète doit changer d'état de manière continue. Mais sa dérivée par rapport au temps peut être discontinue à condition qu'elle ne soit pas elle aussi une inconnue.

V. L'interface mixte

Ici nous avons à distinguer deux problèmes bien distincts. Le premier est celui de la description : comment faire une description cohérente avec deux niveaux d'abstraction et souvent deux langages. Le second est évidemment le problème de la simulation : les simulateurs doivent échanger des données de types différents, et chacun doit émuler les interactions engendrées par l'autre partie.

Comme on l'a vu dans le chapitre précédent, l'approche choisie par VHDL analogique se rapproche de celle du projet Cascade. En effet, un langage unique sera développé pour la simulation en mode mixte. Dans ce langage aucune interaction ne sera pré-définie, l'utilisateur devra tout gérer par lui même. Par contre, la synchronisation reste du domaine de celui qui réalisera un simulateur et pourra dépendre de l'implémentation. Toutefois, un schéma standard sera proposé pour la portabilité.

1. Au niveau de la description

En général, la description se fait à l'aide de deux langages différents dont la sémantique a été étendue. Une pseudo-hiérarchie est créée, que ce soit la description numérique ou la description analogique qui voit l'autre comme une ou plusieurs boîtes noires. On pourra prendre en exemple la description pour VHD$_e$LDO, on se reportera alors au chapitre quatre où plusieurs exemples sont

donnés.

Pour l'échange des données, il y a entre autre deux possibilités. La première est de cacher toutes les valeurs internes à chaque partie et de n'accepter que la communication au travers de l'interface. Dans ce cas, on crée des noeuds de connexion mixtes auxquels sont attachées des fonctions de conversion plus ou moins ouvertes à l'utilisateur. D'un autre coté, la seconde solution consiste à créer des règles de visibilité dans la hiérarchie et autoriser uniquement la lecture des valeurs des états de l'autre niveau. Quant à la conversion, elle est faite dans des composants explicites que l'utilisateur peut réécrire à son gré.

La première solution est la solution adoptée lorsqu'on utilise deux langages différents : par exemple pour VHD$_e$LDO. Quant à la seconde, plus générale, elle est utilisée par Cascade et le sera par VHDL-analogique.

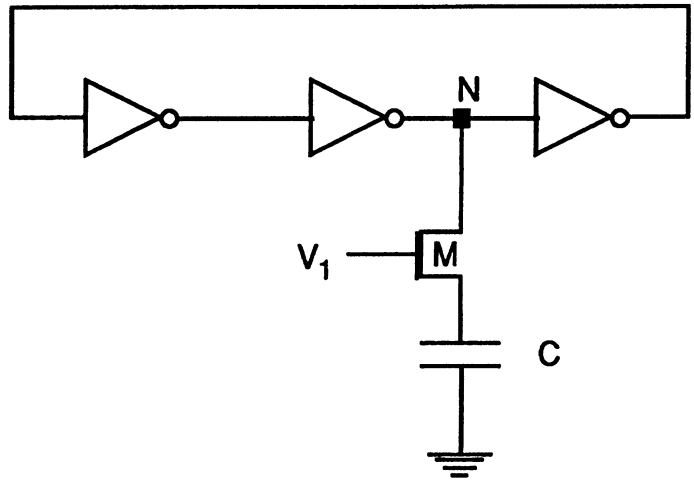
2. Au niveau de la simulation

La simulation doit donner des résultats proches de la réalité. La traduction doit donc être faite le mieux possible autant pour les valeurs que pour les délais.

La traduction de l'analogique vers le numérique n'est pas trop compliquée. Seul l'étage d'entrée du numérique doit être modélisé pour introduire un retard du côté analogique, car son impédance n'est pas infinie.

Par contre dans l'autre sens c'est moins simple, car il faut traduire des valeurs discrètes en un couple de valeurs continues courant-tension. De plus, il faut modéliser l'étage de sortie du numérique.

Un problème encore plus compliqué est celui de l'interface bi-directionelle. Encore faut-il savoir si elle est utile ! On peut quand même trouver un exemple [Kle91] dans lequel elle est mise en valeur (schéma suivant).

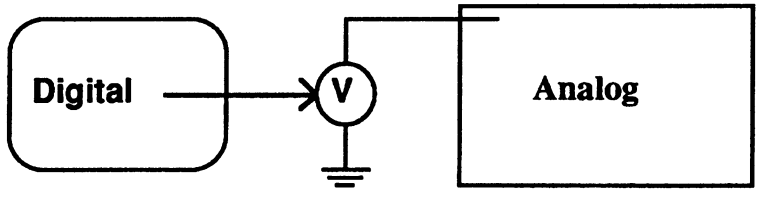


Ce circuit est décrit sur deux niveaux de description : les inverseurs au niveau logique et le transistor MOS et la capacité au niveau électrique. Maintenant il faut choisir où mettre l'interface mixte. La fonctionnalité du circuit est la suivante : il s'agit d'un oscillateur dont la fréquence est contrôlée par une tension. Le transistor est utilisé en temps qu'interrupteur, sa fermeture permet d'augmenter la charge du noeud N. Il y a bien bi-directionalité au noeud N, autant la partie électrique que la partie numérique le contrôle.

La plupart des simulateurs mixtes proposent un ensemble pré-défini d'interfaces analogique-numérique ou numérique-analogique, et aucun ne permet à l'utilisateur de définir ses propres modèles. De plus, la place de ces interfaces est soit laissée libre à l'utilisateur soit choisie automatiquement par programme. Ce dernier cas peut être une cause d'inefficacité, surtout pour des cas un peu complexes (cf. exemple plus haut).

a. L'interface numérique-analogique

i. La source contrôlée

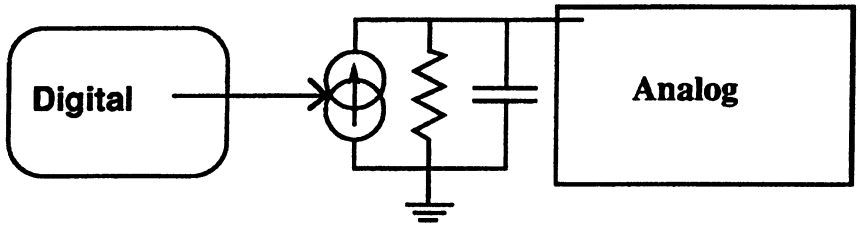


Il s'agit de l'interface la plus simple. Prenons le cas de la source de tension, la valeur numérique contrôle une source de tension dont la valeur varie d'un niveau bas à un niveau haut. Evidemment

la valeur de la source ne peut pas changer instantanément, et une pente est introduite pour simuler le temps de montée. La durée de cette pente est contrôlée par l'utilisateur. Mais cette méthode ne permet pas de modéliser les conflits, c'est à dire un noeud analogique piloté par plusieurs sources numériques. De plus, la charge analogique n'est pas prise en compte, en effet si celle ci est trop petite, alors la valeur imposée par la partie numérique ne devrait pas être atteinte, alors qu'elle l'est avec cette interface. Cette dernière a une impédance nulle, et elle est satisfaisante pour conduire des parties analogiques ayant une forte impédance : par exemple une grille de transistor MOS.

Le cas de la source de courant est symétrique, dans ce cas l'impédance de l'interface est infinie et elle ne peut conduire que des faibles impédances : par exemple des bases de transistors bipolaires.

ii. le circuit IRC



Une méthode plus précise est de modéliser la sortie numérique par un circuit IRC (cf. figure) qui est rajouté dans la description analogique. Il existe son équivalent Norton VRC avec la source de tension en série avec la résistance. Mais, l'avantage du circuit IRC sur le VRC est qu'il n'ajoute pas de noeud à la description.

Les paramètres des trois composants dépendent non seulement de l'état de la partie numérique, mais aussi de la technologie utilisée par le circuit : TTL, ECL ou encore Bi-CMOS. Cette solution permet de modéliser les logiques multi-valuées, c'est à dire les notions de force ou encore de haute impédance (cf. sous-paragraphe suivant). De plus, les conflits peuvent être résolus dans la partie analogique et donc de manière plus fine. Dans tous les cas, il faut que les paramètres de cette interface soient donnés par l'utilisateur.

iii. Les logiques multi-valuées

Pour faire des description plus précise de systèmes logiques, on a introduit les types logiques multi-valuées. Ces types permettent de tenir compte de l'impédance de sortie d'un système et ainsi il

est possible de modéliser les conflits pour les bus multi-sources.

Ces types vont du plus simple quatre états : 'X', '0', '1' et 'Z', au plus complet à 46 états présenté dans [Coel84]. Il est a noter que ce dernier permet de faire de la description multi-technologies. Ainsi, chaque fonction logique ne produira pas les mêmes résultats si elle est réalisée en logique TTL ou bien en CMOS. Chaque état représente en fait la sortie spécifique, valeur et impédance, d'une technologie donnée.

Dans tous les cas, pour convertir ces types et modéliser les conflits du coté analogique, il faut obligatoirement passer par une interface du type IRC. Pour plus de précision, on peut envisager de rajouter à la description analogique l'étage de sortie du système numérique qui pilote l'interface. Dans ce cas, on émule de façon très précise les différentes technologies. Cette possibilité est offerte par le simulateur mixte de Dazix.

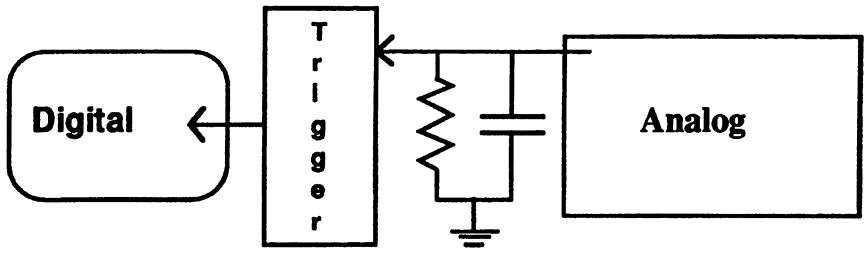
iv. La traduction du X

Dans les logiques multi-valuées, le X ou équivalent représente l'état indéterminé ou encore inconnu. Pour le traduire, on peut dire qu'il s'agit de la valeur moyenne entre la tension haute et basse. Une autre solution consiste à conserver la valeur précédente. Dans tout les cas, aucune des deux solutions n'est satisfaisante, car le X peut aussi bien représenter une altération du signal qu'une résultante d'un conflit, une transition ou bien encore une valeur non encore calculée. Le choix pourra donc être dans certains cas correct et dans d'autres faux.

D'ailleurs, on retrouve ces problèmes au niveau de la simulation numérique, et c'est pour cela que des logiques possèdent les états *non-initialisé* et encore *don't care*.

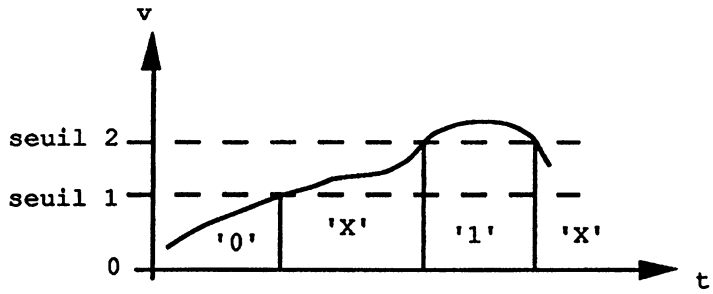
b. L'interface analogique-numérique

Comme cela a été dit auparavant, cette interface n'est pas très compliquée. Il s'agit de modéliser l'étage d'entrée du numérique. Une première solution simple consiste à prendre l'impédance d'entrée du numérique comme infinie et donc de considérer la puissance consommée comme nulle. Pour plus de précision, on peut rajouter un circuit RC à la partie analogique dont les paramètres sont contrôlés par l'état du système numérique.



Ensuite, il faut sélectionner quelle valeur du signal représente le mieux la valeur de la tension du noeud mixte ou du courant de la branche mixte. Pour cela, on donne des seuils, avec hystérésis ou non. D'autre part, on peut envisager de calculer l'impédance du noeud dans le cas où le signal est d'un type multi-valué qui représente la force : par exemple le type VHDL standard std_logic_1164.

D'autre part, on peut aussi propager un état X si la tension se trouve entre deux seuils de décision comme on le voit sur le schéma suivant :



VI. L'initialisation

L'initialisation, aussi bien en simulation numérique qu'en simulation analogique est là pour trouver un point de départ à la simulation transitoire.

1. L'initialisation en simulation électrique

Plus qu'une initialisation, il s'agit d'une résolution d'équations qui permet de trouver un état cohérent et stable du circuit à étudier. Cette partie s'appelle analyse en courant continu ou DC du circuit. Cette analyse est nécessaire et la simulation transitoire ne peut pas commencer si elle n'a pas convergé.

Pour des circuits où l'analyse DC est difficile, l'utilisateur peut donner des valeurs pour aider à la convergence. Dans d'autres cas, il peut donner explicitement le point de départ DC et ainsi éviter

cette analyse.

2. L'initialisation en simulation numérique

Ici, il s'agit réellement d'une initialisation. En effet, les valeurs sont données par l'utilisateur ou spécifiées par le manuel de référence du simulateur. De plus, une partie de l'initialisation va remplir une liste d'événements.

Pour prendre un exemple, voici le cycle d'initialisation de VHDL. Celui-ci est parfaitement spécifié dans le LRM :

```
1.  la valeur de chaque signal est calculée (en exécutant une
    fonction de résolution si besoin est),
2.  les signaux implicites S'Stable et S'Quiet sont mis à TRUE,
3.  la valeur des signaux implicites GUARD est calculée à partir de
    leur expression,
4.  tous les processus sont exécutés une fois jusqu'à ce qu'ils se
    mettent en arrêt.
```

Après ces quatre étapes la simulation commence, le temps est à 0. Par contre, le premier événement peut être au temps zéro mais il aura lieu au premier délai delta.

Pour VHDL, le seul risque est que l'utilisateur écrive un processus sans instruction **wait**, dans ce cas l'initialisation ne peut pas se finir car le processus boucle sans fin.

3. L'initialisation en mode mixte

Le gros problème de l'initialisation en mode mixte est l'initialisation des noeuds d'interface. En effet les valeurs des interfaces analogiques-numériques sont fixées par le simulateur électrique et celles des interfaces numériques-analogiques par le simulateur numérique. Donc chacun ne peut pas effectuer son initialisation si l'autre ne l'a pas fait!

Pour résoudre ce paradoxe, il faut choisir que l'un des deux s'exécute en premier et prenne pour valeurs aux interfaces concernées des valeurs par défaut. Ensuite, les simulateurs s'appellent à tour de rôle pour prendre en compte les valeurs aux interfaces, jusqu'à ce qu'un état stable soit atteint. Malheureusement cette solution peut boucler, ou bien prendre un temps considérable pour un résultat qui n'est pas forcément excellent. En pratique, on effectue cette boucle un nombre fini de fois et

ensuite la simulation commence même si l'état atteint n'est pas satisfaisant. Dans ce cas, on admet que les premiers pas de simulation gommeront les défauts de l'initialisation.

Voici ce que pourrait être le cycle d'initialisation de VHDL-analogique :

1. la valeur de chaque **signal** est calculée (en exécutant une fonction de résolution si besoin est), les signaux mixtes prennent leur valeur par défaut,
2. les signaux implicites S'Stable et S'Quiet sont mis à TRUE,
3. la valeur des signaux implicites GUARD est calculée à partir de son expression,
4. le simulateur analogique effectue son analyse DC,
5. tous les processus sont exécutés une fois jusqu'à ce qu'ils se mettent en arrêt.

Cette solution est la plus simple, on peut en envisager une autre dans lequel on effectue une boucle tant que la stabilité n'est pas atteinte ou qu'un nombre maximal d'itérations n'est pas dépassé :

1. la valeur de chaque **signal** est calculée (en exécutant une fonction de résolution si besoin est), les signaux mixtes prennent leur valeur par défaut,
2. les signaux implicites S'Stable et S'Quiet sont mis à TRUE,
3. la valeur des signaux implicites GUARD est calculée à partir de son expression,
4. le simulateur analogique effectue son analyse DC,
5. tous les processus sont exécutés une fois jusqu'à ce qu'ils se mettent en arrêt.
6. Si l'état n'est pas stable, c'est à dire des connexions mixtes ont des valeurs différentes coté analogique et coté numérique, alors les processus sont remis à leur état initial, compteur est incrémenté, si compteur < nombre_maximum alors on retourne à l'étape 4, sinon on affiche un message de mise en garde.

VII. Problème d'implémentation

Une fois que les différents choix techniques pour l'algorithme de synchronisation, les interfaces et l'initialisation ont été faits, il faut penser à la réalisation informatique. Deux choix sont possibles.

Le premier consiste à réunir les deux coeurs de simulation dans un même programme et ainsi à n'avoir qu'un seul exécutable. Cette solution présente l'inconvénient d'avoir un exécutable qui peut être volumineux. Par contre, les communications sont très rapides. D'autre part, avec les simulateurs VHDL où un exécutable est généré pour chaque simulation, les temps de génération sont allongés car il faut inclure les bibliothèques du simulateur analogique dans l'édition de lien. Cet

inconvénient peut être limité grâce aux techniques modernes d'édition de liens dynamique.

La seconde solution consiste à avoir deux exécutables et définir un protocole de communication via le système ou au moyen de fichiers pour les échanges de données. Cette solution présente l'inconvénient de faire appel au système et elle s'expose ainsi à des pertes de temps si la machine est chargée.

Dans tous les cas, comme les deux tâches ne peuvent s'exécuter en parallèle, les deux solutions sont équivalentes sauf les temps de communication qui peuvent être plus coûteux entre deux processus qu'entre deux parties d'un même programme.

