Initiation au language de modélisation Verilog A(MS)©

François BERRY

1 Présentation

Durant la conception d'un circuit intégré analogique, il est souvent nécessaire de pouvoir simuler chaque partie du circuit de manière indépendante. Toutefois dans de nombreux cas, chaque module est dépendant des modules voisins et il est alors impossible de faire une simulation réaliste de chaque module. En effet, la validation de chaque bloc doit alors attendre que les autres soient terminés et ainsi de suite. Il est donc nécessaire de pouvoir décrire le comportement de chaque bloc au cours du temps en fonction des différents paramètres de simulation, de manière à obtenir une première ébauche de résultats. Cette description s'appelle **la modélisation comportementale**. Il peut aussi arriver dans certains cas, que les temps de simulations deviennent prohibitifs (Simulation d'ensemble ADC, DAC avec plusieurs Ampli Op). Pour valider rapidement un schéma, il est alors intéressant de décrire les blocs coûteux en temps avec Verilog A(MS) de manière à gagner un facteur temps significatif.

Bien que très utilisé en digital, les languages de modélisation permettant de décrire de manière simple un composant analogique sont apparus récemment (1995). Verilog AMS (AMS= analog and mixed signal) fait partie de ces langages et est un des plus aboutis à l'heure actuelle.

Un point particulièrement important à noter est la différence fondamentale entre un language de modélisation analogique tel que Verilog-AMS et un système de simulation comme Spice©ou Spectre©. Un outil tel que Spice permet de réaliser de manière relativement efficace différents type de simulation. Cependant, il est limité à quelques composants de base tels que les résistances, condensateurs, transistors. Aussi, dans certains cas, il peut être intéressant d'utiliser des blocs utilisant un plus au niveau d'abstraction tel qu'un module amplificateur opérationnel, un PLL, un ADC/DAC,... Ces blocs de haut niveau peuvent alors être codé simplement en Verilog-AMS puis introduit dans un schéma de simulation de manière à le valider facilement.

Dans un premier temps, nous nous intéresserons au langage VerilogA dédié à la description de systèmes purement analogiques, puis la seconde partie du TP nous aborderons l'aspect mixte en VerilogAMS.

2 Quelques règles et principes de Verilog-A(MS)

Nous présentons ici les grands principes de fonctionnement du langage et certaines définitions-clefs nécessaires à la compréhension des bases du langage.

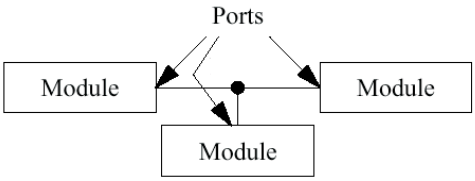
2.1 Composants et connexions

On désignera par **système** un ensemble de composants interconnectés entre eux et réagissant à des stimuli en produisant une réponse. Chaque **composant** peut lui-même être un système (PLL, modulateur,...) ou simplement une brique de base (résistance, transistor, multiplieur...).

La description d'un composant se fait dans un **module**. Les composants sont connectés entre eux par des **nets**(équipotentielles pour l'aspect électrique) sur lesquels transitent des signaux.

Un **signal** est une entité physique transitant sur un ou plusieurs nets. Ainsi un signal numérique codé sur 8 bits transitera sur 8 nets. Un signal peut appartenir soit au domaine continu (analogique) soit au domaine discret (numérique). Un signal qui appartient au deux domaines est dit mixte (mixed signal).

La connexion d'un signal à un composant se fait par l'intermédiaire de **port**. De la même manière que pour les signaux, il existe des ports analogiques, numériques et mixtes.



2.2 Codage d'un composant

Afin d'introduire de manière claire, les possibilités du langage, nous allons décrire un premier composant totalement fictif. Toutefois, il convient avant tout d'introduire la notion de module qui permet de:

- définir le comportement d'un composant
- déclarer l'interfaçage nécessaire à son insertion dans le schéma

La partie déclarative du module sera composé des signaux d'entrée/sortie et des paramètres (réglables ou fixes) du module. On se propose ici de construire un composant nommé `pipo` et qui admettra 4 pins.

Exemple 1 Exemple de module

```
`include "discipline.h"
`include "constants.h"

module pipo (a,b,c,masse);

    input a;
    output b;
    inout c,masse;

    electrical a, b,c,masse;

    parameter real f=2.0e2 from [0:inf) exclude (100:1G);
    parameter integer d=1;

    electrical n1,n2;

    real tempvar;
    integer botablo[1:3];

    // Ceci est une ligne de commentaires
    analog begin

        @(initial_step) begin
            V(n1)<+1.0;
        end;

        // Description comportementale du module
        tempvar= f*2.0;
        V(n1)<+ tempvar*V(a)+V(c);

        @(final_step) begin
            V(n1)<+8.0;
        end;

    end

        // Description structurelle du module
        resistor #(.r(d)) R1 (n1,b);
        inductor #(.l(10n)) L1 (b, masse);

endmodule
```

Nous allons à présent détailler les différentes parties de ce module qui comprend la plupart des déclarations que l'on peut trouver.

1. Inclusion des 2 fichiers standards (disciplines et constants) qui contiennent les déclarations de base. On notera qu'en VerilogA les fichiers ont l'extension *.h* et qu'en VerilogAMS les fichiers ont l'extension *.vams*

2. Déclaration du composant (ici type `pipo`) avec 4 pins (`a, b,c,masse`)

3. Une pin sera une entrée (pin `a`), une pin sera une sortie (pin `b`) tandis que les 2 dernières seront des entrées et sorties (pins `c` et `masse`)
4. Le type de signal transitant par ces quatres pins sera de type électrique. On verra plus loin la possibilité de déclarer d'autre type, tel que le magnétisme ou la thermique.
5. Deux variables paramétrables par l'utilisateur sont déclarées. La première `f` de type réelle admet en valeur par défaut 2.10^2 et peut varier de $[0; 100[\cup]10^9; \infty[$. Le second paramètre `d` est de type entier et admet en valeur par défaut 1.
6. Deux noeuds internes (`n1` et `n2`) sont ici spécifiés de type électrique.
7. Déclaration d'une variable `tempvar` interne au composant de type réel et d'un tableau d'entier de 3 cases nommé `botablo`.
8. Début du bloc de description analogique comportementale. On notera la ligne de commentaire débutant par //
9. Cette partie facultative permet de déclarer une initialisation au premier pas de simulation. Dans le cas présent, la tension du net `n1` aura à l'instant t=0, la valeur `V(n1)`=5.0.
10. Cette partie correspond à la modélisation comportementale d'un morceau du composant. Ici la variable `tempvar` prend la valeur égale à deux fois `f`, puis la tension de la pin d'entrée `a` est multipliée par `tempvar` et additionnée à la tension de la pin d'offset `c`. Pourquoi utilise-t-on soit le signe = soit le signe $<+$?
11. Cette partie facultative permet de déclarer des conditions au dernier pas de simulation. Dans le cas présent, la tension du net `n1` aura à l'instant t=T_final, la valeur `V(n1)`=5.0..
12. Fin de description du bloc analogique comportementale ouvert en 8 par **analog begin**
13. Description structurelle de l'autre morceau du composant. Ici une résistance `R1` de valeur paramétrable dΩ est fixée entre les noeuds `n1` et `b` et une inductance `L1` de valeur fixe 10nH est fixée entre les nœuds `b` et la `masse`.
14. Fin du module ouvert en 2 par **module**

A partir de cette description, dessiner le schéma du composant de type `pipo` et expliquer son fonctionnement.