VIII. Comparaison de différentes implémentations

1. Fideldo

Fideldo [ElT87] est un simulateur mixte analogique-numérique résultat du couplage des simulateurs Fidel [ElT84] et Eldo.

a. Description

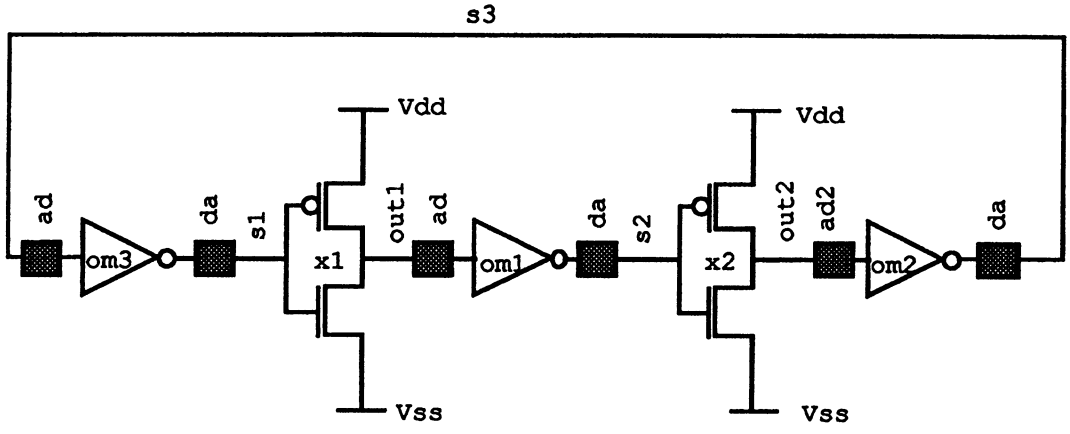
La description numérique est faite avec le langage Fidel : celui-ci permet de couvrir les niveaux d'abstraction suivants :

- interrupteur
- porte logique
- registre
- fonction de transfert en S et en Z [ElT89]
- fonctionnel

Fidel permet aussi de traiter le domaine structurel. Le circuit mixte peut aussi bien être principalement numérique avec quelques bulles analogiques que le contraire. Dans tous les cas, la description complète se fait du côté analogique, les parties numériques apparaissent sous la forme de boîtes noires spéciales. De plus, à chacune des connexions de ces boîtes noires, est attaché un modèle d'interface qui est lui-même pré-défini et seulement paramétré dans le fichier. Du coté numérique,

les connexions destinées à être connectées à l'analogique peuvent être aussi bien de type réel, qu'entier ou bit (booléen).

Regardons un fichier de description correspondant au circuit suivant :




```
* Ring oscillator

* définition des modèles de MOS utilisés
.MODEL MN NMOS level=6 EOX=200e-10 ...
.MODEL MP PMOS level=6 EOX=200e-10 ...

.global 20 30 ! rend visible ces noeud dans les sous-circuits

vdd 30 0 5
vss 20 0 0

* Declaration du sous-circuit
.subckt inverter 1 2
     m1   30   1    2   30   mp   w=200u   l=30u
     m2    2   1   20   20   mn   w=200u   l=30u
.ends inverter

* Instances du sous-circuit
x1   s1   out1   inverter ! en Spice le x en début de ligne
x2   s2   out2   inverter ! indique une instance de sous-circuit

* Instances du modèle Fidel
* les connexions du model fidel sont référencées dans
* l'ordre aprés chaque connexion, on indique le nom de
* l'interface utilisée
om1   out1:ad   s2:da   mod=inv   ! pour Spice étendu Eldo,
om2   out2:ad   s3:da   mod=inv   ! le o en début de ligne signifie
om3     s3:ad2  s1:da   mod=inv   ! instance d'un modèle fidel


* Définition des interfaces
.model ad   atod vth1=1   vth2=4     ! interface analogique-numérique
.model ad2  atod vth=2.5             ! un ou deux seuils à donner
          ! Dans l'autre sens deux valeurs
          ! de tension : une haute, une basse
.model da   dtoa vhi=5    vlo=0   tcom=100n
* Commande de la simulation
.tran 1n 1000n
.plot ...
```

```
...
* Fin de la description Spice
.end

(*
     La description Fidel peut se trouver dans le même fichier
     ou bien dans une librairie
*)
fmodel inv (i,o)
declare input i : logic;
declare output o : logic;
functional
  when i changes
    within 5000 to 5000
      make o=not i
    endwithin;

endfunctional
endmodel
```

Dans cet exemple, trois modèles d'interface sont définis : un pour la conversion numérique vers analogique et deux dans l'autre sens.

b. L'interface

Les interfaces de Fideldo sont les plus simples : il s'agit de la source de tension commandée et de l'impédance infinie. L'utilisateur contrôle certains paramètres. Pour l'interface numérique analogique, il gère les valeurs hautes et basses de la tension de la pile. Et pour l'interface analogique numérique, il a le choix entre un modèle avec un seuil de traduction ou bien un autre avec deux (hystéresis). Ces deux modèles différents sont utilisés dans l'exemple ci-dessus. Il s'agit respectivement des modèles ad2 et ad.

On peut compenser le manque de réalisme de ces interfaces en rajoutant à la description des capacités pour modéliser les différents étages des modèles.

c. Initialisation

L'initialisation se passe en trois phases :

- initialisation de la partie Fidel,
- analyse DC de Eldo avec les valeurs données par Fidel aux interfaces,
- prise en compte des éventuelles modifications des interfaces par Fidel.

Ensuite, la simulation commence. Pour des cas complexes, il peut arriver que l'initialisation ne soit pas finie, c'est à dire que l'état calculé ne soit pas un état stable. Dans ce cas, les premier pas de calculs serviront à finir l'initialisation et ils ne représenteront pas la réponse du système. Il faut ensuite isoler la date où la simulation commence effectivement. Enfin, quelquefois, l'état stable n'est jamais atteint, et les résultats ne sont pas bons. Dans ce cas il faut aider le simulateur à trouver cet état d'initialisation.

2. Cascade

Dans le premier chapitre, nous avons présenté Cascade comme HDL, ici nous le citons comme simulateur multi-niveaux et multi-modes. Comme nous l'avons vu, le HDL Cascade est une union de plusieurs sous-langages qui couvrent chacun un niveau d'abstraction bien précis. De la même manière, le simulateur Cascade est un regroupement de simulateurs qui sont chacun relatif à un des précédents sous-langages. Le simulateur Cascade utilise trois modes de simulation différents selon le langage utilisé :

- discret à séquencement statique, optimum pour les circuits synchrones,
- discret à séquencement dynamique, ou piloté par événements,
- continu.

Pour Cascade, il a fallu définir trois types de synchronisation : le mode continu inclus dans un des deux modes discrets, séquencement statique inclus dans séquencement dynamique et vice-versa.

La synchronisation des modes discrets avec le mode continu se fait par l'algorithme du pas bloqué amélioré. En effet, si le bloc analogique est jugé latent, il peut rendre le contrôle avant la date butoir qui lui a été donnée, et il ne sera réévalué que si une de ces entrées bouge.

Voyons maintenant la synchronisation entre les deux modes discrets. Dans le cas où un bloc à séquencement statique contient un bloc à séquencement dynamique, alors l'échéancier du sous-bloc est consulté à chaque pas et si un événement doit avoir lieu alors le sous-bloc est évalué. Dans le cas contraire, un bloc à séquencement dynamique contient un bloc à séquencement statique, l'algorithme de synchronisation est tout simplement celui du pas bloqué.

On trouvera de plus amples détails sur ces mécanismes dans [Hum84 & LeF85].

En ce qui concerne l'interface mixte, il s'agit de sources controlées en tension dans le sens numérique analogique et du détecteur de seuils avec ajout d'un circuit RC pour le sens contraire.

3. Différents simulateurs mixtes réalisés à partir d'Eldo

La société Anacad possède une bonne expérience en simulation mixte. En effet soit sur une demande d'un client ou tout simplement lors d'une collaboration avec une autre société, Eldo a été couplé avec les principaux simulateurs numériques du marché.

Cette partie a pour but de les présenter succintement, et de faire le tour des différents choix techniques qui ont été faits selon les possibilités du simulateur numérique qui étaient proposées. A la fin, des tableaux récapitulatifs permettront de faire la synthèse sur ces différentes réalisations. Il est à noter que l'algorithme de synchronisation avec retour-arrière n'a jamais été utilisé. En effet, il est jugé comme étant trop inefficace dans la majorité des cas.

a. Mozart-Eldo

Mozart est un logiciel développé par SGS-Thomson pour un usage interne [Gai88]. Il s'agit d'un simulateur numérique qui permet de simuler les niveaux suivants :

- système,
- transfert de registre,
- porte logique,
- interrupteur.

Il possède un langage propre qui traite aussi bien le domaine structurel que comportemental. De plus, celui-ci est hiérarchique et possède la même syntaxe à tous les niveaux et permet de les mixer au sein d'une même description. Enfin Mozart permet de faire de la simulation de faute.

Voyons maintenant les différents choix techniques du couplage Mozart-Eldo [Ben91]. L'algorithme de synchronisation est celui du pas bloqué amélioré, c'est à dire celui ou le simulateur analogique peut dépasser d'un coup la date du prochain événement numérique. Le simulateur

numérique possède une résolution temporelle fixe : la nano-seconde, qui est donnée au départ de la simulation. Lorsqu'Eldo génère un événement mixte la date est arrondie à la nano-seconde supérieure avant d'être passée à Mozart.

La description du circuit à simuler est faite dans le langage de Mozart. Les parties analogiques sont vues comme des sous-circuits de la description numérique. La description de ces sous-circuits analogiques se trouve dans un ou plusieurs fichiers Spice, et est elle-même faite sous forme de sous-circuit. L'analyseur de Mozart génère à partir de sa description et de ces fichiers Spice un fichier pour Eldo. Bien entendu il est impossible d'avoir une description Mozart dans une feuille analogique. Les noms des composants sont passés de l'un à l'autre de manière transparente à l'utilisateur.

Les interfaces AD et DA sont placées soit automatiquement soit explicitement par l'utilisateur. Elles sont totalement paramétrables : valeurs haute et basse, seuils et force pour l'interface AD et le traitement du X, la pente ou les valeurs du circuit RC équivalent entre autre pour l'interface DA. Donc l'interface AD est en fait une traduction sur seuils dont l'impédance coté analogique est infinie. Par contre l'interface DA est un circuit VRC (équivalent Norton des IRC) ou une source de tension commandée avec une pente de montée-descente programmée.

Enfin, un des derniers points intéressants de Mozart-Eldo est la traduction du X. Il peut être traduit de trois façons différentes contrôlables par l'utilisateur :

- par une tension pré-définie,
- par la dernière valeur non indéfinie (1 ou 0),
- par une invalidation des résultats analogiques, c'est à dire que tant qu'un X se trouve sur une interface DA, alors les interfaces AD transmettent X.

Chaque solution a son intérêt, si les deux premières sont des solutions souvent employées et semblent assez naturelles (surtout la première), par contre la troisième peut surprendre. En fait, elle sert à propager l'état X comme cela est le cas du côté numérique.

b. Qsim-Eldo

Qsim est un simulateur numérique qui fait partie de l'environnement de CAO de Compass. Cet environnement comprend une chaine numérique complète qui va de la saisie de schéma jusqu'au layout. En fait son HDL n'est pas utilisé directement, il est en effet plus agréable d'utiliser la saisie de schéma et le générateur de netlist.

Eldo a donc été intégré à cet environnement. La description du circuit se fait toujours en utilisant une saisie de schéma. Ensuite un outil se charge de séparer les parties destinées à Qsim et celles destinées à Eldo, et il introduit les interfaces. Cependant, il est possible d'éditer les fichiers de description et de modifier leur place. Comme pour les autres simulateurs mixtes récents d'Anacad, le logiciel propose les deux sortes d'interfaces, les simples comparateurs et sources commandées et les circuits à modélisation de charge (IRC et RC).

Un des points particulier de Qsim-Eldo est de posséder une interface bi-directionnelle. Il s'agit d'une interface qui est soit analogique-numérique soit numérique-analogique selon un certain nombre de paramètres, dont l'un est l'estimation de la différence de potentiel. Il ne s'agit pas d'une véritable interface bidirectionelle qui introduit un couplage fort entre les deux simulateurs, avec un noeud inout du coté numérique. La synchronisation des deux coeurs de simulation est assurée par l'algorithme du pas-bloqué.

c. Hilo-Eldo

Hilo [Fla81] est un simulateur qui a d'abord été développé à l'université de Brunel (Royaume Uni) avant d'être repris par Genrad. Le couplage Hilo-Eldo est très proche de celui fait pour Fideldo. Une des grosses différences est que dans cette approche c'est Hilo qui pilote Eldo et non le contraire.

Pour la description, chaque langage décrit sa partie, et la liaison se fait à l'élaboration du modèle, comme pour Fideldo. Les interfaces sont les plus simples : les comparateurs et les sources de tension commandées. Comme pour le précédent, l'algorithme de synchronisation est le pas-bloqué.

d. Lsim-Eldo

Le simulateur Lsim est un simulateur multi-niveau qui fait parti de l'environement de CAO-micro-electronique GDT. Il est distribué par Mentor-Graphics. Il possède deux langages de description propres qui sont le langage N pour la description structurelle et le langage M pour la description fonctionnelle. Lsim-Eldo est utilisé entre autre, par le CNET-Grenoble.

La description mixte se fait en utilisant le langage N même pour les parties analogiques. Ensuite, l'analyseur va générer une description Spice pour Eldo. Toutefois, un traducteur N vers Spice existe et permet donc de récupérer les circuits existants. Les interfaces sont placées automatiquement.

Le simulateur mixte se présente sous la forme de deux exécutables communiquant par des *pipes*, cependant les deux processus ne fonctionnent pas en parallèle. En effet, le mode de synchronisation n'est pas celui avec retour arrière, il s'agit du saut de grenouille.

e. Verilog-Eldo

Verilog est le simulateur de Cadence, la première compagnie mondiale de CAO micro-électronique. Le couplage s'est fait sur une demande de SGS-Thomson qui utilise les deux simulateurs et qui voulait donc pouvoir faire de la simulation mixte. Ce simulateur n'est donc pas commercialisé officiellement, et est destiné à l'usage interne de ST.

La coopération entre les deux sociétés n'a pas été très profonde et la réalisation n'est donc pas optimale. Ainsi, pour des raisons de secrets technologiques, il n'a pas été choisi un couplage intime entre les deux simulateurs mais une communication au travers du système (socket).

Quant aux interfaces, elles sont placées automatiquement, les noeuds mixtes sont dédoublés et cela peut poser des problèmes pour certains circuits complexes. Ces interfaces sont de types IRC et RC, car le simulateur Verilog manipule des logiques multi-valuées qui sont sources de conflit d'impédance.

f. Tableaux comparatifs

Le premier tableau, ci-après, indique les choix fait pour l'interface mixte. Le second présente le

choix de la réalisation. Enfin, le dernier rappelle les méthodes de description.

Simulateurs	Interface AD		Interface DA		Synchronisation	
	Simple	Circuit RC	Source Commandée	Circuit IRC	saut de grenouille	pas bloqué
Mozart-Eldo	X			X		X
Lsim-Eldo		X		X	X	
Qsim-Eldo	X	X	X	X		X
Verilog-Eldo		X		X		X
Hilo-Eldo	X		X			X

Simulateurs	Un exécutable	Echange via le système
Mozart-Eldo	X	
Lsim-Eldo	X	
Qsim-Eldo	X	
Verilog-Eldo		X
Hilo-Eldo		X

On constate que la solution consistant à unifier les deux environnement de simulation est la plus employée. En effet, cette solution présente l'avantage de se dégager des problèmes système tels que la communication entre deux processus qui peut vite devenir coûteuse. D'autre part, la méthode de synchronisation ne permettant pas une exécution concurrente, cela n'est pas utile. Donc seuls Verilog-Eldo et Hilo-Eldo se présentent sous cette forme. La raison du choix de cette méthode pour ces deux derniers est uniquement commerciale; ainsi un client ayant une licence sur plusieurs machines pour le simulateur numérique, peut avoir le simulateur mixte sur toutes celles-ci en n'acquérant une licence de Eldo que sur une seule.

Simulateurs	Description du circuit
Mozart-Eldo	La description du circuit se fait dans le langage Mozart, les parties analogiques sont vues comme des sous-circuits, leur description Spice se trouve dans des fichiers sous forme de sous-circuit, un fichier Spice est généré pour Eldo
Lsim-Eldo	Toute la description se fait en langage N (Lsim), à partir de celle-ci un fichier Spice est généré pour Eldo. Toutefois il existe un traducteur Spice vers N pour utiliser les descriptions existantes.
Qsim-Eldo	La description est gérée automatiquement par une saisie de schéma.
Verilog-Eldo	Deux descriptions sont nécessaires, une en Spice l'autre en Verilog. Les interfaces sont définies du coté Eldo, mais on doit les indiquer en Verilog
Hilo-Eldo	idem verilog

4. Mint-Sugar

L'intérêt de ce simulateur [Alt91] est d'être un simulateur mixte dont la partie numérique est un simulateur VHDL [Alt90]. Il est donc intéressant de voir quels choix ont été faits du point de vue de la description. Tout d'abord, il faut noter que le simulateur analogique Sugar [Pat90] est un simulateur à événements et que cela facilite grandement les problèmes de synchronisation : en effet, les mécanismes de simulation sont identiques à ceux du simulateur VHDL et on peut connaitre la date du prochain pas de calcul. Quant à la description analogique elle est faite avec le langage Spice.

a. La description

La description du circuit se fait en VHDL. Aussi, un **paquetage** analogique a été écrit. On y trouve ce qui est nécessaire pour indiquer qu'une **architecture** relative à une **entité** est en fait un composant analogique. Ce paquetage sera présenté en détail plus loin.

Donc, le circuit est décrit de manière complète en VHDL, mais les parties analogiques

apparaissent comme des entités dont l'architecture est vide. La description fine est faite en utilisant le langage de description du simulateur Sugar, qui est en fait un compatible Spice. Le simulateur VHDL doit juste savoir quels signaux seront mixtes et dans quel sens aura lieu l'interaction.

L'utilisation de VHDL pour décrire le circuit dans son ensemble, permet de mener la conception de manière hiérarchique et d'utiliser toute la souplesse et la puissance de VHDL dans ce domaine.

Une entité analogique possède des ports, qui sont ses connexions réelles, et tout comme ceux des entités numériques, ils possèdent un mode. Ces modes sont au nombre de trois et ils définissent la direction des interactions mixtes :

- in : pour numérique analogique,
- out : pour analogique-numérique,
- linkage : pour analogique-analogique, pour éviter des conversions inutiles dans le cas où deux composants analogiques sont connectés ensemble.

Quant au type, il s'agit d'un type utilisateur appelé voltage, qui est en fait un sous-type du type réel. Ces ports sont connectés à des ports d'entités numériques en passant par une fonction de conversion appropriée et pré-définie, la traduction des valeurs se fait en interne de manière cachée à l'utilisateur. Cependant, ce dernier a accès à des attributs (slew et cap) définis dans le paquetage analogique qui permettent de modifier le temps de montée des interfaces DA et la valeur de la capacité attachée aux interfaces AD.