3 Création du composant

Dans cette partie, nous allons créer le composant à travers sa description et son symbole permettant de l'insérer dans un schéma. Pour cela, créer une librairie que l'on nommera par exemple **TPVerilogA**. Cette librairie n'est naturellement pas liée à un kit fondeur donc on ne l'attachera pas à un "tech file"! Puis, créer un composant dans cette librairie en spécifiant la vue **veriloga**. Suivant la version d'ICC, soit un simple éditeur texte va s'ouvrir(ICC 4.4.5), soit l'outil **modelwriter** (ICC 4.4.6) propose une aide à la description à partir de modèle prédéfini.

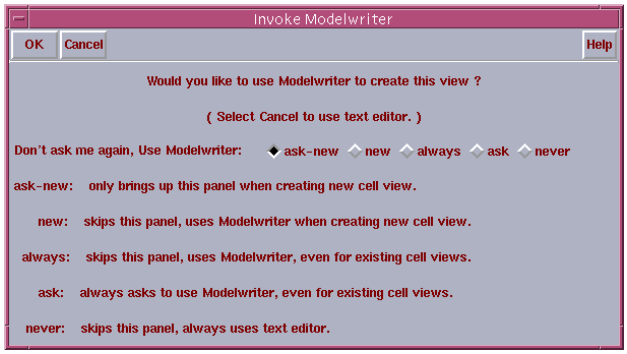


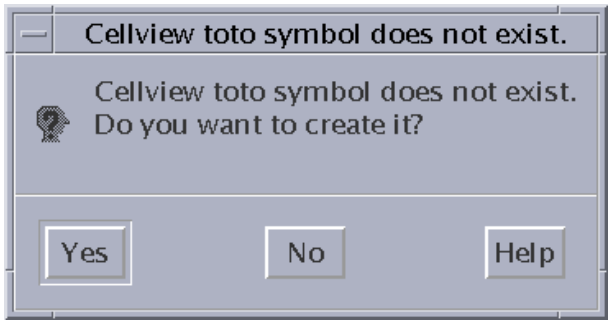
Fig. 1 – *Fenêtre d'invite de Modelwriter*

Bien que ce dernier outil soit relativement bien fait, il ne permet pas à l'étudiant d'assimiler de manière efficace les grandes idées du langage donc nous en limiterons dans un premier temps l'utilisation en cliquant simplement sur **Cancel**. Il est cependant de bon ton de garder l'option "ask new" coché, ce qui permet d'avoir le choix pour chaque nouvelle modélisation d'utiliser modelwriter ou pas.

A l'aide de l'éditeur de texte, saisir le modèle du composant `pipo` tel qu'il est proposé dans le listing 1, puis sauver. A ce moment là, le code saisi est alors vérifié par le "parser" de verilog-A et une fenêtre (Fig. 2) spécifiant les erreurs de syntaxe éventuelles apparaît.

Fig. 2 – *Parsing d'un code Verilog AMS*

Naturellement, il convient de n'avoir aucune erreur ni "warnings". Un fois, la vérification de syntaxe réalisée, le système propose alors de créer un symbole, s'il n'existe pas avec la fenêtre suivante.



En validant cette dernière le système détecte automatiquement les entrées/sorties du composant et la fenêtre suivante permet de positionner les pins en question.

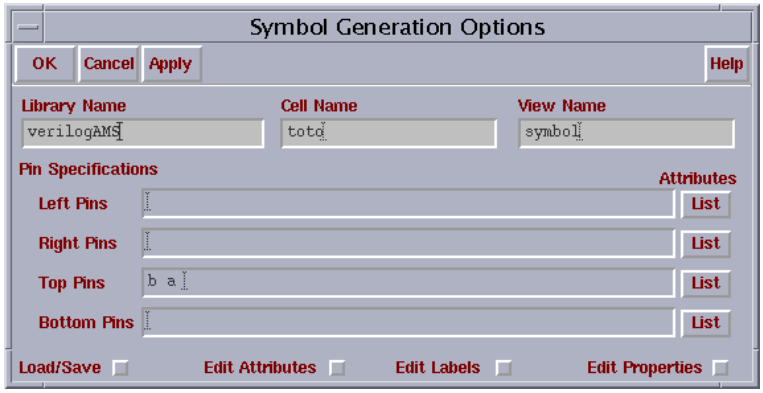


Fig. 3 – *Réalisation du symbole*

Nous disposons à présent d'un composant n'ayant qu'une existence "logicielle" et un symbole. Il convient donc d'en vérifier son fonctionnement à partir d'un simulation analogique.

Pour cela créer un schéma de simulation, insérer le symbole du composant de type `pipo` et lancer le simulateur Analog Artist. Une fois dans le simulateur analogique, après avoir paramétré la simulation, il est important de spécifier au simulateur qu'il devra utiliser la vue **veriloga** pour faire sa simulation. Pour cela, dans les options d'environnement, insérer dans la liste des vues utilisées par le simulateur, la vue **veriloga**:

Lancer la simulation et vérifier le bon fonctionnement du composant réalisé. On notera la possibilité, en éditant les propriétés du composant dans le schéma, de pouvoir modifier le gain et la valeur de la résistance qui était déclaré en paramètre dans le code Verilog-A. Que se passe-t-il pour une résistance nulle (d=0) et pourquoi?

4 Principe de description d'un système analogique

Après avoir vu comment il était possible d'insérer et de simuler un composant décrit, nous donnons dans cette partie les différents principes de description des systèmes analogiques.