Le paquetage analogique est court , aussi le voici en entier :

```
package analog is
    subtype voltage is real range real'low to real'high;
    subtype slewval is real range 0.0 to real'high;
    subtype capval is real range 0.0 to real'high;

    constant slewdef : slewval := slewval'low;
    constant capdef : capval := capval'low;

    attribute sugar_file_name : string;
    attribute slew : slewval;
    attribute cap : capval;
end analog;
```

L'attribut sugar_file_name est l'attribut qui permet de donner le nom du fichier dans lequel on peut trouver la description Spice du sous-circuit analogique. Le point le plus important est de voir que de la sémantique a été ajoutée au langage, cela fait qu'en théorie ce simulateur ne répond pas à la norme VHDL'87. Mais pour faire de la simulation mixte en utilisant VHDL, cela est nécessaire car rien n'est défini dans le manuel de référence du langage.

Regardons un exemple de description d'un couple entité-architecture d'un sous-circuit analogique :

```
-- Sous-circuit amplificateur

library a_std;
use a_std.analog.all;

entity amp is
    generic ( sinp : slewval := slewdef;
              cout : capval := capdef);
    port    ( inp : in voltage;
              outsig : out voltage := 0.0);
end;


architecture circuit of amp is
    attribute sugar_file_name of circuit :
        architecture is "amp.spi";
    attribute slew of inp : signal is sinp;
    attribute cap of outsig : signal is cout;
begin
end;
```

b. Initialisation

Dans un premier temps, les valeurs d'initialisation de VHDL sont calculées, et Sugar dans un deuxième temps effectue une analyse DC en tenant compte des valeurs d'initialisations de VHDL aux interfaces. Les valeurs de cette analyse sont propagées au travers des interfaces du côté VHDL, et si nécessaire des signaux sont réévalués. Cette opération n'est faite qu'une fois, ensuite les processus VHDL sont exécutés une fois comme cela est indiqué dans le LRM.

Cette solution n'est pas optimale. D'ailleurs, dans [Alt91] il est dit avec raison que quelques itérations transitoires sont nécessaires avant que les résultats de simulation soient corrects. Cet aspect est le problème principal de l'initialisation en mode mixte.

c. Interfaces

Mint-Sugar ne propose qu'un seul modèle d'interface par type. Pour l'interface AD, il s'agit d'une capacité rajoutée au circuit analogique dont la valeur est contrôlée par l'utilisateur, mais qui ne dépend pas de l'état du système. Quant aux contrôles des seuils, l'article ne les mentionne pas, et donc il faut supposer qu'ils sont fixés par l'implémentation. Pour l'interface DA, il s'agit du modèle de la pile commandée par la valeur du signal numérique.

d. Synchronisation

Profitant du fait que le simulateur continu Sugar est géré par événements, la synchronisation du simulateur est différente de celles présentées auparavant qui fonctionnent pour le cas général.

En fait à chaque pas de calcul, le contrôle est donnée au simulateur dont l'événement doit arriver en premier. Dans le cas ou un événement mixte arriverait, chaque simulateur peut modifier la liste d'événements de l'autre en propageant des événements mixtes. Dans le cas d'un événement numérique-analogique, par exemple un passage de 0 à 1, la modification de la tension avec la pente voulue est immédiatement prise en compte par Sugar. Enfin dans l'autre cas, pour un franchissement de seuil, Sugar ne calcule pas la date exacte du franchissement du seuil en reculant et en extrapolant cette date, mais il donne la date à laquelle il se trouve présentement en considérant que l'erreur introduite est faible, c'est à dire que ses pas sont assez petits par rapport à ceux du simulateur VHDL.

e. conclusion

La réalisation de cet outil nous montre qu'il est possible de coupler un simulateur VHDL et un simulateur analogique. Cependant, au niveau de la description on est obligé d'introduire des artifices pour permettre d'introduire le niveau électrique, qui reste quant à lui décrit avec le langage Spice.

IX. Conclusion

Au travers de ce chapitre, nous avons abordé différents points qui ont permis de mettre en valeur

les principales caractéristiques de la simulation mixte. L'un des gros problèmes de la simulation mixte est en fait la description et l'interface. Les deux sont en relation car c'est la limitation au niveau de la description qui rend les interfaces non accessibles à l'utilisateur. Il semble d'ailleurs que la non utilisation de la simulation mixte soit liée à ce problème. En effet, la description n'est pas standard et il est difficile de gérer la conception d'un même circuit avec deux description différentes. Heureusement, l'arrivée du langage VHDL-analogique va permettre d'éclaircir tout cela et rendra la simulation en mode mixte plus attrayante. En effet, non seulement l'utilisateur aura un contrôle total de la description, mais il aura en plus accès à une gestion de bibliothèque.

L'autre point est la synchronisation entre les deux noyaux de simulation. L'arrivée de VHDL-analogique ne devrait rien imposer. Cependant, parmi ceux que nous avons présentés, il semble que celui du pas-bloqué soit le plus flexible. Bien entendu, à terme, il faut y intégrer la modification qui permet d'éviter le cas critique où le simulateur analogique a des pas plus grands que le simulateur numérique.

Chapitre IV.
Réalisation Pratique

I. Introduction

Dans ce chapitre, nous présentons la réalisation du simulateur mixte VHD$_e$LDO. Ce simulateur est basé sur Eldo et Mint. Le premier paragraphe est donc une présentation de ces deux simulateurs.

Comme nous l'avons vu dans le chapitre précédent, Mint a déja été couplé avec un simulateur analogique : Sugar. Il existe deux différences notables entre ces deux simulateur mixtes.

La première est une différence entre les deux simulateurs analogiques. Sugar est en fait un simulateur analogique piloté par événement, alors qu'Eldo est un simulateur analogique plus classique. Ainsi, Sugar connaîtra des limitations pour les circuits fortement couplés. Comme nous l'avons déja dit, cette particularité de Sugar rend la synchronisation avec Mint beaucoup plus facile.

Une seconde différence existe au niveau de la description. Nous la présenterons plus en détail dans le second paragraphe.

II. Présentation des deux simulateurs

1. Eldo

Eldo [Hen85] est un simulateur électrique de troisième génération. Il utilise conjointement l'algorithme de Newton et OSR (One Step Relaxation) : un algorithme de relaxation dérivé de Gauss-Seidel. Ce dernier donne de très bons résultats pour les circuits numériques à base de transistors MOS. Cela permet à Eldo de pouvoir simuler des circuits approchant la centaine de milliers de transistors, le plaçant ainsi devant ses concurrents. Néanmoins, pour les circuits purement analogiques et fortement couplés, qui sont résolus par Newton, il est quasiment équivalent aux dérivés industriels de Spice. L'utilisation conjointe de ces deux algorithmes lui permet de minimiser leurs inconvénients. Eldo possède aussi un algorithme lui permettant de simuler les

circuits MOS numériques à un niveau pseudo-électrique, c'est une sorte de simulation *Timing* [Bes91]. Les descriptions ainsi que l'algorithme de simulation utilisés sont simplifiés et de ce fait cela permet d'obtenir un fort gain de temps en perdant toutefois en précision.

Aujourd'hui le CNET Grenoble et Anacad poursuivent une politique de recherche et continuent de développer Eldo. Par exemple, actuellement, des études sont faites pour intégrer une analyse de bruit en mode transitoire [Bol92] en coopération avec le LETI, ainsi qu'un mode d'analyse mixte fréquentiel-transitoire.

Le langage d'entrée est celui de Spice. On trouve en plus des primitives privées telles que divers macro-modèles, parmi lesquels des amplificateurs opérationnels, des portes numériques ou bien encore des convertisseurs analogique-numérique ou numérique-analogique. De plus, il existe le langage Fas qui permet la description comportementale de modèles analogiques.

Eldo permet de simuler un circuit avec deux précisions différentes, une forte pour la partie critique et une plus faible pour le reste. Cette option trouve son champ d'application pour les circuits mixtes où la partie numérique même au niveau transistors ne nécessite pas une forte précision pour converger. C'est pourquoi cette option s'appelle Eldo-mode-mixte.

2. Mint

Mint [Alt90] est un simulateur VHDL développé par l'institut de micro-électronique suédois et intégré dans leur environnement VHDL : VE. Malgré un bon état d'avancement du travail, cet institut n'a pas pu mener ce simulateur à l'état de véritable produit industriel. En effet, cela demande des coûts et un travail de maintenance trop lourd pour un institut de recherche. Il a cependant été utilisé par quelques sociétés en Suède. Ceci montre d'une part que l'état d'avancement du simulateur est bon, et d'autre part que ses performances sont correctes pour supporter une exploitation industrielle.

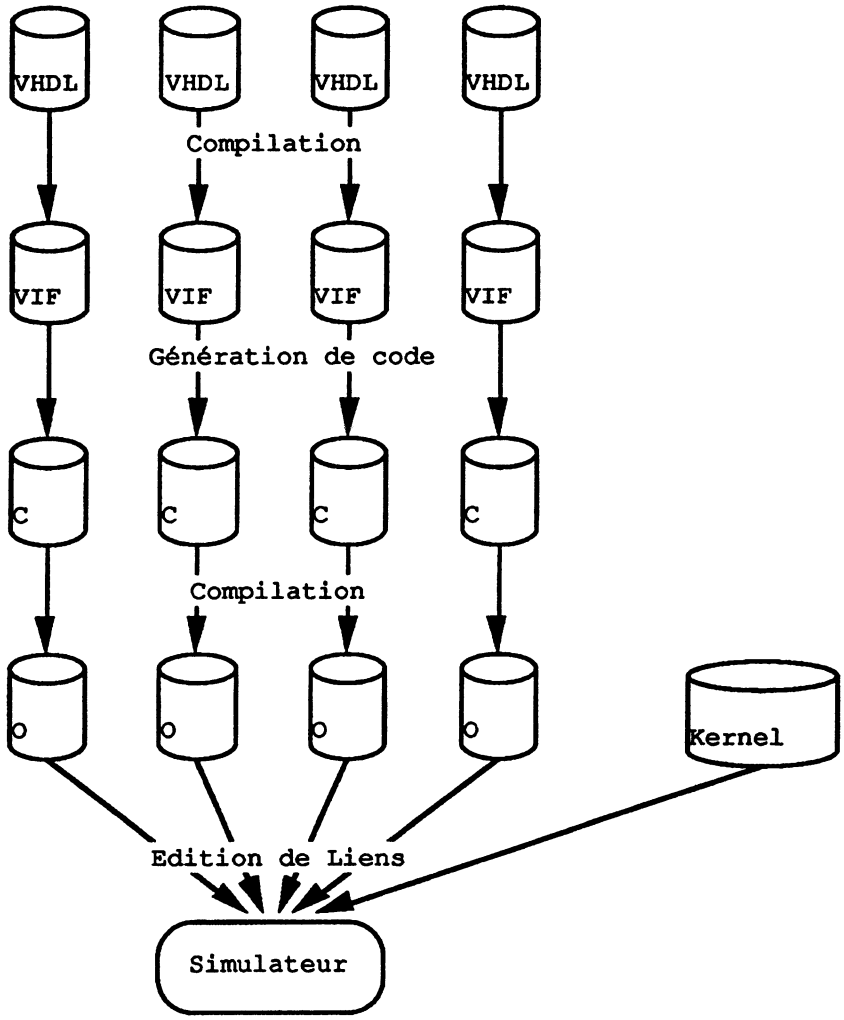
Tout le programme a été écrit en C. Le simulateur est un simulateur compilé, c'est à dire qu'il faut générer un exécutable pour chaque simulation. Cette solution présente une certaine lourdeur au niveau du temps de compilation, mais possède l'avantage d'avoir des temps de simulation beaucoup

plus courts que les réalisations interprétées.

Le code VHDL est traduit en C en passant par un format intermédiaire. Le format utilisé dans VE est celui que CLSI avait proposé pour la normalisation mais qui n'a pas été retenu.

Le programme comporte trois parties principales : un élaborateur, un coeur de simulation et un ensemble de fonctions utilisables dans le code généré qui sont les fonctions de base VHDL : opérateurs arithmétiques et logiques, fonction now ou encore les fonctions du paquetage textio. L'élaborateur et le coeur de simulation sont conformes au manuel de référence. Quant à la structure de données, elle est très complexe et ceci pour pouvoir stocker tous les liens qui existent entre les différents objets VHDL. Par exemple, la structure dans laquelle est stocké le signal, permet de référencer tous les processus sensibles sur ce signal.

En ce qui concerne l'ensemble VHDL toléré par cet environnement, on peut dire que pour la version actuelle il y a des lacunes au niveau de la gestion des vecteurs et aussi quelques petits problèmes pour certains type de processus. Néanmoins, une nouvelle version est actuellement en phase de validation, et normalement ces problèmes sont résolus. Dans tous les cas, les grands principes VHDL sont présents, et les quelques problèmes ne remettent pas en cause notre approche de simulation mixte.



Principe d'un simulateur VHDL compilé

Pour la partie programmation, au vu du code source du programme, on peut dire que celui-ci est facilement maintenable, car la programmation est très claire et très commentée. Toutes les modifications faites pour le mode mixte ont pu être faites sans difficulté.

Enfin, en ce qui concerne les performances, on peut dire qu'elles sont actuellement moyennes. La rapidité d'un simulateur VHDL réside dans la qualité du code C généré, et dans l'ingéniosité de ses structures de données. Pour l'instant, le code C généré n'est pas optimisé. Ceci peut paraître surprenant, mais en fait cela semble raisonnable pour un prototype quand on connaît la complexité de VHDL. En effet, avant de penser à l'optimisation, il faut valider le prototype et laisser cette optimisation pour une phase ultérieure. Ainsi, tous les mécanismes comportementaux ont été traités de la même manière sans aucun cas particulier, il est de même pour la gestion des types.

L'objectif de VHD$_e$LDO aurait pu être la réalisation d'un proto-VHDL-Analogique. Ce travail aurait rajouté un effort important au niveau compilation et n'aurait pas pu être mené à bien par une seule personne. C'est pour cette raison, que nous nous sommes concentrés sur l'interface mixte. La complexité de VHDL est telle que l'écriture d'un compilateur nécessite deux homme-années et de très bonnes connaissances en compilation.

3. Les gains en temps de simulation

Le temps de simulation d'un circuit avec Eldo est linéaire avec le nombre de noeuds du circuit, seule la constante de proportionnalité peut varier : elle est plus importante avec des transistors bipolaires qu'avec des transistors MOS.

D'autre part, le passage simulation électrique simulation numérique VHDL, fait gagner un facteur qui va de mille jusqu'au million.

De ces deux constatations il en découle la règle suivante pour le cas général : le gain en temps de calcul pour une simulation mixte est égale au rapport du nombre de transistors total du circuit sur le nombre de transistors qui vont être simulés au niveau électrique en simulation mixte.

En conclusion, on peut dire que la simulation mixte sera intéressante sur des circuits dont la partie numérique est grosse devant la partie analogique.

III. La description

1. Présentation

Nous avons vu dans le premier chapitre que mis à part le projet Cascade, il n'existe aucun HDL mixte à ce jour. Il faut donc utiliser au moins deux langages de description : un pour la partie analogique et un pour la partie numérique.

Deux méthodes existent pour parvenir à faire une description cohérente à partir de deux langages différents :

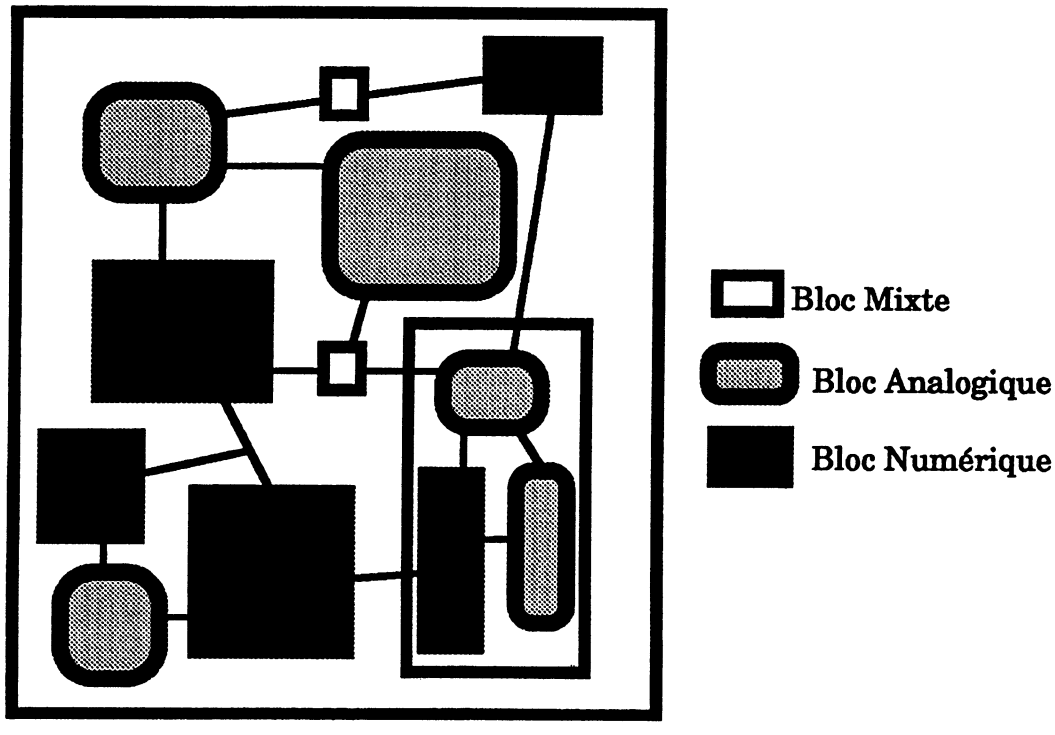
- pour la première, chaque partie est décrite en utilisant son langage et certaines

connexions sont déclarées mixtes dans chacune de ces deux descriptions grâce à des constructeurs spécifiques. Ces constructeurs ont été rajoutés dans le langage en vue de cette description mixte. Enfin, chaque noeud mixte connaît son homologue dans l'autre description.

- pour la seconde solution, le circuit est globalement décrit dans un des deux langages, et certaines feuilles de la description structurelle sont déclarées comme étant de l'autre langage. La description de cette feuille est alors composée d'une référence à un sous-circuit de l'autre langage.

C'est la seconde approche que nous avons choisie. Ainsi, la description du circuit se fait grâce à VHDL, et donc, dans cette description certains couples **entité-architecture** sont en fait des sous-circuits du langage Spice.

Avec notre approche, on a accès à toute les possibilités de description structurelle de VHDL. Le schéma suivant nous montre un exemple de ce qu'il est possible de faire :



Dans une description pour VHD$_e$LDO, on trouve trois classes de blocs (au sens VHDL) : numériques, mixtes et analogiques. Les blocs numériques sont des blocs VHDL classiques. Les blocs mixtes sont des blocs composés de blocs numériques, analogiques ou mixtes, et encore de

processus ou d'instructions VHDL classiques. Les blocs analogiques sont soit des blocs qui référencent un sous-circuit Spice, soit des blocs qui sont composés uniquement d'une interconnexion de blocs analogiques.

Globalement, il est possible de faire le même genre de description avec Mint-Sugar. Cependant comme nous le verrons plus loin, la solution utilisée pour la description est moins claire que celle de VHD$_e$LDO.

2. Les méthodologies de description possibles

L'utilisation d'un HDL pour une description donne lieu, tout comme l'écriture d'un programme, à un style propre à chacun. Ainsi, pour ce qui est de VHDL, certains passeront par des composants configurés par défaut, d'autres utiliseront les unités de configuration ou encore des appels de procédure concurrents.

La solution que nous avons choisie pour VHD$_e$LDO [ElT93] au niveau de la description permet de suivre deux principales méthodes de description pour un circuit mixte. Nous présenterons ces deux méthodes dans le sous-paragraphe 3.

Comme nous venons de le voir, des couples entités-architectures peuvent être des circuits analogiques qui seront simulés par Eldo. Dans cette partie, nous allons étudier comment indiquer qu'un tel couple représente un circuit électrique, et comment le connecter à la description numérique.

Pour introduire des nouvelles informations sur des objets, nous avons utilisé un mécanisme très pratique en VHDL : les attributs pré-définis. Il nous a fallu aussi définir des nouveaux types pour pouvoir identifier les objets analogiques. Toutes ces définitions ont été faites dans un paquetage qui a été intégré dans une bibliothèque qu'il est nécessaire de référencer pour faire de la description avec le simulateur mixte VHD$_e$LDO.

a. Entité-Architecture *analogique*

On indique qu'un couple entité-architecture représente un sous-circuit analogique en utilisant

l'attribut pré-défini **analog_file_name**. Cet attribut est attaché à l'architecture. Son contenu est une chaîne de caractères qui est le nom du fichier dans lequel se trouve la description du sous-circuit Spice. Bien entendu, il est préférable de donner le chemin système complet, sinon la recherche se fera toujours dans le répertoire de travail. Il est également possible d'utiliser des variables d'environnement système.

```
attribute analog_file_name : string;
```

Le nom du sous-circuit Spice doit être le même que celui de l'entité, et les connexions sont associées les unes aux autres dans l'ordre d'écriture.

Ce choix est identique à celui de Mint-Sugar.

b. Les connexions

En VHDL, un **port** possède un **mode** et un **type**. Le mode indique les relations du **port** avec l'extérieur. Il peut être **in, out, inout** ou **linkage**. Ce dernier est un mode spécial qui a été introduit pour pouvoir, semble-t-il, *lier* le simulateur VHDL avec le monde extérieur. Toutefois, il reste d'une utilisation marginale, et son utilisation rend les descriptions non simulables par un simulateur VHDL classique.

Pour notre cas, un couple **entité-architecture** est la représentation d'une partie analogique s'il existe l'attribut analog_file_name sur l'**architecture**. Dans ce cas, ses **ports** sont soit des connexions mixtes pouvant être connectées directement à des **signaux** numériques, soit une connexion analogique. Nous verrons plus loin les détails de cette option qui ouvre les portes à toute la puissance de la description structurelle VHDL.

i. Les connexions mixtes

Pour les connexions mixtes, le mode du port nous donne le type de l'interaction mixte :

- **in** pour le sens analogique-numérique
- **out** pour le sens numérique-analogique

D'autre part, le type du port nous indique le type du signal sur lequel on pourra connecter ce port. C'est à dire ce vers quoi la tension va être traduite, ou ce qui va la contrôler. Ces types sont vérifiés, et

donc ne sont pas laissés libres à l'utilisateur. En voici la liste complète :

- bit, du paquetage standard ('0', '1'),
- std_logic, type normalisé par l'IEEE ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'),
- voltage, un type réel.

Il est également possible d'utiliser les tableaux de dimension quelconque, dont l'élément de base est d'un de ces trois types.

Entre le signal numérique et le noeud analogique, une interface est introduite implicitement. Celle-ci est paramétrable par l'utilisateur. Le nombre de paramètres dépend du type du signal numérique et de la direction de l'interaction. Ainsi pour l'interface voltage dans le sens numérique-analogique (nombre réel -> tension) un seul paramètre est nécessaire, alors que pour l'interface std_logic dans le même sens il en faut dix: tensions haute et basse, temps de commutation, trois valeurs de résistances et de capacités pour les différentes forces. Ces paramètres sont donnés par l'utilisateur dans la description en utilisant des attributs prédéfinis sur le port.

Voici le paquetage analog dans son entier :

```
--=================================================================
-- This package is in VHDLDO library.
--
-----------------------------------------------------------------
-- MIXED interface package specification.
-----------------------------------------------------------------

USE work.ipl.all;
USE work.analog_interface.all;

PACKAGE analog IS

-----------------------------------------------------------------
-- Predefined ATTRIBUTEs
-----------------------------------------------------------------
  -----------------------------------------------------------------
  -- analog netlist file reference
  -----------------------------------------------------------------
  ATTRIBUTE analog_file_name : string;

  -----------------------------------------------------------------
  -- parameters on analog input ports for BIT & 'MVL9', DtoA
  -----------------------------------------------------------------
  ATTRIBUTE a_low_voltage  : real;
  ATTRIBUTE a_high_voltage : real;
  ATTRIBUTE a_fall_time    : real;
  ATTRIBUTE a_rise_time    : real;

  -----------------------------------------------------------------
```

```
  -- parameters on analog output ports for BIT & 'MVL9', AtoD
  ------------------------------------------------------------------
  ATTRIBUTE a_vth_rise : real;
  ATTRIBUTE a_vth_fall : real;
  ATTRIBUTE a_load_cap : real;

  ------------------------------------------------------------------
  -- parameters on analog input ports only for REAL, DtoA
  ------------------------------------------------------------------
  ATTRIBUTE a_tcom     : real;

  ------------------------------------------------------------------
   -- parameters on analog output ports only for REAL, AtoD
  ------------------------------------------------------------------
  ATTRIBUTE a_threshold : real;

  ------------------------------------------------------------------
  -- parameters on analog input ports only for 'MVL9', DtoA
  ------------------------------------------------------------------
  ATTRIBUTE a_r_strong : real;
  ATTRIBUTE a_r_weak   : real;
  ATTRIBUTE a_r_z      : real;
  ATTRIBUTE a_c_strong : real;
  ATTRIBUTE a_c_weak   : real;
  ATTRIBUTE a_c_z      : real;

------------------------------------------------------------------
-- Types : Voltage and volt
------------------------------------------------------------------
  TYPE voltage IS RANGE real'low TO real'high;

  TYPE voltage_vector IS ARRAY (natural RANGE <>) OF voltage;

  ------------------------------------------------------------------
  -- resolution function
  ------------------------------------------------------------------
  FUNCTION res_voltage (v : IN voltage_vector) RETURN voltage;
  ATTRIBUTE foreign OF res_voltage : FUNCTION IS "ai_ResolveVoltage";

-- Resolved voltage
  SUBTYPE volt IS res_voltage voltage;

  TYPE volt_vector IS ARRAY (natural RANGE <>) OF volt;

------------------------------------------------------------------
-- Global signal for analog...
------------------------------------------------------------------
  SIGNAL a_gnd : volt;

END analog;
```

C'est ici que réside la différence entre Mint-Sugar et VHD$_e$LDO. Dans Mint-Sugar, les connexions d'un composant analogique sont obligatoirement de type voltage. Le mode indique la direction de l'interaction mixte, mais pour connecter un port avec un signal de type BIT, il faut utiliser une fonction de conversion. Les paramètres de l'interface mixte sont aussi donnés au moyen d'attribut pré-définis. Cette solution présente l'inconvénient d'être moins lisible que la nôtre.

D'autre part, cette solution crée une ambiguïté au niveau de la description VHDL. En effet, un signal de type voltage pourra être connecté à un port de mode in ou out et de type voltage sans créer d'erreur syntaxique VHDL, mais en créant une erreur sémantique car ces ports représentent des connexions mixtes. De plus, cette erreur ne sera relevée que par l'élaborateur. Ceci peut être une source d'erreur pour les utilisateurs.

ii. Les connexions analogiques

C'est ce type de connexion qui permet d'accéder à la description structurelle VHDL pour la partie analogique et ainsi de définir des blocs analogiques qui ne référencent pas de description Spice. L'utilisateur n'est donc pas obligé de regrouper cette partie dans un ou deux gros sous-circuits pour les connecter à la partie numérique, il peut introduire plusieurs niveaux de hiérarchie et faire appel, par exemple, à des entités-architectures structurelles ne comportant que des composants analogiques. Dans ce cas particulier, l'entité-architecture ne référence aucun fichier puisqu'elle n'est pas terminale.

La connexion analogique est un port de mode linkage et de type *volt*, ou de type tableau dont l'élément de base est de type *volt*. Le fait d'imposer ce type permet d'éviter l'utilisation anarchique du mode linkage.

Pour réaliser la description structurelle, les ports de ce mode sont connectés à des signaux de type volt. Ces derniers n'existent du côté VHDL que pour la description, ils représentent en fait les noeuds analogiques, aucun événement n'intervient sur eux durant la simulation.

Comme nous le verrons au travers des exemples, cette solution nous ouvre les portes de la description structurelle VHDL pour la partie analogique. Ainsi, on a accès aux instructions de générations répétitives et conditionnelles, à la généricité pour des descriptions structurelles, ou encore aux unités de configuration.

Les entités-architectures analogiques, qui ne possèdent que des connexions analogiques, peuvent être connectées à la partie numérique au moyen d'un composant d'interface explicite. Ce type de composant possède deux connexions, une analogique et une mixte. Leur sous-circuit analogique Spice associé est un fil connectant les deux connexions. Enfin, il est possible de spécifier les valeurs

relatives aux attributs de l'interface mixte, et cela au moyen des paramètres génériques des interfaces.

Avec VHD$_e$LDO, il existe une bibliothèque (`vhdldo_mixed`) dans laquelle on trouve un paquetage de composants et les entités-architectures correspondant à ces interfaces, pour chacun des types mixtes et dans les deux sens. Ces composants évitent d'avoir à modifier une description d'un composant analogique pour pouvoir le connecter à un signal numérique.

c. La généricité

En VHDL, une architecture peut être paramétrée grâce à la clause générique. Ces paramètres peuvent être aussi bien des délais que des tailles de bus de données.

Dans le langage Spice, un sous-circuit peut aussi avoir des paramètres. Mais le concept est moins puissant, ainsi il n'est pas question de pouvoir contrôler le nombre de connexions.

Cependant, nous avons fait un parallèle entre ces paramètres. Ainsi, une entité analogique qui est l'image d'un sous-circuit Spice, peut référencer dans sa clause générique les paramètres de son sous-circuit Spice. Les paramètres doivent avoir le même nom. D'autre part, il doit y en avoir le même nombre de chaque côté. Un dernier point, ces paramètres génériques ne peuvent être que de type réel. Cette limitation vient du langage Spice. Ainsi nous avons proposé la possibilité d'utiliser le type chaîne de caractères.

Voici un exemple d'entité-architecture générique :

```
entity resistor is
    generic ( res : real := 1.0e3);
    port (p, n : linkage volt);
end resistor;

architecture analog of resistor is
    attribute analog_file_name of analog :
        architecture is "cir.ckt";
begin
end analog;
...
   Dans le fichier cir.ckt on trouve :
.subckt resistor n1 n2
    R1 n1 n2 res        -- il y a correspondance entre le nom
.ends                   -- VHDL et le nom Spice du paramètre
```

Par contre, pour une description analogique structurelle, il est possible de paramétrer la taille

d'un bus de connexion. Nous verrons plus loin au travers d'un exemple comment cela peut être fait.

Dans Mint-Sugar cette notion n'existait pas. L'utilisation de la description structurelle analogique était alors limitée. En effet, si un circuit utilise plusieurs capacités de valeurs différentes, alors il faut plusieurs entité-architectures. On peut donc conclure que sans cette caractéristique il est impossible d'utiliser les concepts de description structurelle VHDL.

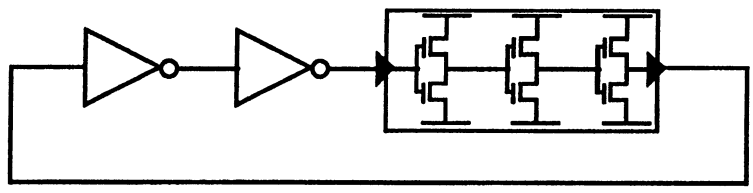
3. Les différentes possibilités de description

Comme en programmation, la description donne lieu à des résultats qui diffèrent selon les personnes. Pour illustrer ceci, voyons comment peut être décrit un anneau de cinq inverseurs dont trois successifs sont décrits au niveau électrique.

a. Première solution

La description des trois inverseurs analogiques va se faire dans un seul sous-circuit, qui est d'ailleurs un inverseur. Ensuite, Il faut créer un couple entité-architecture qui va le référencer, et dont les deux connexions sont déclarées de type bit, l'une de mode in et l'autre de mode out. De plus, on paramètre correctement les interfaces au moyen des attributs pré-définis. Enfin on réalise une description globale pour la simulation.

On a alors le schéma suivant :



L'inconvénient majeur de cette approche est qu'elle n'utilise pas assez la hiérarchie. De plus, une modification du circuit risque d'entraîner des modifications dans les deux descriptions.

b. Seconde solution

De manière plus hiérarchique, on va mettre la description d'un inverseur analogique dans un seul sous-circuit, et on va alors créer un couple entité-architecture relatif à celui-ci. Pour choisir le type des connexions, regardons le circuit. Si on prend le type BIT pour le port d'entrée et de sortie, on

va avoir deux doubles traductions analogique-numérique-analogique lors de la connexion de deux inverseurs analogiques à la suite. Celle-ci risque de nuire à la précision. On a donc besoin de la connexion analogique-analogique. Pour résoudre ce problème il faut alors trois descriptions pour le même circuit :

- BIT à l'entrée, VOLT à la sortie,

```
entity inv1 is
    port     (input : in bit;
              output: linkage volt);
end inv1;
```

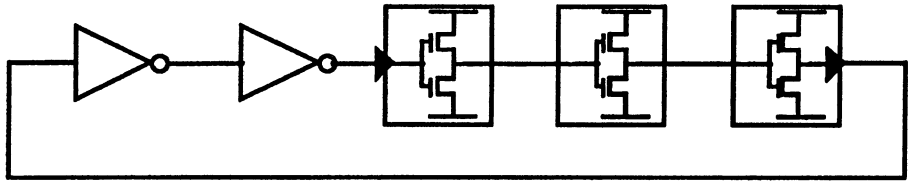
- VOLT à l'entrée, VOLT à la sortie

```
entity inv2 is
    port (input, output : linkage volt);
end inv2;
```

- VOLT à l'entrée, BIT à la sortie.

```
entity inv3 is
    port     (input : linkage volt;
              output: out bit);
end inv3;
```

On obtient donc une description structurelle illustrée par le schéma suivant :



Cette solution est déjà un peu plus complexe, et en conservant la description initiale de l'inverseur faite dans la première solution, avec pour ses connexions, le type BIT à l'entrée et à la sortie, on peut facilement déplacer les inverseurs analogiques et modifier ainsi la description. Malheureusement, il faut quatre descriptions d'un même inverseur pour avoir cette souplesse!

c. Troisième solution

Pour cette solution, nous allons conserver la description de l'inverseur avec des connexions analogiques en entrée et en sortie. Pour les interfaces, nous allons faire appel aux composants d'interface, dont voici les descriptions pour le type bit :

```
entity a2d_bit is
    generic (vth_fall, vth_rise : real := 2.0e-9;
                        load_cap : real := 1.0e-12);
    port     (input : linkage volt;
              output: out bit);

    -- définition des attributs sur le port output
    attribute a_vth_fall of output : signal is vth_fall;
    attribute a_vth_rise of output : signal is vth_rise;
    attribute a_load_cap of output : signal is load_cap;

end a2d_bit ;

architecture simple of a2d_bit is
    attribute analog_file_name …
begin
end simple;

entity d2a_bit is
    generic (…);
    port     (input : in bit;
         output: linkage volt);

    -- définition des attributs sur le port output
    attribute …
…
end d2a_bit ;

architecture simple of a2d_bit is
    attribute analog_file_name …
begin
end simple;
```

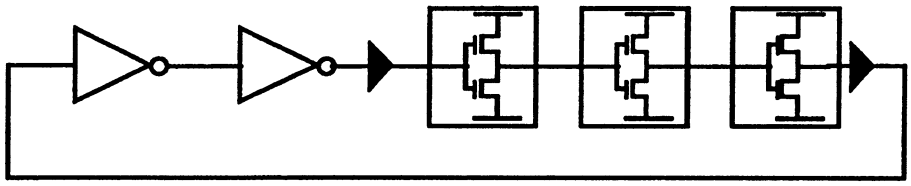
Au niveau du sous-circuit analogique, ces composants sont des résistances nulles entre leurs deux connexions :

```
.subckt d2a_bit a d     ! les ports n'ont pas forcément
                        ! le même nom
                        ! mais attention à l'ordre
    .connect a d  ! ce n'est qu'un fil …
.ends
```

Ces boîtes doivent être placées explicitement dans la description, soit par le concepteur soit automatiquement par le générateur de netlist.

Notre description devient alors beaucoup plus claire, avec certes deux composants inexistants en plus. Cependant, la modification du circuit se fait alors avec une grande facilité et on n'a plus à retoucher les sous-circuits analogiques, tout le travail se passe au niveau VHDL.