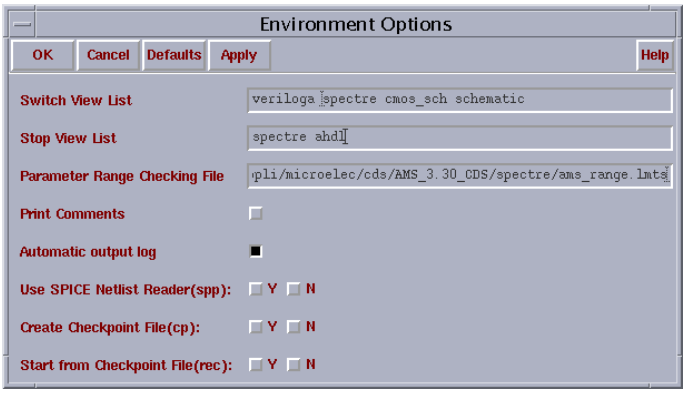


Fig. 4 – *Insertion de la vue veriloga dans Analog Artist*

4.1 Types de description

La description d'un composant ou plus généralement d'un système peut se faire de différentes manières. Toutefois, on n'en dénombrera deux grandes catégories:

4.1.1 Description structurelle (Structural description)

Le système est décrit à partir de ses composants élémentaires et fait apparaître une hiérarchie explicite dans la description. Typiquement en considérant le système suivant décrivant une PLL élémentaire constituée d'un multiplieur, d'un filtre passe-bas et d'un VCO (Voltage Controled Oscillator):

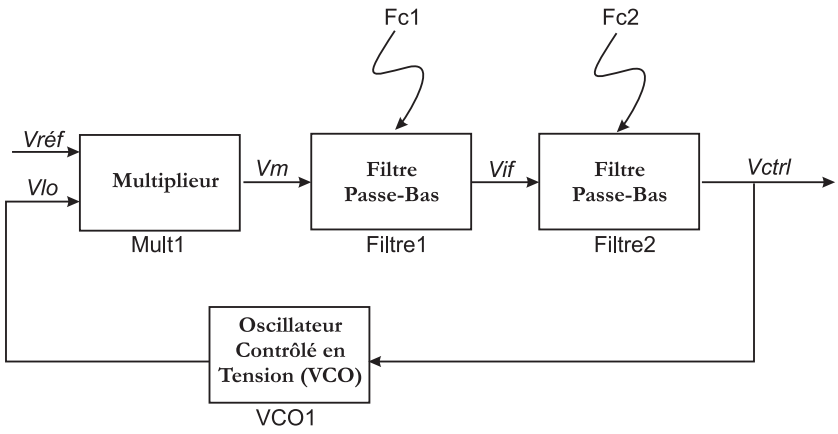


Fig. 5 – *Schéma structurel d'une PLL*

on obtiendra la description suivante:

Exemple 2 Description structurelle de la PLL de la figure 4

```
module PLL (Vréf,Vctrl);

    input Vréf;
    output Vctrl;
    electrical Vréf,Vctrl;

    parameter real fc1=100.0, fc2=30.0;
    electrical Vlo, Vif, Vm;
    // Description de la PLL

    Multiplieur Mult1(Vréf, Vlo, Vif);
    FiltrePB  #(.freq_coupure(fc1)) Filtre1(Vm, Vif);
    FiltrePB  #(.freq_coupure(fc2)) Filtre2(Vif, Vctrl);
    VCO vco1(Vctrl, Vlo);

endmodule
```

On notera tout d'abord la méthode d'instantiation des blocs. Prenons l'exemple du filtre passe-bas soit la ligne:

```
FiltrePB  #(.freq_coupure(fc)) Filtre1(Vif, Vctrl);
```

Dans cette ligne `FiltrePB` définit le type de composant, l'argument `#(.freq_coupure(fc))` permet de régler le paramètre `freq_coupure`(propre au type de composant FiltrePB) à la valeur `fc` et la partie `Filtre1(Vif, Vctrl)` explicite le nom du composant dans le schéma (ici Filtre1) et ses connexions (Vif et Vctrl). On notera que ceci est la même syntaxe que le bloc 13 de l'exemple 1 (page 3).

4.1.2 Description comportementale (Behavioral description)

Contrairement à la description structurelle, la description comportementale ne considère par la structure du composant mais son comportement. Ainsi, une description de type comportementale est basée sur des relations mathématiques qui lie les signaux d'entrée aux signaux de sortie.

4.2 Type de systèmes

Afin de réaliser une simulation complète d'un système, il ,est nécessaire comme nous l'avons vu précédemment d'en avoir une description soit structurelle soit comportementale. Ces différentes descriptions permettent au compilateur de transformer le code en équations différentielles puis au simulateur de les résoudre. Toutefois on dénombre habituellement deux type de systèmes en description analogique:

– Les systèmes de type conservatif

– Les systèmes de type "signal flow"

4.2.1 Systèmes conservatifs

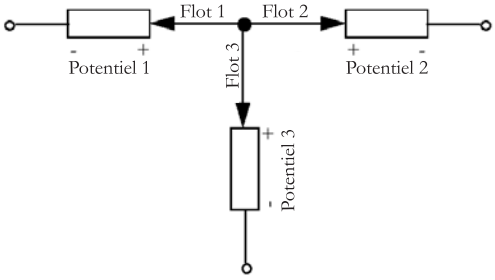
Un système dit conservatif est un système répondant au loi de kirchoff. On le dit conservatif dans le sens où la puissance globale du système est conservée.

Un des aspects les plus importants à garder à l'esprit est que dans un système conservatif, chaque nœud ou signal est associé à **deux** grandeurs qui sont de type:

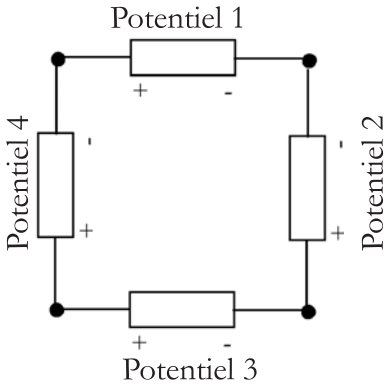
– potentiel (potential)
– flot (flow)

S'il on considère un système électrique, il est facile d'associer ces deux grandeurs au potentiel (dont la différence crée une tension) et au courant.

Ces deux grandeurs vérifient alors les loi de Kirchoff qui sont:



flot1+flot2+flot3=0



potentiel1+potentiel2+
potentiel3+potentiel4=0

Lorsque l'on déclare un port avec l'attribut `electrical`, cela revient à dire qu'il répond à la grandeur physique électricité et que par conséquent il est défini par une tension et un courant. D'autres disciplines sont prédéfinies avec les grandeurs correspondantes et sont données sur le tableau ci-dessous:

Discipline	Déclaration du type de signal	Type de Potentiel	Type de Flot	Accès au type potentiel	Accès au type flot
Electricité	electrical	Tension	courant	V	I
Magnétisme	magnetic	Force Magnéto-motrice	Flux	MMF	Phi
Thermique	thermal	Temperature	Puissance	Temp	Pwr
Déplacement linéaire	kinematic	Position	Force	Pos	F
Déplacement angulaire	rotational	Angle	Force angulaire	Theta	Tau

Dans un système conservatif, il est aussi commun de définir la notion de **branche** (*branch*). Une branche est un "chemin de flot" [1] entre deux noeuds. Chaque branche a un potentiel associé et un flot. La convention prise en Verilog-A(MS) est:
Le flot va dans le sens des potentiels décroissants.

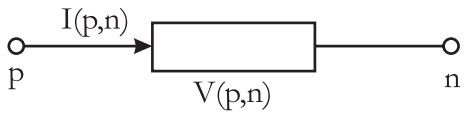


Fig. 6 – *Convention Tension/Courant dans une branche*

De plus la convention veut que l'on note **I(p,n)**, le courant circulant du nœud p au nœud n et **V(p,n)** le potentiel entre le nœud p et le nœud n.

Ecrire et simuler le modèle Verilog-a d'une résistance prenant en compte la température ambiante. On rappelle qu'une résistance peut être modélisée par la relation suivante:

$$R = R_{nom}.(1 + \alpha(T_{amb} - T_{nom}) + \beta(T_{amb} - T_{nom})^2)$$

avec:

- R_{nom}: Résistance nominale,
- α: Coefficient de température linéaire,
- β: Coefficient de température quadratique,
- T_{nom}: Température nominale (par exemple 27^{o}C soit 300^{o}K)
- T_{amb}: Température ambiante.

Dans cette modélisation, il est nécessaire de connaître la température de simulation qui est une condition environnementale de la simulation. L'accès à ces variables se fait de la manière suivante:

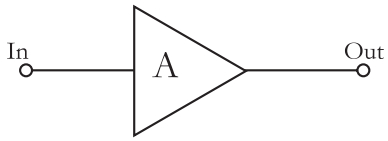
Variable environnementale	Accès
Température de simulation	**$temperature()**
Temps courant de simulation	**$realtime()**
Tension thermique de jonction (ambiante)	**$vt()**
Tension thermique de jonction à T=temp	**$vt(**`temp`**)**
Génération d'un nombre aléatoire	**$random()**

Modifier le modèle en ajoutant une dérive thermique dû à l'échauffement propre de la résistance. On prendra par exemple une dérive linéaire dans le temps de coefficient paramètrable γ (en Ω.W.s par exemple).

4.2.2 Systèmes de type "signal flow"

La description de type "signal flow" est la représentation qui est souvent utilisée sous forme de schéma blocs (ou boîte noire) comme en automatique. Dans ce cas de figure, les sorties ne sont pas influencées par les entrées (impédance de sortie nulle et d'entrée infinie). Le support est à temps continu. Typiquement la représentation d'un étage de gain en représentation "signal flow" sera la suivante:

1. Traduction brutale de l'anglais: "path of flow"



On considère la description d'un ampli-op parfait admettant la fonction de transfert suivante :

$$V_{sortie} = A.(V_{noninverse} - V_{inverse})$$

où A est le gain différentiel de l'amplificateur.

A partir de cette description, écrire le programme Verilog-A de ce composant en posant A comme paramètre réglable par l'utilisateur. Réaliser un schéma de simulation (ampli inverseur) et simuler. On pourra par exemple utiliser les résistances précédentes.

5 Les différents opérateurs

Dans cette section, on se propose de passer en revue les différents opérateurs permettant de construire des modèles plus élaborés.

5.1 Opérateurs analogiques

Comme nous avons pu le voir précedemment, il est possible de réaliser des dérivés et intégrations temporelles. En voici le détails:

5.1.1 Intégration temporelle

$$\int_0^t (expr).dt + ic$$

cette expression est codée par la ligne suivante:

idt(expr,*ic*,*reset*)

où :

- expr: est l'expression à intégrer
- ic : (paramètre facultatif) est une condiotn initiale
- reset : (paramètre facultatif) est un "flag" (0 ou 1) déclenchant le reset de l'intégration.

5.1.2 Dérivée temporelle

$$\frac{d}{dt}(expr)$$

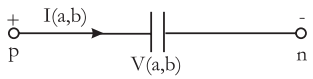
cette expression est codée par la ligne suivante:

ddt(expr)

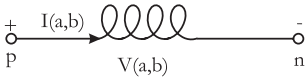
où :

– expr: est l'expression à dériver

Ecrire puis simuler les modèles d'une capacité parfaite et d'une inductance parfaite. Dans une grande bonté, on rappelle que:



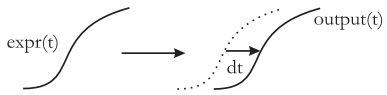
$$I_c = C.\frac{d}{dt}.V(a,b)$$



$$I_L = \frac{1}{L}\int_0^t V(a,b).dt$$

5.1.3 Insertion d'un retard

Il est peut être nécessaire pour certains systèmes d'insérer un retard pur afin de reproduire plus fidèlement leur comportement. cette expression est codée par la



ligne suivante:

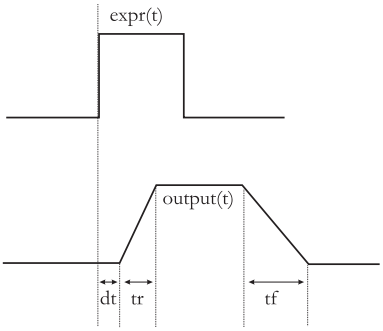
delay(expr,dt)

où :

– expr: est le signal à retarder
– dt: est le retard en seconde

5.1.4 Gestion des transition

Dans une modélisation mixte qui nécessite la collaboration de partie digitale et analogique, il est fondamentale de pouvoir gérer la forme des transitions. En d'autres termes comment une transition parfaite (digitale) se transforme en transition imparfaite (analogique)



cette expression est codée par la ligne suivante:

transition(expr,dt,tr,tf)

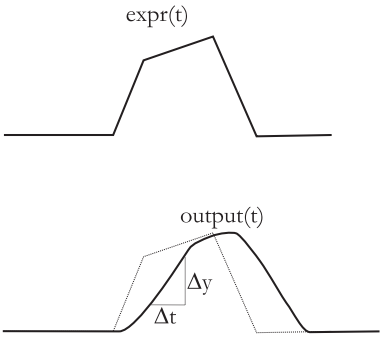
où :

– expr: est le signal d'entrée

– dt: est le retard en seconde entre le signal d'entrée et celui de sortie
– tr: est le temps de montée du signal
– tf: est le temps de descente du signal

5.1.5 Gestion de la pente des signaux

Cet opérateur permet de déclarer les "slew rate" maximums admissibles sur un signal. Il est possible de régler séparément la pente maximale de montée et celle de descente.



Ainsi sur l'exemple de la figure ci-dessus, le signal `output(t)` recopie `expr(t)` tant que $\frac{\Delta y}{\Delta t} < pm$.

Cette expression est codée par la ligne suivante:

slew(expr,pm,pd)

où :

– expr: est le signal d'entrée
– pm: est la pente maximale de montée admissible
– pd: est la pente maximale de descente admissible

Reprendre la modélisation de l'ampli-op et rajouter des défauts (slew-rate, décalage, mode commun, assymétrie, ...)

5.2 Opérateurs événementiels et conditionnels

5.2.1 Gestion des conditions

Verilog-A(MS) supporte la définition de tests conditionnels de la forme **if-else**. La syntaxe d'un tel test se fait de la manière suivante:

```
if(condition)
 <Action si la condition est remplie>
else
 <Action si la condition n'est pas remplie>
```

On notera la possibilité de coder une condition à partir des opérateurs ternaires **?-:**. La forme générale est alors donnée par:

```
condition? <Action si la condition est remplie> : <Action si la condition n'est pas remplie>
```

Dans l'exemple 3 est présenté un code utilisant cet opérateur. Que fait ce programme? Reprendre ce programme et réécrire la partie grisée en 1 ligne avec les opérateurs ternaires.