La description prend alors la forme suivante :



Et en voici la description VHD$_e$LDO correspondante :

```
entity anneau is
end anneau;

library vhdldo, vhdldo_mixed;
use vhdldo.analog.all, vhdldo_mixed.interfaces.all;

architecture structurelle of anneau is
    component inv_analogique
        port (input, output : linkage volt);
    end component;

    component inv_digital
        port (input : in bit;output : out bit);
    end component;

    signal d : bit_vector (1 to 3);
    signal a : volt_vector (1 to 4);

begin
    i1 : inv_digital port map (d(1), d(2));
    i2 : inv_digital port map (d(2), d(3));

        -- interface fictive numerique analogique
    d2a : d2a_bit port map (d => d(3), a => a(1))

    i3 : inv_analogique port map (a(1), a(2));
    i4 : inv_analogique port map (a(2), a(3));
    i5 : inv_analogique port map (a(3), a(4));

        -- interface fictive analogique numerique
    a2d : a2d_bit port map (a => a(4), d => d(1))
end structurelle;
```

d. Résumé

Comme on vient de le voir, notre approche de description permet de décrire un système mixte de plusieurs manières. Selon le niveau de complexité du système, l'utilisateur adoptera l'une ou l'autre. Pour notre part, il nous semble que pour des grosses descriptions la troisième semble la plus adaptée, elle permet une plus grande souplesse de description surtout au niveau de la hiérarchie, et permet une réutilisation facile des blocs analogiques sans avoir à les retoucher.

IV. Les choix techniques

1. La synchronisation

Nous avons déja vu les principaux algorithmes de synchronisation possible entre ces deux types de simulateurs. Pour un premier prototype, il nous a semblé que l'algorithme du pas-bloqué était le plus performant, ou du moins qu'il représentait le meilleur compromis étant donné les différents cas de figure. C'est celui-ci qui a été implanté dans le simulateur VHD$_e$LDO.

Bien entendu, pour une phase ultérieure il sera possible d'en intégrer d'autres et de laisser l'utilisateur choisir lequel il veut utiliser selon son cas de figure. En tout état de cause, le plus intéressant est l'algorithme du pas bloqué amélioré, et c'est vers celui-là que nous allons migrer.

Cependant, cet algorithme demande une analyse soignée de la partie numérique pour savoir si le prochain évènement va interférer avec la partie analogique.

2. L'élaboration

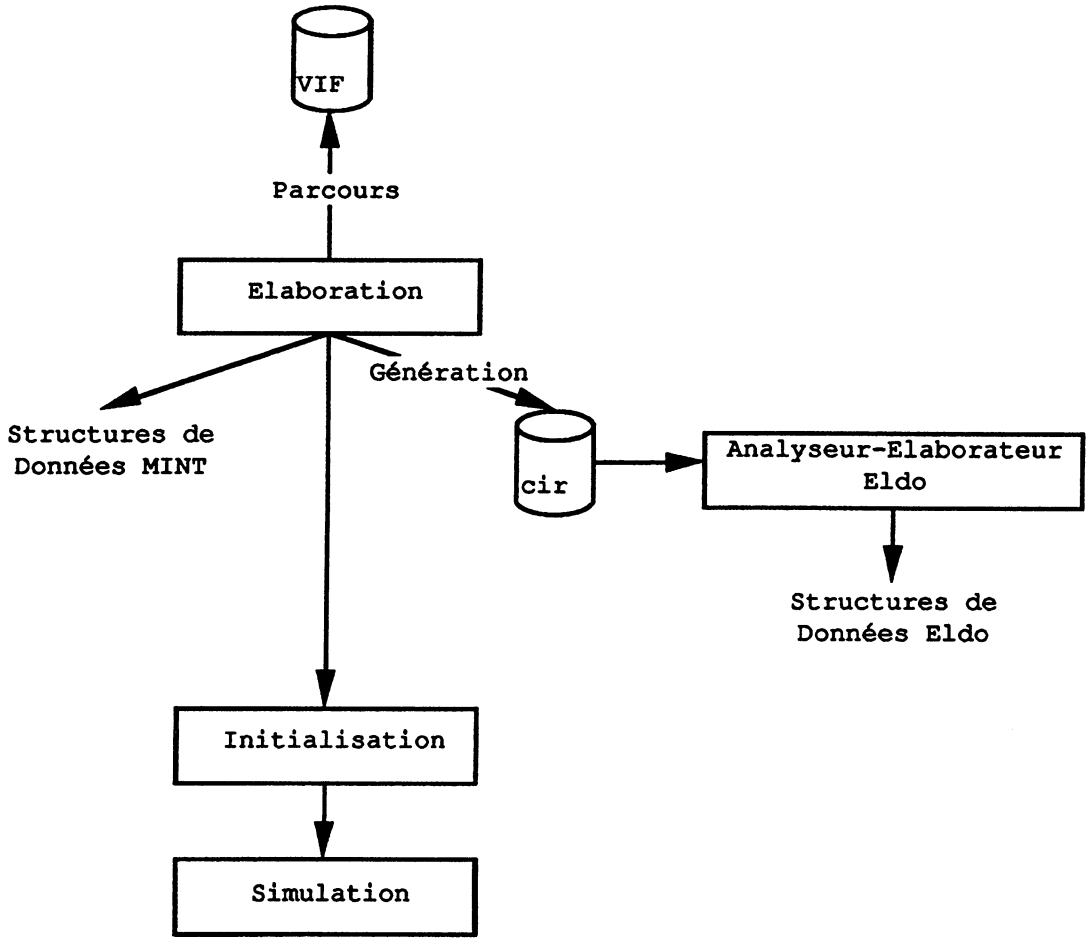
L'élaboration des parties purement numériques est conforme à ce qui est spécifié dans le manuel de référence. Par contre, pour les feuilles analogiques il a fallu ajouter de la sémantique pour générer une description Spice pour Eldo. Une solution ultérieure, plus rapide mais plus coûteuse à mettre en oeuvre, serait d'intégrer la sémantique attachée à la description Spice dans l'élaboration VHDL.

Comme nous l'avons constaté, il est possible d'utiliser le mode linkage dans une description structurelle avec certaines restrictions. Celles-ci sont vérifiées pendant la phase d'élaboration.

Lors de la génération de la description Spice, il faut reconnaître si les connexions de la partie analogique sont des noeuds mixtes ou bien s'il s'agit de noeuds analogiques.

Dans le premier cas, le nom généré est celui du port formel de l'entité, préfixé par le nom complet VHDL de l'instance de cette entité. Pour éviter les confusions entre la hiérarchie Eldo et VHDL, dans la netlist, les noms VHDL sont séparés avec des signes '/' et non des points. Enfin, il faut lire les valeurs des attributs attachés à ce noeud mixte pour définir l'interface et générer l'information

pour Eldo.



Synoptique de VHD$_e$LDO

Pour le second cas, les noeuds internes analogiques, il faut générer le nom du signal auquel le port formel est connecté. De la même manière, c'est le nom complet du signal qui est généré, c'est à dire qu'il possède les informations de la hiérarchie.

Un autre cas particulier est celui des vecteurs, en effet il est impossible d'utiliser de tels objets en Spice, alors que notre approche de description le permet. Donc il faut *éclater* le vecteur. Pour l'instant les vecteurs sont générés de 0 à n-1, on ne lit pas l'intervalle effectif VHDL. Pour les vecteurs à plusieurs dimensions, ils sont générés à une dimension, de 0 à n-1. La lecture du segment VHDL ne présente pas beaucoup de difficulté, par contre la gestion des tableaux à dimensions multiples nécessite un traitement un peu plus complexe.

3. L'initialisation

Nous avons choisi la solution la plus simple qui consiste à ne pas faire boucler les simulateurs. Cependant, dans une seconde phase du développement, la solution avec oscillation sera intégrée.

Néanmoins, il faut bien être conscient que la majorité des circuits ont besoin d'une phase d'établissement pour commencer à donner des résultats cohérents, et ceci autant en numérique qu'en analogique. Cela vient de la complexité des circuits; par exemple pour un filtre, plus son ordre est élevé, plus sa durée d'établissement est longue. Donc, partant de ces conclusions, on peut admettre que les problèmes de l'initialisation seront résolus en même temps.

Il faut quand même admettre que certains circuits ont besoin de bonnes valeurs d'initialisation sinon la simulation aura des résultats incohérents. C'est pour cette raison que la solution avec itérations sera intégrée.

Dans tous les cas, l'utilisateur peut intervenir et fixer les valeurs de départ, aussi bien du côté analogique que numérique.

4. La simulation

Le cycle de simulation de VHDL a été modifié légèrement pour pouvoir intégrer l'appel à Eldo et donc l'algorithme de synchronisation. Ainsi, avant de changer de date, VHDL doit lancer Eldo jusqu'à son prochain événement. De plus, il se peut que la partie analogique génére un événement que VHDL devra intégrer dans son calendrier. Dans ce cas, la date de prochain événement de VHDL ne sera pas celle prévue.

La gestion de l'appel d'Eldo est identique à un processus différé (**post-poned**) de VHDL'92. Ce processus est cependant différent, car il doit être appelé à chaque pas de calcul de VDHL.

V. Exemples

1. Une ligne RC

a. Description

Au travers de cet exemple simple, nous allons mettre en valeur les possibilités de descriptions structurelles qu'offre VHDL. Le bloc que nous allons décrire est une modélisation d'une ligne de transmission. Il s'agit d'une série de cellules RC mises bout à bout. L'intérêt principal de VHDL est de proposer un composant dont le nombre de cellules est générique. Il s'agit donc d'un exemple purement analogique.

D'autre part, nous utilisons dans cette description la bibliothèque 'vhdldo_analog' qui contient les entités-architecture des composants de base d'Eldo, ainsi que le paquetage 'devices' de composants VHDL qui leur est associé.

```
library vhdldo;
use vhdldo.analog.all;

entity cellule_rc is
    generic (r, c : real);
    port (p, n, ref : linkage volt);
endcellule_rc;


library vhdldo_analog;
use vhdldo_analog.all, vhdldo_analog.devices.all;

architecture structurelle of cellule_rc is

begin

    res : resistor
        generic map (r)
        port map (p, n);

    cap : capacitor
        generic map (c)
        port map (ref, n);

end structurelle;


entity ligne is
    generic (nb_cell : positive := 10);
    port (p, n, ref : linkage volt);
end ligne;

library vhdldo;
use vhdldo.analog.all;
```

```
architecture structurelle of ligne is

    component cellule_rc
        generic (r, c : real);
        port (p, n, ref : linkage volt);
    end component;

    signal a : volt_vector (1 to nb_cell-1);

begin

    c1 : cellule
        generic map (r, c)
        port map (p, a(1), ref);

    l : for i in 1 to nb_cell-2 generate
        ci : cellule
            generic map (r, c)
            port map (a(i), a(i+1), ref);
    end generate l;

    cn : cellule
        generic map (r, c)
        port map (a(nb_cell-1), n, ref);

end structurelle;
```

Le langage Spice n'est pas générique, il est donc impossible de décrire un tel composant. D'autre part, même pour un n fixé, la description est vite très lourde, alors que l'instruction de génération VHDL permet de la simplifier.

b. Simulation

Voici la description de plus haut niveau qui permet de simuler la ligne RC. Le noeud de référence est défini globalement dans le paquetage 'analog', et il s'appelle 'a_gnd'. Ce noeud est reconnu à l'élaboration pour la génération de la netlist Spice.

Enfin, dans la description structurelle, nous avons branché une source *muette* sur laquelle on peut configurer celle que l'on veut dans l'unité de configuration.

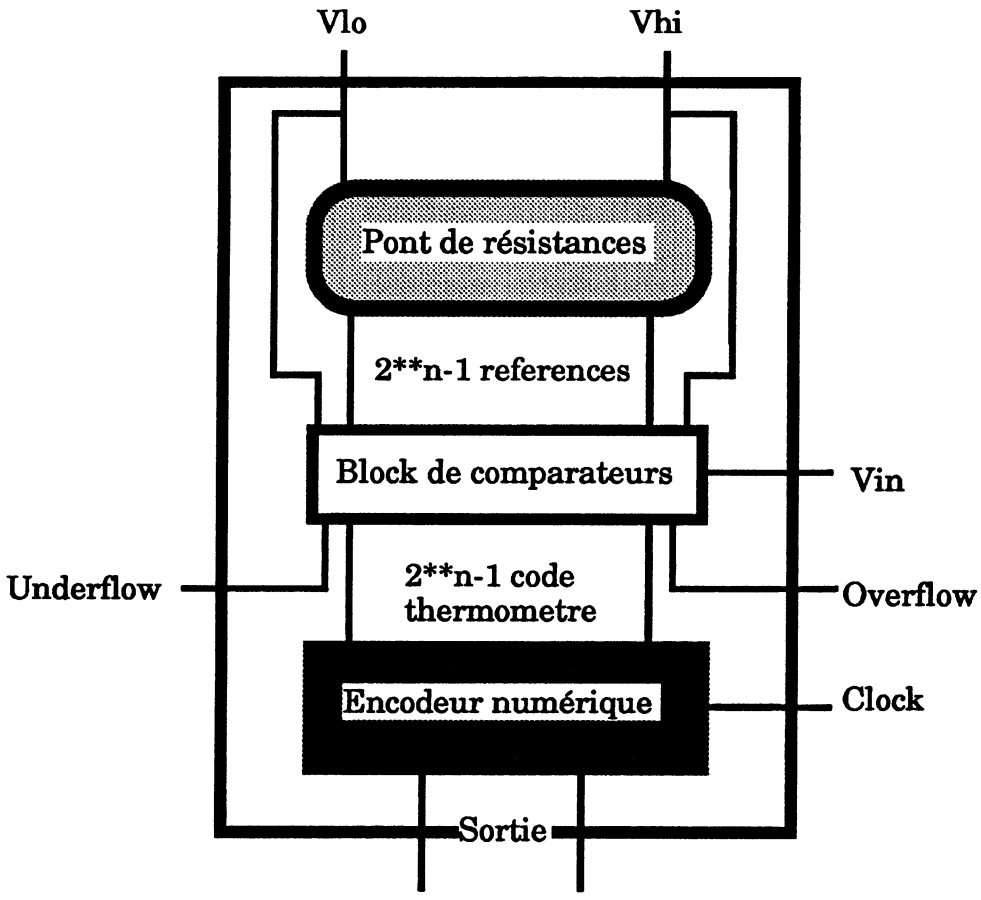
```
entity test is
end test;

library vhdldo;
use vhdldo.analog.all;
library vhdldo_analog;
use vhdldo_analog.devices.all;

architecture structurelle of test is
```

```
    component ligne
        port (p, n, ref : linkage volt);
    end  component;

    signal a, b : volt;

begin
    l : ligne port map (a, b, a_gnd);

    source : v_source port map (a, a_gnd);
end test;

library vhdldo_analog;
use vhdldo_analog.devices.all;
use vhdldo_analog.all;
configuration a of test is
    for structurelle
        for l : ligne use entity ligne(structurelle)
                generic map (n => 10);
                for structurelle
                    for all : cell
                              use entity cell(structurelle)
                              generic map(r => 10.0,
                                          c => 1.0e15);
                              end  for;
                end  for;
        end  for;
        for source : v_source use entity v_pulse
                generic  map   (td => 100.0e-9,
                                tr => 1.0e-9,
                                tf => 1.0e-9,
                                pw => 1.0e-6,
                                per => 2.0e-6);
        end  for;
    end  for;
end a;
```

2. Un convertisseur analogique-numerique n bits flash

a. Présentation et architecture

Les convertisseurs flash sont des convertisseurs rapides destinés à des applications qui nécessitent des forts débits : l'exemple typique étant la transmission d'images vidéo. Le schéma suivant nous montre l'architecture interne d'un convertisseur flash :



Architecture d'un convertisseur flash

Le fonctionnement du circuit est très simple. Le pont de résistances fabrique les références auxquelles va être comparé le signal d'entrée. Cette comparaison a lieu dans le bloc de comparateurs. La sortie de ce bloc est un code thermomètre, appelé ainsi car la valeur codée est représentée par un nombre de zéros dans le vecteur de sortie, cela rappelle la hauteur de mercure dans un thermomètre. Par exemple, la valeur trois sera codée 000111...11. Ensuite la partie encodeur va transformer ce code en nombre en binaire.

Nous allons donc modéliser ce composant grâce à notre approche de description. L'intérêt principal est de pouvoir décrire un composant de manière générique par rapport à son nombre de connexions.

b. La description

La description du convertisseur va être une description structurelle. Dans un premier temps nous

allons décrire chacun des trois blocs. Ensuite, nous réaliserons la description finale, et nous la testerons dans un petit exemple.

i. Le pont de résistances

L'entité du pont de résistance comporte deux paramètres génériques : le nombre de bits du convertisseur et la valeur de la résistance de base. Les connexions sont au nombre de trois : deux connexions pour référencer les bornes de travail du convertisseur et une troisième qui est le vecteur de références. La taille de ce vecteur dépend du paramètre générique n.

```
library vhdldo;
use vhdldo.analog.all; -- accès au types volt et volt_vector

entity pont_res is
    generic (n : positive;
                    r : real); -- valeur de la résistance de base
    port     (Vhi, Vlo : linkage volt;
              Ref : linkage volt_vector (1 to 2**n-1));
end pont_res;
```

Le pont de résistances est une ligne de résistances qui relie chacune des références en partant de la référence haute Vhi, et en arrivant à la référence basse Vlo. D'autre part, nous utilisons la bibliothèque `vhdldo_analog` dans laquelle se trouvent les entités représentant les composants de base d'Eldo. A chaque entité de cette bibliothèque il existe un composant dans le paquetage `devices`, paquetage sur lequel on demande la visibilité.

```
library vhdldo_analog;
use vhdldo_analog.devices.resistor;

-- visibilité sur l'entité rèsistor
-- pour la configuration par défaut
use vhdldo_analog.resistor;

architecture structural of pont_res is

begin

    first : resistor
        generic map (r)
        port map (Vhi, ref(2**n-1));

    general : for i in 2**n-1 downto 1 generate
        res : resistor
                generic map (r)
                port map (ref(i), ref(i-1));
    end generate general;

    last : resistor
        generic map (r)
        port map (Vlo, ref(1));
```

```
end structural;
```

ii. Le bloc de comparateur

Ce composant est un composant mixte puisque sa sortie est un vecteur de bits.

```
library vhdldo;
use vhdldo.analog.all; -- accès au types volt et volt_vector

entity comp_bloc is
    generic (n : positive);
    port     (Vhi, Vlo, Vin : linkage volt;
              Ref : linkage volt_vector (1 to 2**n-1);
              q : out bit_vector (1 to 2**n-1);
              underflow, overflow : out bit);
end comp_bloc;
```