Exemple 3 Que fait le module Mystere?

```
`include "discipline.h"
`include "constants.h"

module Mystere (a,b,c);
    input a,b;
    output b;

    electrical a, b,c;

    real tempvar;

    analog begin
        if (V(a) > V(b))
            tempvar = V(a);
        else                                              à réécrire
            tempvar = V(b);
        V(c)<+ tempvar;

    end

endmodule
```

5.2.2 Gestion des événements

Le comportement analogique d'un composant peut être modélisé en utilisant ce que l'on appelle les événements analogiques (*analog events*). Ces événements ont trois caractéristiques:

- ils peuvent être déclenchés ou détectés dans un modèle comportemental
- ils ne bloquent pas l'exécution du module (contrairement à un mauvais **if-else**)
- ils sont détectés par l'opérateur @.

Il sont codés de la manière suivante:

```
@ (Condition evenementielle) Action associee
```

Toutefois ces conditions événementielles se présentent sous deux formes que nous allons détailler.

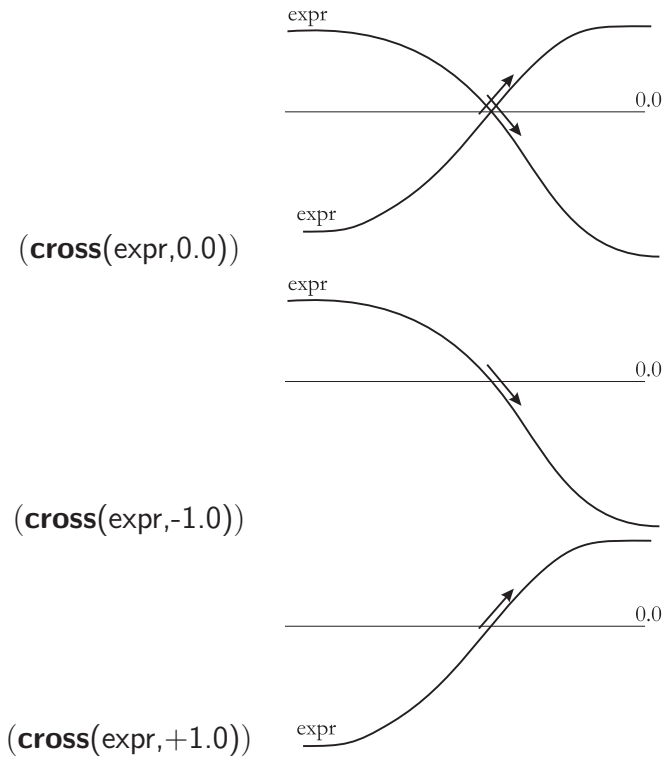
5.2.2.1 Détection de seuil La détection de seuil fait appel à l'opérateur **cross** qui permet de déclencher une condition dès que le signal passe par la valeur 0. Le codage de cet opérateur se fait de la façon suivante:

cross(expr,dir,timetol,valuetol)

où :

- expr: est le signal d'entrée à comparer à 0
- dir: est le sens de passage (de - à + ou de + à -)
- timetol(facultatif): est la tolérance sur la résolution temporelle
- valuetol(facultatif): est la tolérance sur la valeur de expr lors de la comparaison

Ci dessous est présenté la spécification du paramètre dir

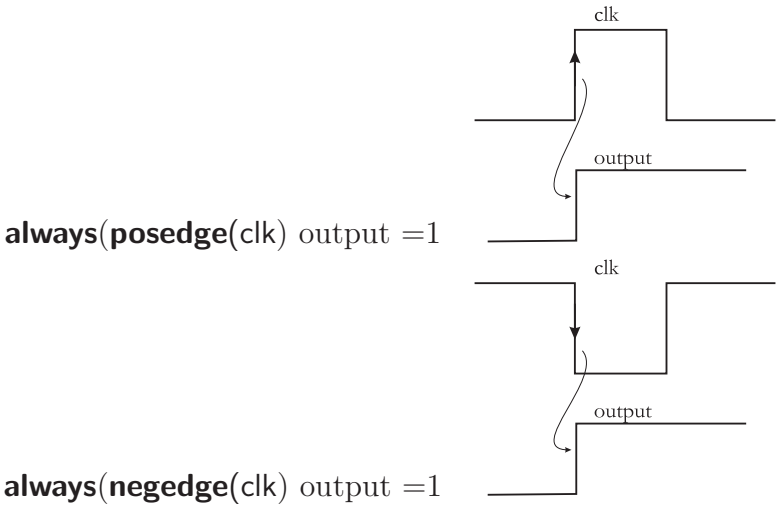


A partir de toutes les instructions vues à présents, modéliser un composant qui fournira soit l'intégrale soit la dérivée du signal d'entrée *Input*. Le choix du traitement sera fait par une impulsion supérieure à un seuil paramètrable sur une broche nommée *Control*. Si le résultat dépasse un second seuil paramètrable nommé *saturation*, alors un flag $overflow$ passera à l'état haut (+5v).

Posedge et Negedge

Bien que ***n'intervenant qu'en digital et en mixte***, il est important de noter ces deux opérateurs utilisés la plupart du temps pour la détection de front d'horloge. En considérant un signal d'horloge `clk`, et un signal `output` passant de $0 \rightarrow 1$ sur

front montant ou descendant, il vient:



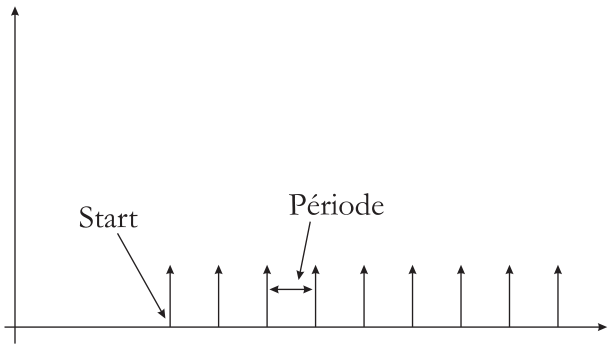
Dans l'exemple ci-dessus, le mot `always` permet de vérifier la condition à chaque point de simulation. Gardons tout de même en tête que cette instruction n'est pas valable en analogique pur!

5.2.2.2 Timer La condition **timer** est utilisée pour générer des événements à intervalles réguliers et de période fixe. La syntaxe utilisée est la suivante:

timer(start,période)

où :

- start: temps de départ du timer par rapport au temps absolu (début de la simulation)
- période: période selon laquelle les événements seront générés.