La structure interne de ce composant est identique à celle du pont de résistances, il s'agit d'une chaîne de comparateurs connectés entre une référence, l'entrée et une sortie.

```
library vhdldo_analog;
use vhdldo_analog.devices.comparator;

library vhdldo_mixed;
use vhdldo_analog.interface.a2d_bit_vector;
use vhdldo_analog.a2d_bit_vector;

architecture structural of comp_bloc is

    signal q_analog : volt_vector (1 to 2**n-1);
    signal not_overflow;

    for all : comparator use entity
        vhdldo_analog.comparator
        generic map(Vhi => 5.0, Vlo => 0.0, tpd => 2.0e-9)
begin

    first : comparator
        port map (Vhi, Vin, not_overflow);

    general : for i in 2**n-1 downto 1 generate
        comp : comparator
                port map (ref(i), Vin, q_analog(i));
    end generate general;

    last : comparator
        port map (Vlo, Vin, underflow);

    overflow <= not not_overflow after 2 ns;

    -- interface explicite entre q_analog et q
    int : a2d_bit_vector
        generic map (vth_rise => 2.7,
                                 vth_fall => 2.3,
                                 load_cap => 1.0e-12)
```

```
        port map (a => q_analog, d => q);

end structural;
```

iii. L'encodeur numérique

Ce composant est un composant numérique très classique. Tout d'abord, on va définir un paquetage dans lequel on va écrire une fonction de conversion entre un entier naturel et un vecteur de bits.

```
package fonction_locale is
    function natural2bitvect (size, nb : natural)
        return bit_vector;
end  package;

package body fonction_locale is
    function natural2bitvect (size, nb : natural)
        return bit_vector is
        variable res : bit_vector (1 to size);
        variable int : natural := nb;
    begin
        for i in 1 to size loop
                if (int mod 2) = 1 then
                                res(i) := '1';
                end if;
                int := int / 2;
        end loop;
        return res;
    end natural2bitvect;
end  package body;
```

Maintenant, voici la description de l'encodeur :

```
entity encodeur is
    generic (n : positive; front : bit);
    port     (q : in bit_vector (1 to 2**n-1);
              clock : in bit;
              sortie : out bit_vector (1 to n));
end comp_bloc;

use work.fonction_locale.natural2bitvect;
architecture comportementale of encodeur is
begin
    process
        variable compteur : natural;
    begin
        wait until clock=front and not clock'stable;
        compteur := 0;
        loop
                exit when q(compteur+1)='1'
                compteur := compteur+1;
        end loop;
        sortie <= natural2bitvect(n, compteur);
    end  process;
end comportementale;
```

iv. Le convertisseur

Pour la description structurelle, nous supposerons que les descriptions des composants ont été faites dans le paquetages local_components.

```
entity flash_adc is
    generic (n : positive; r : real; front : bit);
    port     (Vhi, Vlo, Vin : linkage volt;
              clock : in bit;
              underflow, overflow : out bit;
              sortie : out bit_vector(1 to n));
end flash_adc;


library vhdldo;
use vhdldo_analog.all;
use work.local_components.all;

architecture structurelle of flash_adc is
    signal Ref : volt_vector (1 to 2**n-1);
    signal q : bit_vector (1 to 2**n-1);
begin
    pr : pont_res
         generic map (n, r)
         port map (Vhi, Vlo, Ref);
    bc : comp_bloc
         generic map (n)
         port map     (Vhi, Vlo, Vin, Ref, q
                                  underflow, overflow)
    en : encodeur
         generic map (n, front)
         port map (q, clock, sortie);
end structurelle;
```

3. Un convertisseur Analogique Numérique semi-flash

a. Présentation du problème

Les convertisseurs semi-flash sont des convertisseurs analogique-numérique dérivés des convertisseurs flash. Dans l'exemple précédent, nous avons décrit un convertisseur flash. L'inconvénient majeur de cette architecture est le grand nombre de comparateurs et donc sa forte consommation.

Les convertisseurs semi-flash permettent de diminuer le nombre de convertisseurs mais perdent en rapidité : on passe de un échantillon par coup d'horloge, à un tous les deux coups d'horloge. Le principe de ces convertisseurs est le suivant : on divise le nombre de bits en deux parties égales ou non, les Bits de Poids Fort (Most Significant Bits : MSB) et les Bits de Poids Faible (Less Significant

Bits : LSB). Le codage est effectué en deux temps. Une première comparaison détermine les MSB, et commande le pont de résistances pour positionner les références de la comparaison des LSB. Ensuite, une seconde comparaison détermine les LSB. Il faut donc un pont de résistances avec des références MSB fixes, et des références LSB variables, deux blocs de comparateurs et deux encodeurs.

L'architecture de l'exemple est une architecture à recouvrement (overlapped) [Abr93]. Celle-ci permet de sortir un échantillon par coup d'horloge. Cependant les échantillons sortent avec un délai de deux coups d'horloge.

Le pont de résistances est identique à celui d'un convertisseurs flash. Cependant, chaque noeud de référence est connecté à sa référence LSB potentiel via un interrupteur commandé par la rétro-action de l'encodeur MSB. De plus pour des raisons de géométrie dûes au gain de place sur le silicium, les segments LSB sont une fois dans un sens une fois dans l'autre. En fait le pont de résistances est replié. C'est pourquoi, la parité des MSB est une entrée de l'encodeur LSB pour coder correctement les LSB. En effet, dans un sens il faut compter les zéros, et dans l'autre il faut compter les uns.

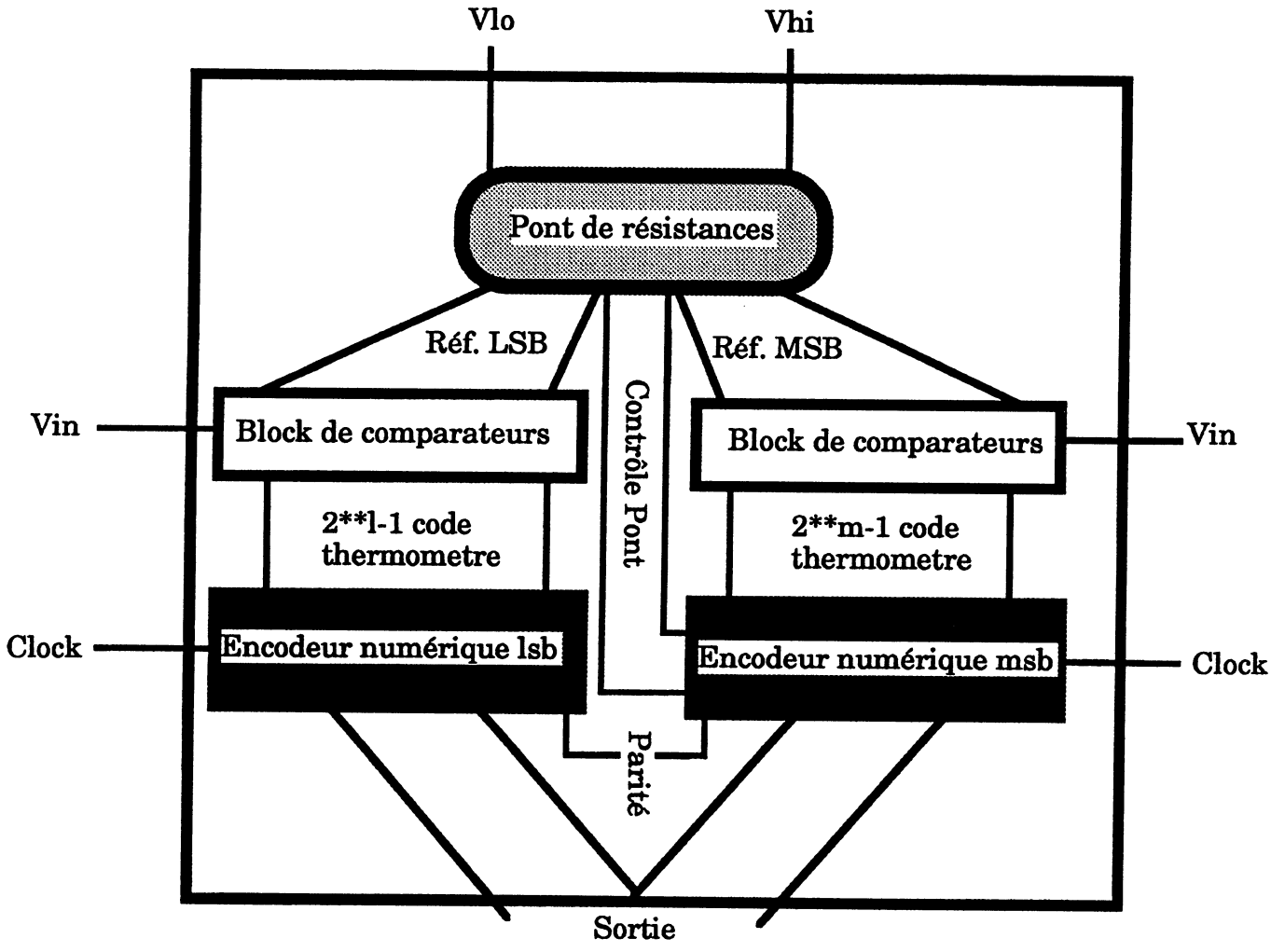


Schéma d'un convertisseur Semi-Flash

L'architecture utilisée pour cet exemple est spéciale, et permet de conserver la rapidité du convertisseur flash : il s'agit, en effet, d'un convertisseur semi-flash à un coup d'horloge.

Le circuit est en cours de réactualisation, on l'adapte pour une nouvelle technologie. La partie numérique ne souffre pas de ce changement de technologie, par contre la partie analogique doit être soigneusement recalculée pour répondre au nouveau critère de rapidité. Il faut donc refaire toute une série de simulations pour valider la réalisation. Malheureusement, le circuit est assez important : un millier de MOS pour la partie analogique et trois mille pour la partie numérique, et donc la simulation au niveau électrique est très lente. C'est pour cette raison que le passage à la simulation mixte est intéressant.

D'autre part, la partie numérique est maintenant décrite en VHDL et pourra ultérieurement être

synthétisée en utilisant un outil de synthèse. La seule contrainte sera de respecter le sous-ensemble synthétisable. Cet aspect n'est pas négligeable quand on sait combien est fastidieuse la conception d'une partie de traitement numérique.

b. Le passage à la simulation mixte

Le fichier de description de départ est un fichier Spice généré par l'extracteur de netlist à partir du plan de masse. Cette netlist est augmentée des capacités parasites, ce qui rend la simulation électrique plus précise puisqu'elle tient compte de l'implantation physique.

La netlist générée est une netlist sans hiérarchie. La première étape a donc été de reprendre cette netlist et de reformer des sous-circuits pour traduire la hiérarchie du circuit. Cette étape était de toute façon nécessaire pour pouvoir utiliser un outil de routage automatique.

Ensuite, à chaque sous-circuit que l'on voulait simuler au niveau électrique, on a associé un couple entité-architecture. De plus, pour tous les blocs numériques, il a fallu écrire une architecture comportementale : cela est allé de la simple bascule jusqu'à un encodeur qui traduit un code thermomètre en binaire. Le type des signaux a été le type standard IEEE à neuf états. Ce choix ne s'est pas fait parce que le circuit possède des cas de signaux multi-sources, mais il s'inscrit dans l'optique de l'utilisation de l'outil de synthèse de Synopsis.

L'étape suivante a été d'écrire la description structurelle du convertisseur dans son entier. Enfin, on a écrit une entité de test. La description du convertisseur représente mille cinq cents lignes de code VHDL.

Une dernière étape consisterait à réécrire chaque composant analogique (pont résistif, bloc de comparateurs) en utilisant les possibilités de VHD$_e$LDO : paramètres génériques et description structurelle utilisant des instructions de générations (voir Annexe B). En effet un tel circuit peut être réutilisé dans un montage plus complexe mais avec des constantes différentes : nombre de bits total, partition LSB-MSB, nombre de bits de correction d'offset ou encore passage à une architecture pipe-line. Grâce à VHDL, ces changements peuvent être rapidement pris en compte. Avec le langage Spice il faudra jouer du copier-coller pour arriver au même résultat.

c. Les résultats

Le passage à la simulation mixte nous a permis de gagner un facteur cinq par rapport à la simulation électrique pure et un facteur trois par rapport à la simulation électrique en mode mixte. Cela est conforme avec la règle qui est le rapport du nombre de transistors du circuit sur le nombre de transistors simulés au niveau analogique.

Eldo	Eldo mixte	VHD_eLDO
35 heures	23 heures	7 heures

Enfin, nous avons effectué la simulation du bloc numérique seul en générant des stimuli sur les signaux mixtes analogique-numériques. Ces stimuli étaient équivalents au comportement en simulation mixte. Le résultat est très clair, la simulation ne consomme que quelques dizaines de secondes de temps CPU. Cela veut dire que, quand bien même le simulateur VHDL serait optimisé pour tourner plus vite, le gain sur cette simulation serait négligeable face à la durée de simulation de la partie analogique.

Néanmoins, le facteur de gain permet de lancer des simulations de près d'une vingtaine d'heures, chose qui aurait été impossible avec la simulation analogique. En effet, elles auraient alors tourné pendant plus de trois jours.

D'autre part, la partie numérique est facilement modifiable, et ainsi on peut valider différents algorithmes pour gérer la correction de quelques erreurs possibles dues aux limitations du comportement analogique. Ces modifications pourront être rapidement prises en compte dans le circuit final grâce à la synthèse.

Avec le gestionnaire de bibliothèque de VHDL, on peut gérer différente descriptions de simulation avec les unités de configuration. Par exemple, on peut associer plusieurs architectures à une entité représentant un sous-circuit analogique. Chacune de celles-ci référençant un fichier différent et donc une structure différente du sous-circuit. Ensuite, il ne reste plus qu'a utiliser des unités de configuration pour simuler telle ou telle solution.

On entrevoit donc, à travers cet exemple concret, les possibilités qui sont offertes au concepteur

grâce à l'utilisation de langages de description de haut niveau. Cela lui donne accès à des outils de synthèse automatique pour la partie numérique, mais aussi cela lui permet de changer facilement sa description en passant d'une description structurelle à un macro-modèle ou encore à une description comportementale.

L'intérêt de posséder un langage unique pour la description analogique numérique nous apparaît comme important. Ici, nous avons dû utiliser le langage FAS qui n'a rien de commun avec VHDL, cela est fastidieux autant au niveau de la liaison qu'au niveau des connaissances à posséder.

VI. Bilan

Les grands enseignements sur cette réalisation sont venus du dernier exemple où le programme a réellement été confronté à des problèmes concrets.

En premier lieu, nous avons pu constater l'intérêt d'un tel outil pour les concepteurs. Cet intérêt porte d'une part sur le gain de temps pour la simulation, mais aussi sur la possibilité d'utiliser VHDL et tous les outils utilisables avec ce langage, pour les partie numériques de la réalisation. Enfin, on peut noter la possibilité de décrire des structures analogiques avec VHD$_e$LDO au moyen des ports de mode linkage, et ainsi de profiter de la puissance VHDL quant à ce type de description.

Il est à noter que ce travail a été mené en collaboration avec le CNET et Anacad, que le simulateur mixte VHD$_e$LDO est appelé à être un produit de cette société. L'intérêt de posséder un tel simulateur, et surtout les connaissances au niveau du langage est énorme vue l'importance sans cesse croissante de VHDL.

A travers cette réalisation, nous avons pu valider l'approche structurelle de VHDL-Analogique. En effet, VHD$_e$LDO offre la possibilité de décrire un composant de manière structurelle et avec des paramètres génériques (cf. le convertisseur flash). Il offre donc aux concepteur toute la puissance de VHDL pour le domaine structurel : gestion de bibliothèques, instructions de génération, composant non dédié à une entité et encore l'unité de configuration. Il est également possible d'utiliser le langage de description comportementale FAS. Bientôt, une évolution de celui-ci devrait offrir une syntaxe proche de VHDL, il se nommera HDL-A. On aura alors accès à un langage proche de

Conclusion

Nous avons présenté notre travail, la participation à la normalisation de VHDL-analogique et la réalisation du simulateur mixte VHD$_e$LDO.

En ce qui concerne la normalisation, elle en est actuellement dans ses phases terminales : la phase de conception du langage est bien commencée et bon nombre de problèmes sont résolus. Notre travail s'est intégré dans la mise en place des nouveaux concepts. Le point important a été de conserver la philosophie VHDL. Ainsi, nous n'avons pas opté pour le rajout d'un nouveau constructeur pour les systèmes analogiques, mais pour une extension des structures existantes. La différence entre un système analogique et un système numérique se fait au niveau de ses interfaces ou de son contenu. D'autre part, grâce à l'expérience du langage FAS, nous avons pu éviter certains écueils et rapidement introduire les notions importantes.

Pour le simulateur VHDeLDO, il est actuellement à l'état de prototype. Il commence à être utilisé au CNET Grenoble. Comme nous l'avons vu dans le dernier chapitre, notre approche de description permet de bénéficier de la puissance de VHDL en ce qui concerne la souplesse de modélisation structurelle et de gestion de bibliothèque. D'autre part, elle pourrait s'appliquer facilement pour coupler n'importe quel simulateur avec VHD$_e$LDO.

L'étape suivante sera l'intégration complète du langage HDL-A. Tout d'abord cette intégration se fera simplement, pour migrer dans un deuxième temps vers une intégration plus forte dans laquelle on pourra définir des interactions mixtes.

Enfin, c'est là que les deux travaux se rejoignent, VHD$_e$LDO sera appelé à être un prototype de VHDL-analogique, et cela sera la phase terminale de la validation du langage.

Annexe A

I. Ligne RC

1. Fichier Spice généré

Voici le résultat de simulation de l'exemple du chapitre quatre : la ligne de cellules RC. Tout d'abord la netlist complète générée par VHD$_e$LDO.