5.2.2.3 Remarques Comme nous l'avons vu au tout début, il existe aussi les événements **initial_step** et **final_step** permettant des déclenchements uniquement au premier et au dernier point de simulation.

6 Premiers pas avec des signaux mixtes

Verilog-AMS est né de la fusion de Verilog-A et Verilog-HDL. Pour cette raison, le language AMS contient de nombreux opérateurs (surchargeant largement

Verilog-A) que nous ne pouvons aborder en quelques heures. Pour cette raison, le lecteur intéressé se référera avantageusement à la documentation Cadence en ligne (open book) ou tout simplement à *"Verilog-AMS Langage Reference Manual"* (www.ovi.org).

6.1 Introduction au flot de conception AMS

Comme on peut aisément s'en douter la difficulté majeure en conception mixte est de faire cohabiter des signaux analogiques (à temps continu) et digitaux (à temps discret). Pour cette raison, les disciplines standards de verilogA ont été augmentées et remises à jour. Ainsi la bibliothèque `discipline.h` a été augmentée et renommée `discipline.vams`. Cette dernière contient entre autre ces nouvelles lignes:

Exemple 4 discipline.vams

```
// Electrical - Analog
discipline electrical
    domain continuous                                          1

    potential = Voltage
    flow = Current
enddiscipline

// Logic - Digital
discipline logic
    domain discrete                                            2

enddiscipline
```

où les lignes respectivement 1 et 2 spécifient l'espace temporel de travail. Ainsi un module mixte (type comparateur) prenant deux signaux (`a` et `b`) d'entrée analogiques à comparer et un signal (`s`) de sortie digital tel que:

$$\begin{cases} V(a) \geq V(b) \Rightarrow s = 1 \\ V(a) < V(b) \Rightarrow s = 0 \end{cases}$$

sera codé de la manière suivante:

Exemple 5 comparateur.vams

```
‘include discipline.vams
‘default_discipline logic                                    1

module comparateur(a,b,s);
input a,b;
output s;

electrical a,b;
logic s;                                                     2

reg s;                                                       3

integer temp;

analog begin
    if (V(a)>= V(b))
    temp=1;
    else
    temp=0;                                                  4
end

always begin
    #5 s=temp;                                               5
end

endmodule
```

La partie 1 spécifie que par défaut les entrée/sortie sont de type `logic`. Donc si `a` et `b` (dans la partie 2) n'avaient pas été spécifiée `electrical`, alors ils auraient été considéré comme des ports `logic`.

Dans la partie 3, la sortie est déclarée de type `reg`. Cette déclaration vient du Verilog (digital) et indique que s appartient à la classe *reg*. Par conséquent s est équivalent à une variable dans un programme informatique. Il subit des affectations instantanées par instructions et conserve son état jusqu'à la prochaine affectation.

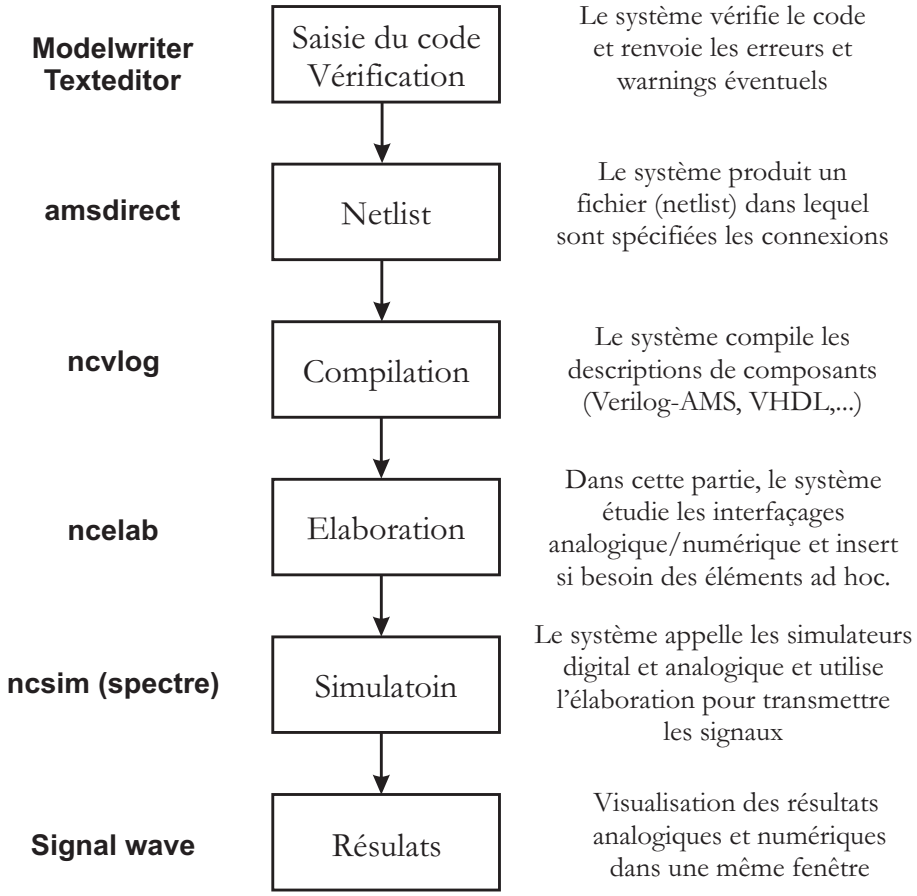
La partie 4 correspond à la partie analogique du module. **Dans un module mixte, il ne peut y a avoir qu'un seul bloc analogique.**

La partie 5 correspond à la partie numérique du module. **Dans un module mixte, il peut y a avoir autant de bloc digital que nécessaire.**

on notera la variable intermédaire `temp` de type entier permettant de passer du monde analogique au monde digital.

Avant de réaliser un premier composant, il est toutefois nécessaire de comprendre comment Cadence travaille avec un code Verilog-AMS. L'analyse (parsing) du code et sa transformation passent par plusieurs étapes dont certaines sont communes à tout système de simulation et d'autres propres aux systèmes mixtes. En voici le

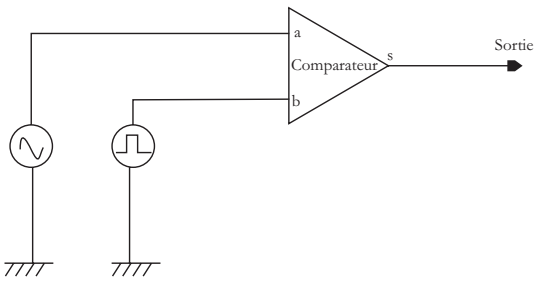
détail:



6.2 Présentation de l'environnement Cadence AMS

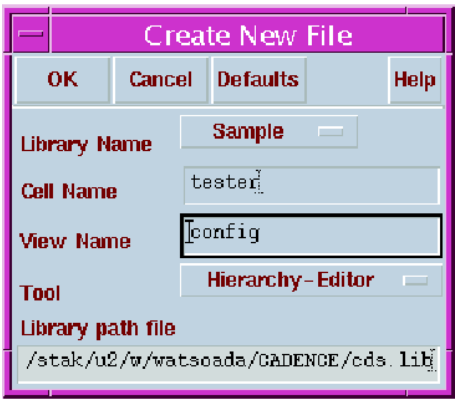
Les outils de Cadence pour faire du mixte sont divers et variés. En effet, il est possible soit de lancer toutes les opérations à partir de shells, soit d'utiliser l'environnement graphique dédié à cet effet. Dans notre cas, nous opterons pour la seconde solution bien que ce ne soit pas forcément aisée... Là encore, la lecture de l'openbook Cadence peut s'avérer d'un grand secours mais pas toujours très efficace(!)

Afin d'illustrer notre propos, nous allons reprendre le comparateur vu précédemment et le simuler de manière mixte. Dans le cas présent, ce composant admet deux entrées analogiques et une sortie digitale. Après avoir saisi le code du comparateur, céer un symbole et l'insérer dans un schéma de test du type ci-dessous:

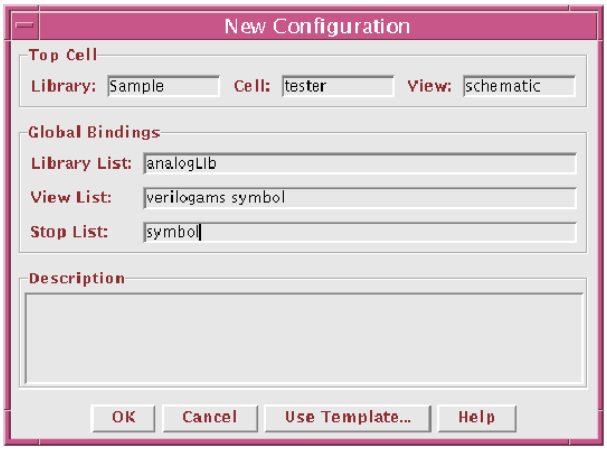


Une fois le schéma de simulation sauvé, revenir dans le library manager et créer

une nouvelle vue au schéma de simulation en invoquant l'outil Hierarchy-Editor.



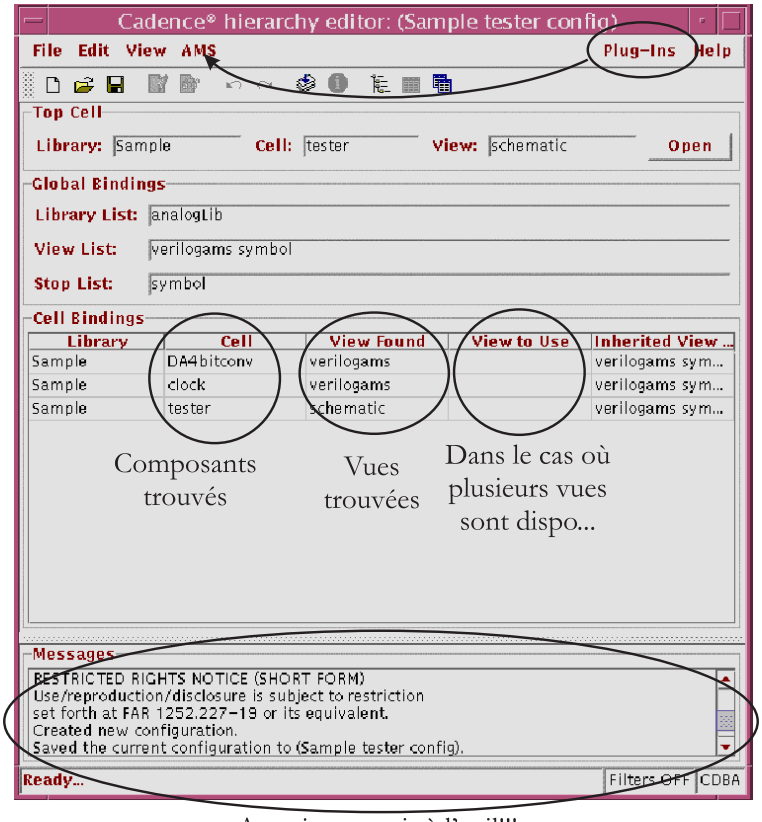
L'outil Hierarchy-Editor permet de gérer de manière relativement souple les vues à utliser pour la simulation mixte. A la première ouverture, une fenêtre de configuration demande les caractéristiques du schéma.



Dans cette fenêtre, on prendra soin de remplir les caractéristiques du schéma père (top cell) avec la vue schematic. Dans la zone Global Bindings, il convient de rentrer les bibliothèques qui seront utilisées (analogLib, HRDLIB,...) et ainsi que les vues utlisables.

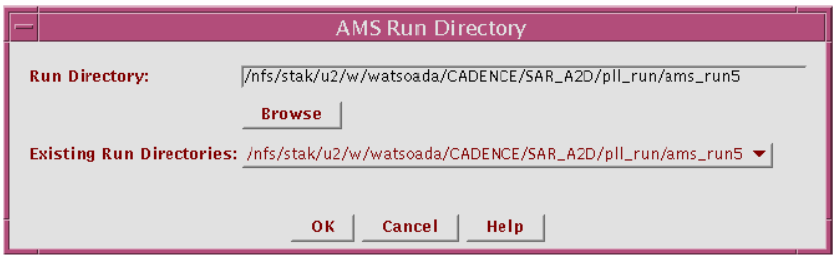
Une fois cette première étape de configuration passée, `Hierarchy-Editor` se

lance:



A toujours tenir à l'oeil!!!
Les messages ne s'affichent
plus dans la CIW...

Dans cette fenêtre, la première chose à faire est de sauver la configuration par File ¿ Save. Puis cliquer sur l'onglet Plug-Ins et choisir AMS. De cette manière l'outil est configurer en mode mixte, ce que l'on verra avec l'apparition d'un 4^{ieme} onglet à coté de View. Dans le menu AMS, choisir la seule option valable Run Directory....