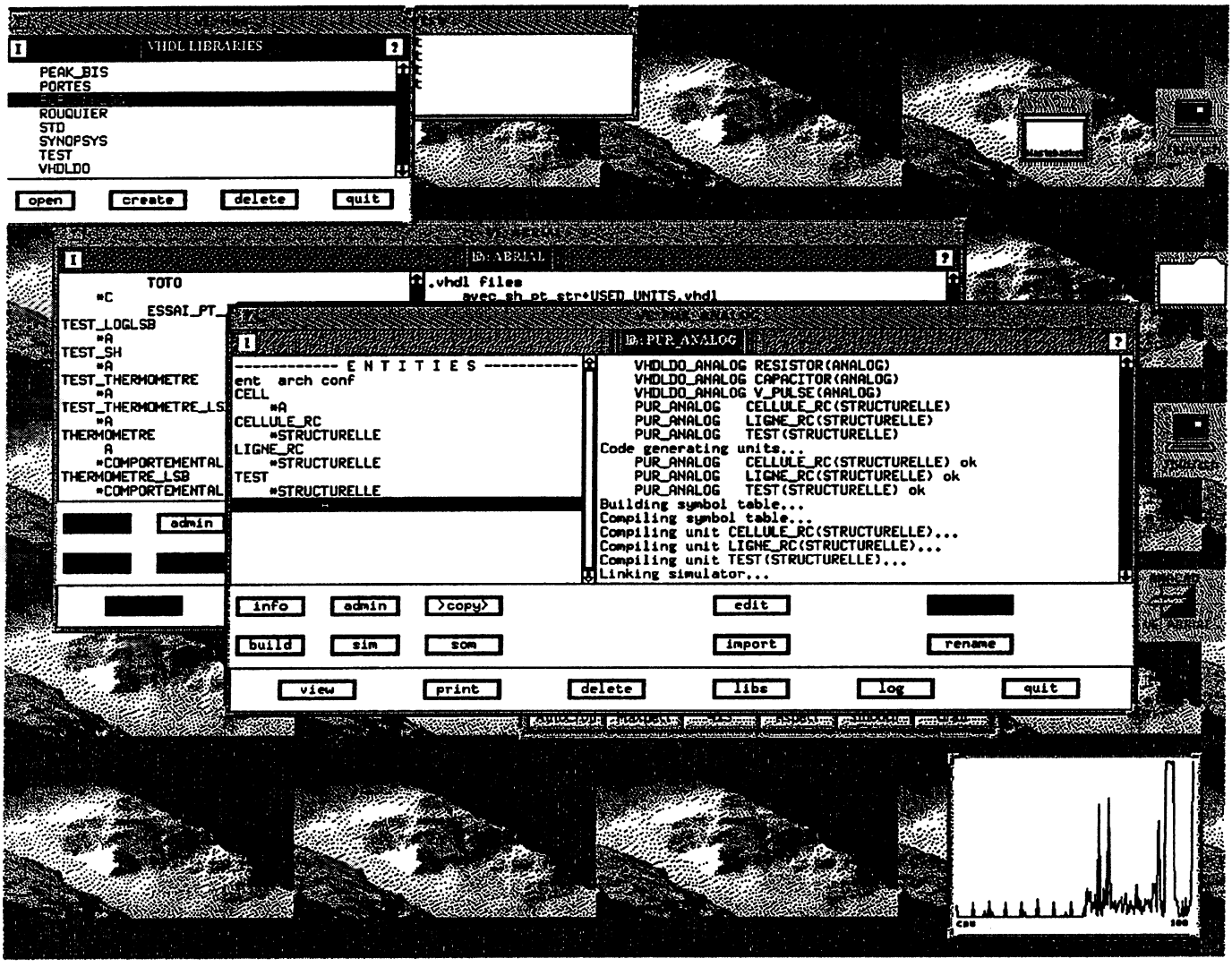
```
* début d'insertion du fichier a.cmd
.tran 10n 3u



.options eps=1e-6


.plot tran v(a) v(l/a[5]) v(b)


* début de la netlist générée


** INSTANCE
XSOURCE    ! NAME
+ A     ! connection 1
+ 0     ! connection 2
+ V_PULSE      ! Name of the Subckt
+ V0 = 0.000000e+00
+ V1 = 5.000000e+00
+ TD = 1.000000e-07
+ TR = 1.000000e-09
+ TF = 1.000000e-09
+ PW = 1.000000e-06
+ PER = 2.000000e-06
** LIBRARY
.LIB $UNIVERSE/VHDLDO_ANALOG/devices.ckt



** INSTANCE
XL/C1/RES    ! NAME
+ A     ! connection 1
+ L/A[0]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02
```

```
** INSTANCE
XL/C1/CAP    ! NAME
+ L/A[0]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[2]/CI/RES    ! NAME
+ L/A[0]     ! connection 1
+ L/A[1]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[2]/CI/CAP    ! NAME
+ L/A[1]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[3]/CI/RES    ! NAME
+ L/A[1]     ! connection 1
+ L/A[2]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[3]/CI/CAP    ! NAME
+ L/A[2]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[4]/CI/RES    ! NAME
+ L/A[2]     ! connection 1
+ L/A[3]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[4]/CI/CAP    ! NAME
+ L/A[3]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[5]/CI/RES    ! NAME
+ L/A[3]     ! connection 1
+ L/A[4]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[5]/CI/CAP    ! NAME
+ L/A[4]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
```

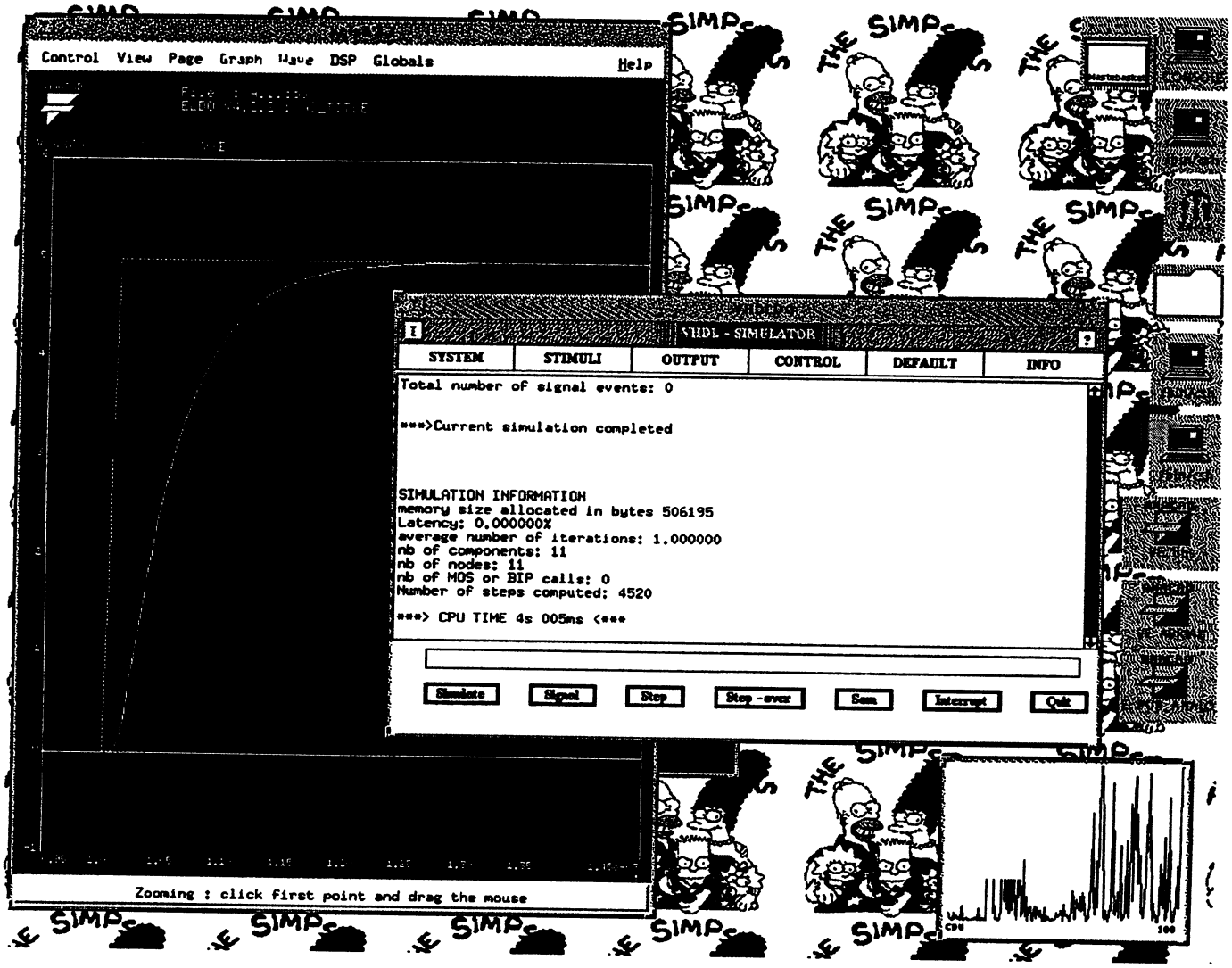
```
+ CAP = 1.000000e-12

** INSTANCE
XL/L[6]/CI/RES    ! NAME
+ L/A[4]     ! connection 1
+ L/A[5]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[6]/CI/CAP    ! NAME
+ L/A[5]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[7]/CI/RES    ! NAME
+ L/A[5]     ! connection 1
+ L/A[6]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[7]/CI/CAP    ! NAME
+ L/A[6]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[8]/CI/RES    ! NAME
+ L/A[6]     ! connection 1
+ L/A[7]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[8]/CI/CAP    ! NAME
+ L/A[7]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/L[9]/CI/RES    ! NAME
+ L/A[7]     ! connection 1
+ L/A[8]     ! connection 2
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/L[9]/CI/CAP    ! NAME
+ L/A[8]     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12

** INSTANCE
XL/CN/RES    ! NAME
+ L/A[8]     ! connection 1
+ B     ! connection 2
```

```
+ RESISTOR      ! Name of the Subckt
+ RES = 1.000000e+02

** INSTANCE
XL/CN/CAP    ! NAME
+ B     ! connection 1
+ 0     ! connection 2
+ CAPACITOR      ! Name of the Subckt
+ CAP = 1.000000e-12
.END
```

2. Compilation

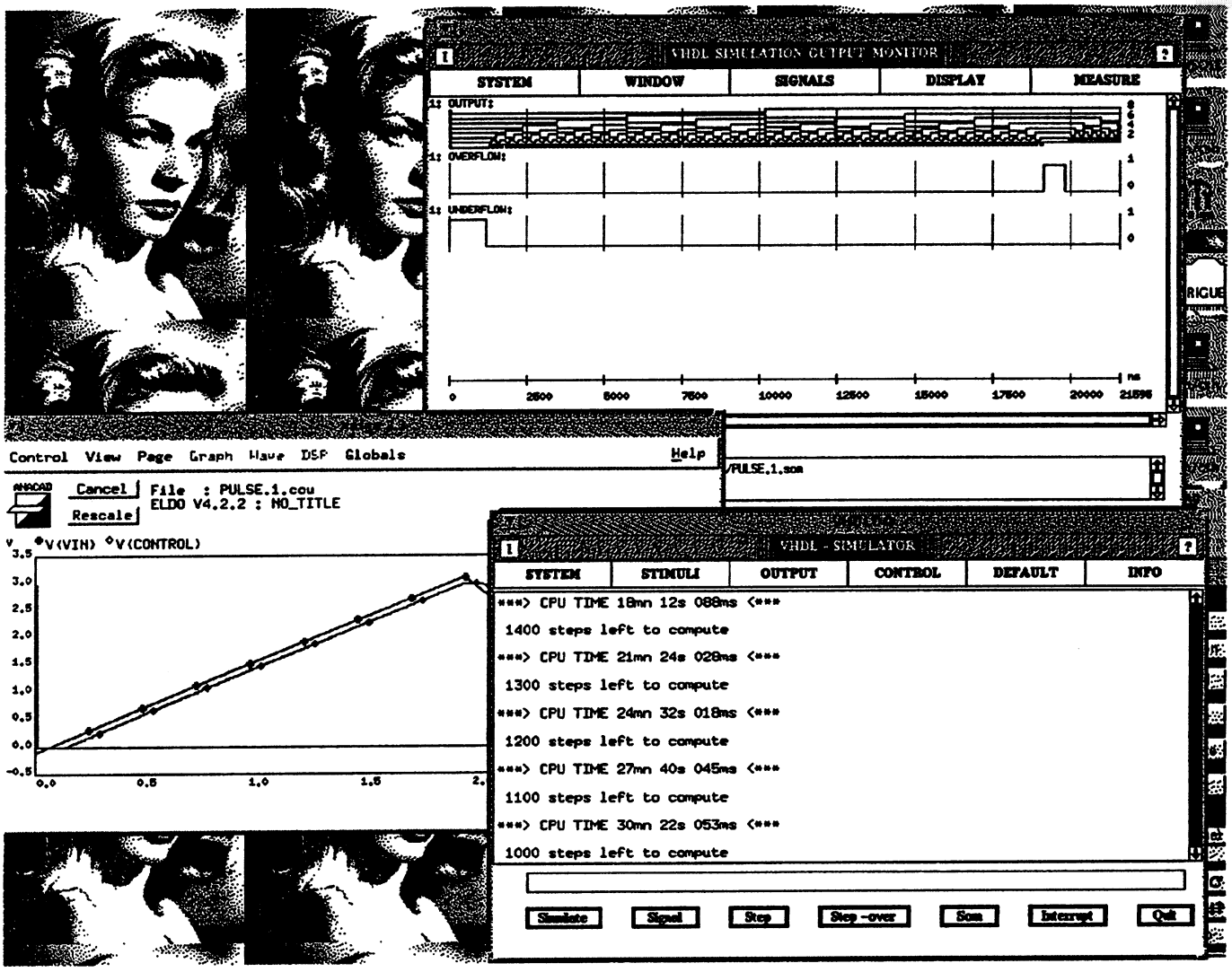


3. Simulation et résultats



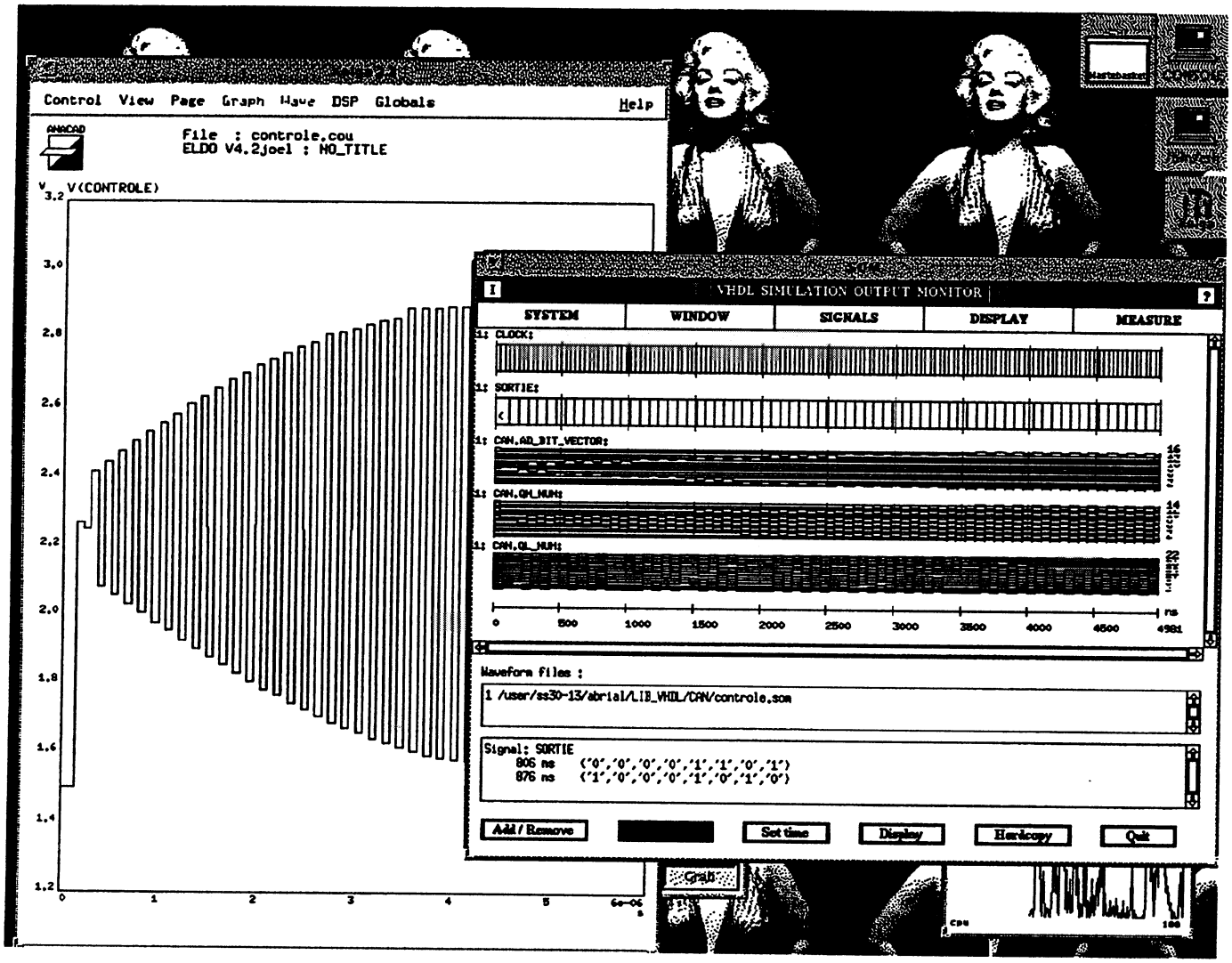
Cette copie d'écran nous montre la fenêtre de simulation à droite et les courbes de résultats à gauche. Sur le graphique on voit bien le retard imposé par la ligne de cellules RC. La courbe la plus à gauche représente le stimulus d'entrée, la courbe intermédiaire la tension aux bornes de la cinquième cellule et la courbe de droite la sortie de la ligne à retard.

II. Le convertisseur Flash



La simulation est encore en cours. A gauche, nous voyons les courbes analogiques : le stimulus d'entrée et la sortie du convertisseur recodé en analogique. En haut à droite se trouvent les chronogrammes numériques : le vecteur de sortie du convertisseur et les contrôles de débordements positif et négatif.

III. Le convertisseur Semi-Flash



La simulation consiste en un test en fréquence du convertisseur : un battement. Il s'agit de mettre comme stimulus une sinusoïde de fréquence la fréquence d'horloge moins un epsilon. Le résultat est alors un battement qui fait apparaitre une basse fréquence de fréquence epsilon. A gauche on voit la sortie analogique, et à droite les chronogrammes numériques. On y trouve l'horloge, la sortie, le vecteur de contrôle du pont de résistance ainsi que les codes thermomètres LSB et MSB.

Annexe B

I. Description générique du pont de résistance

Cette entité-architeture sert dans la description du convertisseur semi-flash présenté dans le quatrième chapitre.

```
library vhdldo;
use vhdldo.analog.all;

entity pont_resistif is
    generic (nb_msb, nb_lsb : positive;
             offset : natural;
             r : real := 8.64309e-1);
    port (ad : linkage volt_vector (1 to 2**nb_msb);
          refl : linkage volt_vector (1 to 2**lsb + 2*offset - 1);
    constant deuxp_nb_msb : positive := 2**nb_msb;
    constant deuxp_nb_lsb : positive := 2**nb_lsb;
end pont_resistif;

library vhdldo_analog;
use work.all, work.local_component.all;
use vhdldo_analog.all, vhdldo_analog.all;

architrecture structural of pont_resistif is

    type volt_matrix is array (natural range <>, positive range <>)
        of volt;

    -- deuxp_nb_msb segments composes de deuxp_nb_lsb resistances
    -- => deux-_nb_lsb-1 noeuds par segment
    signal noeud : volt_matrix   (0 to deuxp_nb_msb-1,
                                  1 to deuxp_nb_lsb-1);

begin

    -- createur du diviseur de tension,
    -- deuxp_nb_msb segments de deuxp_nb_lsb resistances,
    -- de vp, refm(1), ..., refm(deuxpn_nb_lsb-1), vm

    -- segment de depart entre vp et refm(1),
    -- et vm et refm(deuxp_nb_msb-1)
    r1_vp : resistor generic map(r) (vp, noeud(0, 1));
    r1_vm : resistor
        generic map(r)
        port map(refm(deuxp_nb_msb-1), noeud(deuxp_nb_msb-1, 1));

    segment : for j in 2 to deuxp_nb_lsb-1 generate
        ri_vp : resistor
            generic map(r)
            port map(noeud(0, j-1), noeud(0, j));
        ri_vm : resistor
            generic map(r)
```

```
          port map(noeud(deuxp_nb_msb-1, j-1),
                   noeud(deuxp_nb_msb-1, j));


rn_vp : resistor
     generic map (r)
     port map (refm(1), noeud(0, deuxp_nb_lsb-1));
rn_vm : resistor
     generic map (r)
     port map (vm, noeud(deuxp_nb_lsb-1, deuxp_nb_lsb-1));

segment : for i in 2 to deuxp_nb_msb-2 generate
     r1 : resistor
          generic map(r)
          port map(refm(i), noeud(i, 1));

     ri : for j in 2 to deuxp_nb_lsb-1 generate
          ri : resistor
               generic map(r)
               port map(noeud(i, j-1), noeud(i, j));
     end generate ri;

     rn : resistor
          generic map(r)
          port map(refm(i+1), noeud(i, deuxp_nb_lsb-1));
end generate res;

-----------------------------------------------------------------
--
-- on attache les deuxp_nb_lsb-2 entrees de chaque segment
-- aux refl via les interrupteurs commandes par le vecteur ad,
-- chaque noeud est connecte a deuxp_nb_msb interrupteurs
--
-- les noeuds(n, -) sont connectes a refl(n+1) dans le cas
-- general
--
-- les segments pairs, entre refm(2i) et refm(2i+1), voient
-- leur refl aller dans le meme sens que noeud,
-- alors que les impeirs les voient en sens inverse
pair : for i in 1 to (deuxp_nb_msb/2-1) generate
     -- genere les intervalles pairs, refl croissant
     offset_switch : for j in 1 to offset-1 generate
          soffplus : switch_local
               port map (noeud(2*i, deuxp_nb_lsb-offset+j),
                         ad(2*i+1), refl(j));
          soffmoins : switch_local
               port map (noeud(2*i+1, offset-j),ad(2*i+1),
                         refl(2*offset+deuxp_nb_lsb-j));
     end generate offset_switch;

     switch_refm2i : switch_local
               port map (refm(2*i), ad(2*i+1),refl(offset));
     switch_refm2i1 : switch_local
               port map (refm(2*i+1), ad(2*i+1),
                         refl(offset+deuxp_nb_lsb));

     interne : for j in deuxp_nb_lsb-1 generate
          sw : switch_local
               port map (noeud(2*i, j), ad(2*i+1),
                           refl(j+offset));
     end generate interne;
end generate pair;
```

```
impair : for i in 0 to (deuxp_nb_msb/2)-2 generate

     offset_switch : for j in 1 to offset-1 generate
          soffplus : switch_local
               port map (noeud(2*i, deuxp_nb_lsb-offset+j),
                         ad(2*i+2),
                         refl(deuxp_nb_lsb+2*offset-j));
          soffmoins : switch_local
               port map (noeud(2*i+2, offset-j),ad(2*i+2),
                         refl(j));
     end generate offset_switch;

     switch_refm2i : switch_local
               port map (refm(2*i+1), ad(2*i+2),refl(offset));
     switch_refm2i1 : switch_local
               port map (refm(2*i+2), ad(2*i+2),
                         refl(offset+deuxp_nb_lsb));

     interne : for j in deuxp_nb_lsb-1 generate
          sw : switch_local
               port map (noeud(2*i+1, j), ad(2*i+2),
                         refl(deuxp_nb_lsb-j+offset));
     end generate interne;
end generate impair;

-- traitement des deux segments extremes
--
-- le premier de vp a refm(1), refl meme sens que noeud
offset_switch_vp : for j in 1 to offset-1 generate
     soffplus : switch_local
          port map (vp, ad(1), refl(j));
     soffmoins : switch_local
          port map (noeud(1, offset-j), ad(1),
                    refl(j+offset+deuxp_nb_lsb));
end generate offset_switch_vp;

bout1 : if (offset>0) generate
     switch_vp : switch_local
          port map (vp, ad(1), refl(offset));
     switch_refm1 : switch_local
          port map (refm(1), ad(1), refl(deuxp_nb_lsb+offset));
end generate bout1;

interne_vp : for j in 1 to deuxp_nb_lsb-1 generate
     sw : switch_local
          port map (noeud(0, j), ad(1), refl(offset+j));
end generate interne_vp;



-- le 2nd de vm a rfm(deuxp_nb_msb-1), refl et noeud sens oppose
offset_switch_vm : for j in 1 to offset-1 generate
     soffplus : switch_local
          port map (noeud(deuxp_nb_msb-2, deuxp_nb_lsb-offset+j),
                    ad(deuxp_nb_msb),
                    refl(deuxp_nb_lsb+2*offset-j));
     soffmoins : switch_local
          port map (vm, ad(deuxp_nb_msb), refl(j));
end generate offset_switch_vm;
```

```
    bout2 : if (offset>0) generate
        switch_vm : switch_local
            port map (vm, ad(deuxp_nb_msb), refl(offset));
        switch_refml : switch_local
            port map (refm(deuxp_nb_msb-1), ad(deuxp_nb_msb),
                      refl(deuxp_nb_lsb+offset));
    end generate bout2;

    interne_vm : for j in 1 to deuxp_nb_lsb-1 generate
        sw : switch_local
            port map (noeud(deuxp_nb_msb-1, j), ad(deuxp_nb_msb),
                      refl(deuxp_nb_lsb+offset-j));
    end generate interne_vm;

end structural;
```

BIBLIOGRAPHIE

[Abr93] A. Abrial, J. Bouvier, J-M. Fournier, P. Senn; "A low power 8-b 13.5-MHz video cmos ADC for visiophony ISDN applications"; IEEE journal of solid states circuits, Vol. 28, N° 7, Juillet 1993

[Alt90] M. Altmäe; "MINT : A VHDL simulation system"; Proc. EDAC'90, Avril 1990

[Alt91] M. Altmäe, D. Patrick, C. Lyden; "Mixed VHDL - Analog simulation"; Proc. Euro VHDL'91 pp 260-266 Septembre 1991

[Ana91] Anacad Computer System; "The Eldo-FAS users' manual";

[Ben91] J. Benkoski, J. Besnard, S. Gai, M. Magni, E. Profumo; "Mozart-MM : A mixed-mode and multi-level simulation system"; Proc. of ISCAS'91, pp 2387-2390, Juin 1991

[Ber93] Bergé J.-M., A. Fonkoua, S. Maginot, J. Rouillard; "VHDL Designer' Reference"; Kluwer Academic Publishers 1993

[Bes91] J. Besnard, J. Benkoski, B. Hennion; "Eldo-XL : a software accelerator for the analysis of digital MOS circuits by an analog simulator"; Proc. EDAC'91, pp 136-139, Février 1991

[Bol92] P. Bolcato, R. Poujois; "A new approach for noise simulation in transient analysis"; Proc. ISCAS'92 pp 887-890 Mai 1992

[Bor81] D. Borrione; "Langages de description de systèmes logiques - Proposition pour une méthode formelle de définition"; Thèse d'état INPG Juillet 1981

[Bor85] D. Borrione, C. Le Faou; "Overview of CASCADE multi-level hardware description language and its mixed-mode simulation mechanisms"; Proc. CHDL'85 pp 239-260

[Bor87] D. Borrione, A. El Fadi, C. Le Faou; "Multi-level simulation in CASCADE : examples"; Rapport techniques IMAG-ARTEMIS Février 1987

[Bor93a] D. Borrione, R. Piloty; " CONLAN : Presentation of basic principles, applications and relation to VHDL"; NATO Advanced Study Institute, Fundamentals and standards in Hardware Description Languages, Kluwer Academic Publishers; 1993

[Bor93b] D. Borrione; "CASCADE"; NATO Advanced Study Institute, Avril 1993; Fundamentals and standards in Hardware Description Languages, Kluwer Academic Publishers; 1993

[Bou91a] R. Boute; "Declarative Languages - Still a long way to go"; proc. CHDL'91, pp165-192, Marseille, Avril 1991

[Bou91b] R. Boute; "Language concepts for analog circuit description in VHDL"; VHDL-Forum spring'91 Meeting, Marseille

[Cha92] R. Chadha, C. Visweswariah, C.-F. Chen; "M^3 : A multi-level mixed-mode mixed A/D simulator"; IEEE trans. on CAD, vol. 11, N° 5, pp 575-585, Mai 1992

[Che84] C.-F. Chen, C.-Y. Lo, H. N. Nham, P. Subramaniam; "The second generation MOTIS mixed-mode simulator"; Proc. 21st DAC'84, pp10-17, Juin 1984

[Con67] Condom, Odishaw et al.;"Handbook of physics";Mc Graw and Hill book company, 2nd edition, 1967

[Cot90] R. A. Cottrell; "Event-driven behavioural simulation of analogue tranfer functions", Proc. EDAC'90, pp240-243, Avril 1990

[Dur87] Y. Durand; "Manuel d'introduction à CASCADE"; Rapport technique IMAG-ARTEMIS Novembre 1987

[ElT84] H. El Tahawy, "FIDEL : Un langage de description et de simulation des circuits VLSI", Thése de doctorat INPG Novembre 1987

[ElT87] H. El Tahawy, G. Mazare, B. Hennion, P. Senn; "A new implementation technique for the simulation of mixed (digital-analog) VLSI circuit"; Digest of technical papers. ICCAD'87, Novembre 1987

[ElT89] H. El Tahawy, A. Chianale, B. Hennion; "Functional verification of analog blocks in Fideldo : a unified mixed-mode simulation environment"; Proc. ISCAS'89 pp 2012-2015, mai 1989

[ElT91] H. El Tahawy, D. Rouquier, D. Rodriguez; "Towards a VHDL modeling and simulation language for mixed (analog/digital) VLSI circuits"; Proc. Euro-VHDL'91 pp 251-259, septembre 91

[ElT92] H. El Tahawy, D. Rodriguez, D. Rouquier; "Towards analog-VHDL : some of the problems for mixed simulation"; Proc. VIUF Spring'92, Scottsdale, 3-6 Mai 1992

[ElT93] H. El Tahawy, D. Rodriguez, J-J. Mayol, S. Garcia-Sabiro; "VHD$_e$LDO : a new mixed mode simulation"; Proc. EuroDAC EuroVHDL 93, Hamburg, Septembre 1993

[Fla81] P. L. Flake; "HILO mark 2 hardware description language"; Proc. CHDL'81, pp 95-108; 1981

[Gai88] S. Gai, P. L. Montessoro, F. Somenzi;"MOZART : A concurrent multi-level simulator"; IEEE trans. on CAD/ICAS, vol. 7, N° 9, Septembre 1988

[Get87] I. Getreu; "Analog modeling language spans all system design levels"; Electronic design Mai 1987 pp95-104

[Hen85] B. Hennion, P. Senn, D. Coquelle; "A new algorithm for third generation circuit simulators : the one step relaxation method"; Proc. 22nd DAC'85, juin 1985

[Hil79] D. Hill, W. Van Cleemput; "A tool for generating structured multi-level simulators"; Proc. DAC'79, pp 272-279, Juin 1979

[Hum84] M. Humbert; "Projet CASCADE : une approche de la simulation hiérarchisée multi-modes"; Thèse de Docteur-Ingénieur INPG Octobre 1984

[Kle91] H. W. Klein, "Mixed analog-digital simulation in the 1990's", E. Meyer Editor 1991

[LeF85] C. Le Faou; "Hierarchical multi-level mixed-mode simulation in Cascade"; rapport de recherche IMAG N° 513 Mars 1985

[LeF91] C. Le Faou, J. Mermet; "Requirements for analog and mixed-mode simulation in VHDL"; VHDL-Forum spring'91 Meeting, Marseille

[Lel82] E. Lelarasmee, A. E. Ruehli, A. L. Sangiovanni-Vincetelli; "The waveform relaxation method for time domain analysis of large scale integrated circuits"; IEEE Trans. on Computer-Aided-Design, vol. CAD-1, N° 3, pp 131-145, Juillet 1985

[Lip77] G. J. Lipovski; "Hardware Description Languages : voices from the tower of Babel"; Computer, Vol. 10, N° 6, Juin 1977, pp 14-17

[Man92] H. A. Mantooth, M. Vlach; "Beyond Spice with Saber and MAST"; Proc. ISCAS'92, pp

???-???, Mai 1992

[MHD93] "The MHDL Language Reference Manual"; Intermetrics Inc., IR-VA-025, Février 1993

[Nag75] L. W. Nagel; "SPICE2 : A computer program to simulate semi-conductor circuits"; ERL. Memo N° UCB/ERL M75/520 University of California Berkeley Mai 1975

[Odr86] P. Odryna, S. Nassif; "The ADEPT timing simulation algorithm"; VLSI systems design, Mars 1986

[Pat90] D. Patrick, C. Lyden; "A event-driven transient simulation algorithm for MOS and bipolar circuits"; Proc. EDAC'90, pp230-234, Avril 1990

[Ped84] D. O. Pederson; "A historical review of circuit simulation"; IEEE trans. of circuit and systems, CAS-31, N° 1, pp 103-111, janvier 1984.

[Pil80] R. Piloty, D. Borrione, F. Hill, M. Barbacci, D. Dietmeyer, P. Skelly; "An overview of CONLAN : a formal construction method for Hardware Description Languages"; Proc. IFIP Congress 1980, Tokyo & Melbourne 1980

[Pil83] R. Piloty, D. Borrione, F. Hill, M. Barbacci, D. Dietmeyer, P. Skelly; "CONLAN Report"; Lecture notes in Computer Science N° 151, Springer Verlag, 1983

[Pil85] R. Piloty, D. Borrione; "The CONLAN project : concepts, implementations and applications"; Computer, pp 81-92, Février 1985

[Rod93a] D. Rodriguez; "Description et Simulation mixte analogique numérique - Propositions pour VHDL-analogiques"; Rapport de Recherche IMAG-ARTEMIS N° 908-I-, Janvier 1993

[Rod93b] D. Rodriguez; "Analog-VHDL : as an application, a real example"; Proc. CHDL'93, Ottawa, Avril 1993

[Rou91] D. Rouquier, D. Rodriguez; "Analog simulation using VHDL"; VHDL-Forum spring'91 Meeting, Marseille

[Sak81] K. A. Sakallah; "Mixed mode simulation of electronic integrated circuit"; PhD thesis, Design Research Center, Carnegie-Mellon, Pittsburgh, PA, Novembre 1981

[Sak85] K. A. Sakallah, S. W. Director; "Samson2 : an event-driven VLSI circuit simulator"; IEEE trans. on CAD, vol. CAD-4, N° 4, Octobre 1985

[Sal87] R. A. Saleh, A. R. Newton; "An event driven, relaxation-based multirate integration scheme forcircuit simulation"; Proc. ISCAS'87, Philadelphia, Pennsylvania, pp 600-603, Mai 1987

[Sal90] R. A. Saleh, A. R. Newton; "Mixed-mode simulation"; Kluwer Academic Publishers; 1990

[Vac93] A. Vachoux, K. Nolan; "Analog and mixed-level simulation with implications to VHDL"; NATO Advanced Study Institute, Fundamentals and standards in Hardware Description Languages, Kluwer Academic Publishers; 1993

[VHD87] "The VHDL Language Reference Manual"

[VSt90] M. T. Van Stiphout, J. T. J. Van Eidhoven, H. W. Buurman; "PLATO : A new piece wise linear simulation tool"; Proc. EDAC'90, pp 235-239, Avril 1990

[Whi87] J. K. White, A. Sangiovanni-Vincetelli; "Relaxation techniques for the simulation of VLSI circuits"; Kluwer Academic Publishers; 1987

INDEX

Résumé

Les outils informatiques prennent une place de plus en plus importante dans la conception de circuits VLSI. Les langages de description de matériel constituent l'interface entre ces outils et les utilisateurs. Parmi ceux-ci, il existe un standard qui est VHDL, destiné à la description de systèmes numériques. Actuellement une extension analogique est en cours de normalisation. Les deux premiers chapitres de cette thèse sont consacrés l'un aux langages de description de matériel et à une présentation de VHDL, ainsi que des remarques et analyses à propos de son extension analogique. Le second thème de cette thèse est la mise en évidence de l'importance de la simulation en mode mixte numérique-analogique. Le troisième chapitre présente les principes généraux de la simulation mixte; différentes implémentations de simulateurs mixtes sont présentés. Enfin, le dernier chapitre est consacré à la réalisation d'un simulateur mixte dont la partie numérique est un simulateur VHDL. Cette réalisation repose sur une approche de description qui permet d'utiliser la souplesse de description structurelle de VHDL pour des systèmes analogiques et mixtes.

Mots clés

simulation, simulation mixte, CAO micro-électronique, langages de description de matériel, VHDL, VHDL-analogique, description mixte

Mixed analog-digital Description and Simulation : Study of Analog-VHDL, Implementation of a mixed simulator

Abstract

CAD tools take an increasingly important place in the design of VLSI circuits. Hardware Description Languages are the interface between the tools and the users. Among these languages, there is a standard which is VHDL. It is designed for the description of digital circuits, but an analogue extension is currently being normalised. The two first chapters of this thesis are dedicated to the hardware description languages, to an overview of VHDL, and commentaries about its analogue extension. The second topic of this thesis is the mixed mode digital-analogue simulation of VLSI circuits. The third chapter presents the general concepts of the mixed mode simulation, and several implementation of mixed simulators. Finally, the last chapter is dedicated to the implementation of a mixed mode simulator, the digital part of which is a VHDL simulator. This work relies on the definition of a specific method which makes use of the VHDL facilities for structural description for analogue and mixed systems.