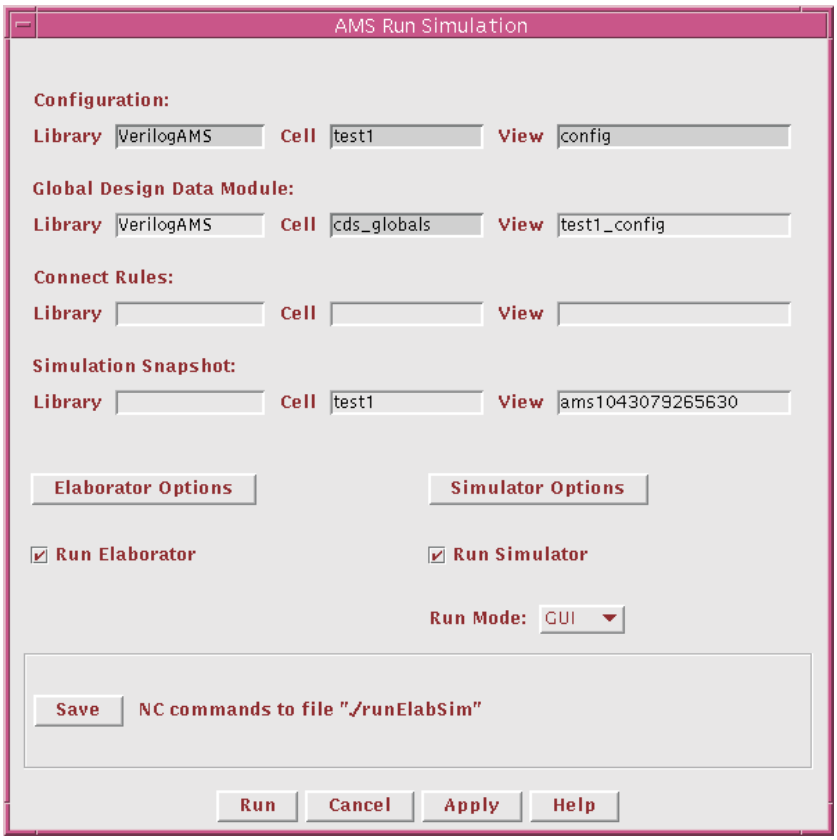


A l'aide de cette fenêtre, il est possible de régler le répertoire de travail dans lequel seront stockés les données et courbes de simulation. On notera qu'à chaque nouvelle simulation, Cadence crée un nouveau répertoire en indentant le nom.

L'étape suivante consiste à configurer le simulateur[2] Pour cela, dans la barre de

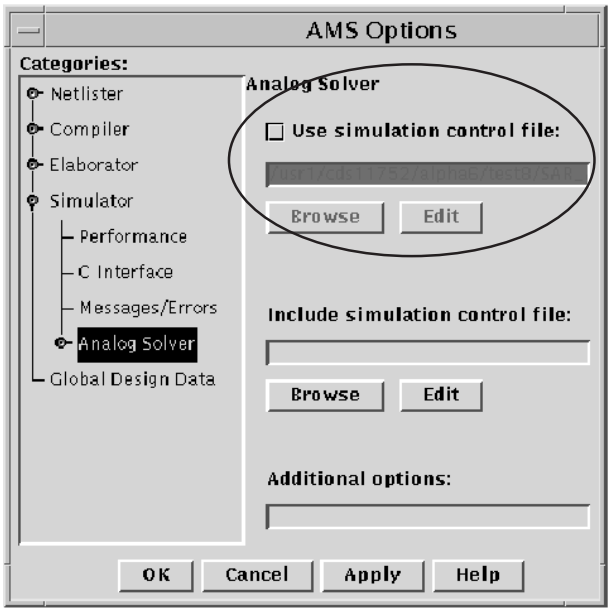
2. La description donnée ici est très sommaire et permet une première prise en main. Le lecteur intéressé se reportera aux 600 pages de l'Open-Book...

menu AMS, cliquer sur l'option AMS Run Simulation. La fenêtre suivante apparaît:



Prendre soin d'effacer l'item `mixedsignal` dans la zone Connect Rules. Il semble qu'il y ait un petit bug quand on n'a pas besoin de règles de connexion...

Cliquer sur l'onglet Simulator Options, puis aller dans la sous rubrique Analog Solver:



et spécifier les caractéristiques de la simulation analogique par un fichier du type:

```
simulator lang=spectre
saveNodes options save=all rawfmt=sst2 rawfile="sch.tran"
tran1 tran stop=100ns errpreset=conservative maxiters=10 cmin=0f
```

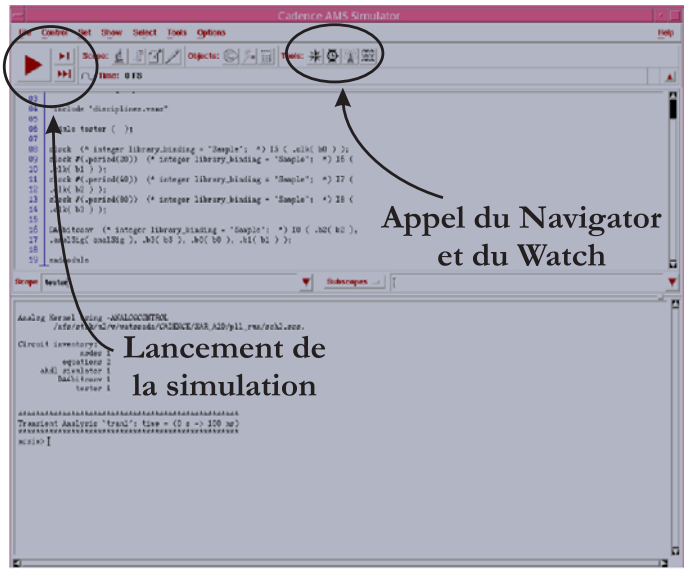
Dans ce fichier (à l'extension *.scs), on notera entre autre l'option `stop=100ns` qui permet de régler la durée de simulation à 100ns. Une fois le fichier sauvé faire apply pour retourner à la fenêtre de simulation puis faire run. Une fois de retour sur le `hierarchy-editor`, retourner dans la barre de menu AMS . Selectionner l'onglet Design Prep.. ce qui fait apparaître la fenêtre suivante:

Cette fenêtre spécifie la type de "netlistage" et de compilation. Elle répond à la question: "A chaque modification, faut refaire une netlist et une simulation complètes ou bien faut-il remettre à jour que ce qui a été modifié?".

Une fois, les choix validés on cliquera sur run, ce qui vérifiera la netlist du schéma:

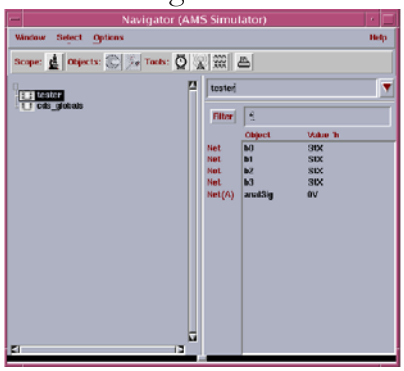
Une fois la préparation du design effectué et sans erreur ni warning, il convient alors de lancer le simulateur. Pour cela, retourner dans la barre de menu AMS, cliquer sur

l'option AMS Run Simulation pour faire apparaître la fenêtre du simulateur:



Comme indiqué sur la figure, il est alors possible de lancer la simulation avec le gros bouton ▶. Afin de faciliter la visualisation des waves, on peux aussi appeler le Navigator et le Watch.

Navigator



Watch

Dans le Navigator, on sélectionnera les signaux à visualiser puis on lancera la simulation. Une fois terminer, il suffit d'invoquer le signal Scan Waveform par  . Ce dernier ouvre alors la fenêtre avec les différents signaux analogiques et digitaux: