Ипе, Ипопе, Month.

Ипеот, Ипопе, Mônth.

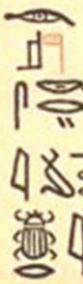
Ипо, « Mnévis.

Скр.сбр, « Sakr, Sokar.

Soit les n

2° Le

tiale du mot  qui répond au copte ⲛⲟⲩⲧⲉ, *Dieu*. Ce caractère est remplacé par le signe  dans les textes hiératiques :



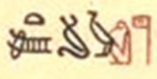
оусрн, оусірє, Osiris.

сбрі, « Sakri, Sochari.

амсе, « Amsèth.

ерє, « Thré, Thoré.

Souvent même les noms propres des dieux se trouvent déterminés dans les textes hiéroglyphiques par ces deux caractères à la fois, ce qui constitue alors un déterminatif tropico-figuratif ; exemples :



тмоу.емоу, *Thmou, Athmou.*

сѣб.сѣк, Sévék.

анпоу, анотп, Anubis.

Il importe d'observer que le caractère déterminatif figuratif de

Lire & Comprendre VHDL & AMS

ISBN 978-1-4092-2787-8
Imprimé en Espagne
Lulu éditeur

Illustration de couverture:
Grammaire égyptienne ou principes généraux de l'écriture sacrée égyptienne, ch. V p 110
Jean François Champollion, édition 1836.

Table des matières

1 Clés de ce manuel

1.1 Il y a, il n'y a pas

On trouvera dans ce manuel toutes les informations nécessaires à la compréhension de VHDL et de VHDL-AMS en tant que langages ayant une sémantique de simulation ou de synthèse. Pour garder une longueur raisonnable à ce document, on n'y duplique que rarement l'information et il y est fait un usage systématique et général des renvois sous la forme paragraphe/page.

1.1.1 Les exemples

Les exemples sont là uniquement pour éclairer la syntaxe ou une méthode. Ce manuel n'est pas un recueil de modèles, chose qu'on trouvera par ailleurs facilement et en particulier sur Internet.
Voir particulièrement http://tams-www.informatik.uni-hamburg.de/vhdl/ , cf bibliographie [HAM] chapitre 19 page 169.

1.1.2 Les techniques de modélisation

Ce manuel s'intéresse au langage et essaye de ne rien oublier. Pour les techniques de modélisation, qui transcendent d'ailleurs les langages utilisés, on trouvera une variété de livres et de tutoriaux, voir la bibliographie, et le livre compagnon de celui-ci «*Écrire & Comprendre VHDL & AMS* » (bibliographie [ROU1] chapitre 19 page 169) pour ce qui est des différents styles utilisés en VHDL.

1.1.3 La barre à droite

ATTENTION barre à droite: Comme la connaissance d'un langage implique de savoir lire des choses qu'on n'écrirait peut-être jamais, et en particulier d'anciens modèles ou des modèles écrits par des gens qui utilisent toutes les finesses du langage, ce manuel essaye d'être complet ; chaque fois que les constructions exposées sont très peu usitées (et que le débutant peut s'en passer dans un premier temps, voir ci-dessous 1.4), les paragraphes correspondants seront marqués d'une barre à droite, comme ce paragraphe-ci.

1.2 La BNF

La grammaire dans le LRM est notée dans un formalisme très simple (BNF). Ce manuel donne les syntaxes BNF de différentes constructions, sans toutefois descendre aux niveaux les plus élémentaires de la grammaire. La BNF complète est en annexe.

- ::= signifie « est défini par »
- | est le « ou ».
- {…} signifient « répétition zéro fois ou plus ».
- […] signifient « zéro fois ou une fois » (optionnel donc).

Les mots-clés seront en gras, les règles en casse normale. Tous les symboles autres que ceux définis supra sont « dans le langage ». Il arrive –rarement- que des éléments du langage entrent en collision avec ces métasymboles. Dans ce cas ils seront mis en gras et marqués par une note (voir note 22 page 160).

1.3 [AMS] VHDL analogique et mixte

Les éléments du langage qui concernent VHDL-AMS sont signalés par une balise dans les titres des paragraphes [AMS] et par une fonte spécifique en italique, comme ce paragraphe-ci. Il s'agit des éléments qui concernent seulement *AMS. Un modèle purement analogique aura besoin de quantité de concepts communs aux deux mondes digital et analogique, par exemple la notion de composant. La supposition forte est que quelqu'un qui s'intéresse à VHDL-AMS s'intéresse toujours à VHDL.*

1.4 L'usuel et l'inusité

VHDL a déjà une longue vie et il se trouve que certaines de ses constructions sont rarement utilisées. À cela plusieurs raisons :

- VHDL fut un langage de comité, avec une inflation de propriétés due à l'enthousiasme des ingénieurs membres de l'IEEE travaillant hors contraintes, autour d'une table. Par exemple la possibilité d'écrire des valeurs « à dimensions » (§5.2.4 page 34) n'a sans doute jamais été utilisée vraiment que pour le type TIME, et seulement dans le paquetage standard : §16.1.1 page 115.
- La grande généralité de VHDL s'est avérée être contre-productive dans plusieurs domaines. Ainsi la possibilité de définir soi-même ses types logiques et les fonctions de résolutions (possibilité inédite et qui a exalté bien des concepteurs, voir §7.5.2 page 58) a conduit, dans les années 90, à une profusion de paquetages incompatibles entre eux rendant l'interopérabilité des modèles difficile voire impossible (à 3, 7, 9 et jusqu'à 46 états logiques !). Un comble pour un standard ; cela fut résolu par l'arrivée du paquetage STD_LOGIC_1164 (§16.1.3 page 118), lequel a rendu anecdotique l'écriture par le concepteur de types logiques et de résolutions, ainsi que l'utilisation des signaux et blocs gardés (§11.8.1 page 91). Et accessoirement a rendu aussi inutile toute une partie du langage ayant trait aux conversions de types lors des associations et passages d'arguments (fin du §10.3 page 76).
- L'héritage d'ADA, qui a rendu le langage extrêmement carré, a donné aussi quelques scories. Par exemple la grande complexité des règles de surcharge: §6.2.5 page 49 et §12.7.3 page 104.

2 Lexique

Ce chapitre traite des mots du langage, c'est-à-dire des éléments lexicaux séparés par des blancs, des retours à la ligne ou d'autres éléments lexicaux.

2.1 Blancs, RC, etc.

VHDL est un langage «*form-free*», indépendant de la mise en page. Les mots du lexique sont séparés par des *séparateurs*, et en règle générale, partout où est attendu un séparateur, on peut en mettre plusieurs. Les séparateurs sont le blanc, le blanc insécable, la tabulation, le retour à la ligne, et ne sont pas nécessaires quand il y a un *délimiteur* (§2.5 ci-dessous).
Les quelques exceptions sont généralement naturelles: dans les chaînes de caractères, tous les caractères sont significatifs et on ne peut pas insérer de retour à la ligne. Les commentaires qui se terminent avec la ligne –voir §2.3 ci-dessous- ne supportent donc pas d'en contenir.
Attention: il faut une espace entre l'entier et son unité dans le cas des types physiques (§5.2.4 page 34). Mais son omission n'entraîne en général qu'un *warning* à la compilation, ce qui peut toutefois être gênant s'il y en a des centaines.

2.2 Casse

VHDL ignore la casse (majuscule/minuscule) sauf en deux circonstances: les caractères et leurs chaînes, et les identificateurs étendus. Par exemple, `Toto, TOTO, ToTO` sont trois instances du même identificateur. "AB" et "ab" sont deux chaînes de caractères différentes (mais x"ab" et X"AB" deux chaînes de bits identiques, voir §2.7.7 page 15).

2.3 Commentaires

Comme en Ada, le commentaire commence par deux tirets et finit avec la ligne.

```
S<='0' ; -- ceci est en commentaire
```

Il n'y a pas de commentaires «parenthésé» comme en Pascal (*…*) ou C /*…*/. Néanmoins, en cas d'urgence et pour mettre en commentaire un grand bloc de code dans une architecture, on peut jouer avec l'instruction **`if`**…**`generate`**, voir §14.2 page 112. Le programmeur C reconnaîtra là une utilisation courante et perverse du #if 0…#endif.

```
Commentaire: if FALSE generate
… ici bloc de code qui disparaîtra avant simulation
… mais qui sera compilé  (erreurs signalées)
end generate;
```

2.4 Mots-clés

Les mots-clés sont réservés, c'est-à-dire qu'on ne peut pas s'en servir comme identificateurs (sauf les identificateurs étendus, § 2.6.2 page 13). Ils seront écrits en gras dans ce document. Certains mots-clés comme **range** ou **across** se trouvent être aussi le nom d'attributs, et à ce titre ils peuvent apparaître en tant que mot clé et en tant qu'identificateur, selon l'endroit.

abs	**disconnect**	**is**	**out**	**sla**
access	**downto**	**label**	**package**	**sll**
after	**else**	**library**	**port**	**sra**
alias	**elsif**	**linkage**	**postponed**	**srl**
all	**end**	**literal**	**procedure**	**subtype**
and	**entity**	**loop**	**process**	**then**
architecture	**exit**	**map**	**protected**	**to**
array	**file**	**mod**	**pure**	**transport**
assert	**for**	**nand**	**range**	**type**
attribute	**function**	**new**	**record**	**unaffected**
begin	**generate**	**next**	**register**	**units**
block	**generic**	**nor**	**reject**	**until**
body	**group**	**not**	**return**	**use**
buffer	**guarded**	**null**	**rol**	**variable**
bus	**if**	**of**	**ror**	**wait**
case	**impure**	**on**	**select**	**when**
component	**in**	**open**	**severity**	**while**
configuration	**inertial**	**or**	**signal**	**with**
constant	**inout**	**others**	**shared**	**xnor**
				xor
AMS				
across	***nature***	***quantity***	***subnature***	***through***
break	***noise***	***reference***	***terminal***	***tolerance***
limit	***procedural***	***spectrum***		

2.5 Délimiteurs

Les délimiteurs sont tous les symboles non alphabétiques qui rythment la syntaxe: virgules, parenthèses, plus… À noter que certains d'eux sont surchargeables (les opérateurs) et qu'on peut définir ou redéfinir les primitives appelées quand ils apparaissent.

& ' ( ) * + , - . / : ; < = > | [ ] => ** := /= >= <= <> et *[AMS]* **==**

Auxquels nous ajoutons ici la double apostrophe " et le dièze **#** qui ne sont pas des délimiteurs *stricto sensu* car ils font toujours partie d'éléments lexicaux identifiés par ailleurs.
Quelques substitutions possibles et utilisées très rarement:

- Le ! (point d'exclamation) peut être utilisé en place de | (barre verticale)
- Le : (deux points) peut remplacer le # (dièse) dans les littéraux basés, s'il le remplace aux deux occurrences.
- Le % (pour cent) peut remplacer la double apostrophe (") si il la remplace aux deux occurrences et que la chaîne de caractères ne contient pas de ".

2.6 Identificateurs

Les identificateurs sont les noms qui sont choisis par le concepteur pour les différents objets du langage: noms d'entités, de signaux, de variables, etc.

2.6.1 Identificateurs simples

Les identificateurs simples (l'immense majorité des identificateurs utilisés) ne comprennent que les caractères alphabétiques (A à Z et a à z avec ou sans accents, sans distinguer majuscule et minuscule), les chiffres (0 à 9) et des blancs-soulignés (*underscore*, le '_').
La norme autorise les caractères accentués, majuscules et minuscules, de la norme ASCII.

Limitations:

- Pour de simples raisons de lisibilité, le blanc-souligné ne doit pas apparaître au début, ni à la fin, ni à côté d'une autre blanc-souligné. Deux identificateurs qui ne diffèrent que par lui sont différents (à la différence d'autres éléments lexicaux, voir § 2.7.1 page 14 par exemple).
- On ne peut pas commencer par un chiffre.
- On ne peut pas utiliser un mot-clé.

Exemples: `TOTO, NabUCHodoNoSor, X123, A_B_C, Ääöù`
Contre-exemples: **`buffer (mot-clé)`**`, 2AB, A$, X__Y, Z_`

2.6.2 Identificateurs étendus

Il arrive que les limitations sur la syntaxe des identificateurs simples soient gênantes dans certains cas, en particulier quand le modèle que l'on écrit correspond à quelque objet déjà écrit dans un autre langage ou venant d'une bibliothèque qui n'a pas les mêmes conventions de nommage, et qu'on voudrait bien « y coller ». Pour ces cas très particuliers –et rares-, il est possible de s'affranchir de toutes les limitations en encadrant l'identificateur de deux barres obliques inversées (*backslash*). Si l'on veut mettre une barre oblique inversée dans l'identificateur, il suffit de la doubler. Dans ce cadre,

- Les majuscules et minuscules sont différenciées.
- Les blancs et blancs insécables font partie de l'identificateur
- Tout caractère «graphique» au sens ASCII peut être mis dans l'identificateur (pas le retour à la ligne donc)
- On peut utiliser des mots réservés, par exemple \buffer\.
- Un identificateur étendu est toujours distinct d'un identificateur simple, même si le contenu est le même: ainsi VHDL, \VHDL\ et \vhdl\ sont 3 identificateurs distincts.

Exemples: \group\ , \123\, \a\\b\

2.7 Littéraux

Un littéral est un mot du langage qui porte une valeur. 23 par exemple est un mot du langage qui porte la valeur 23. Attention: les littéraux sont attachés à des «genres de types» mais pas à un type particulier. 23 peut être une valeur correcte pour plusieurs types entiers qui sont incompatibles entre eux. '0' peut être une valeur littérale du type BIT (la première) ou du type CHARACTER (la 48ème). Les «genres de types» sont :

- Genre entier: tous les types définis par
  `type` `T` **`is range`** *`littéral_entier`* **`[down]to`** *`littéral_entier;`*
  Attention: même si le type est défini de 0 à 10, l'apparition de la valeur littérale 11 ne servira pas à lever les ambiguïtés en cas de surcharge ; voir §6.2.5 page 49.
- Genre réel: tous les types définis par
  `type` `T` **`is range`** *`littéral_réel`* **`[down]to`** *`littéral_réel;`*
  Même remarque que ci-dessus.
- Genre caractère: tous les types énumérés (§5.2.2.1 page 32) contenant au moins un caractère littéral dans leur énumération. Par exemple **`type`** `T` **`is`** `(TOTO, 'X', TITI);` . Attention: dans ce cas le littéral 'A' n'appartient pas au type, mais cela ne sera pas utilisé pour lever des ambiguïtés à la compilation s'il faut décider entre CHARACTER et T pour résoudre la surcharge à l'appel d'une procédure, par exemple.
- Genre chaîne de caractères: les vecteurs (tableaux à une dimension) d'objets d'un type de genre caractère. Ainsi dans le paquetage STANDARD les types STRING et BIT_VECTOR sont deux types incompatibles mais tous deux de genre chaîne de caractères puisque vecteurs d'un type caractère.

- Genre énuméré: les types énumérés (§5.2.2.1 page 32) définissent des valeurs symboliques qui deviennent littéraux *ipso facto*. Comme pour les autres genres, un tel identificateur peut appartenir à plusieurs types énumérés.

2.7.1 Entiers littéraux

Un entier littéral est défini par une succession de chiffres et de blancs-soulignés (*underscore*, '_'). Les blancs-soulignés n'ont aucun impact sur la valeur de l'entier littéral, ils sont là juste pour permettre une aération des nombres «longs».
On peut utiliser la notation «10 puissance» avec un E et un exposant. 2E4 est $2x10^4$ et est identique à 20000. Évidemment l'exposant ne peut être ni négatif ni fractionnaire.
Le blanc-souligné ne peut pas apparaître au début, ni à la fin, ni à côté d'un autre blanc_souligné.
Exemples: 23, 1_000_000, 34E3

2.7.2 Réels littéraux

Un réel littéral est défini par une partie entière, un point décimal séparateur, une partie fractionnaire, et optionnellement la lettre E, un signe optionnel et une puissance de 10.
Comme d'habitude pour les littéraux, les blancs-soulignés n'ont aucun impact sur la valeur du réel littéral, ils sont là juste pour permettre une aération des nombres «longs».
Le blanc-souligné ne peut pas apparaître au début, ni à la fin, ni à côté d'un autre blanc_souligné, ni à côté du point décimal, ni dans l'exposant.
Exemples: 2.3, 1_000.000, 3.4E-3

2.7.3 Basés

On peut écrire les littéraux entiers et réels dans n'importe quelle base de 2 à 16. À noter que le dièse ici utilisé peut être remplacé, s'il n'est pas sur le clavier, par le deux-points «:».

2.7.3.1 Entiers littéraux basés

On écrit la base, un dièse, le nombre dans la base, puis un autre dièse pour terminer la partie «en base». Ensuite, optionnellement, un exposant écrit en base 10 qui représente la puissance de la base par laquelle il faut multiplier l'ensemble.
Exemples:
16#12AB# : en base 16, le nombre 12AB
2#0010#E3 : en base 2, 0010 multiplié par 2^3.

2.7.3.2 Réels littéraux basés

On écrit la base, un dièse, le nombre dans la base avec le point (qui n'est plus forcément «décimal»), un autre dièse qui termine la partie «en base». Ensuite, optionnellement, un exposant éventuellement négatif écrit en base 10 qui représente la puissance de la base par laquelle il faut multiplier l'ensemble.
Exemple:
16#1A.34#E3: donne en base 10 $(1x16 +10 + 3x16^{-1} + 4x16^{-2}) \; x16^3$

2.7.4 Énumérés littéraux

Un type énuméré définit des identificateurs qui sont des valeurs symboliques du type (§5.2.2.1 page 32). Dès sa déclaration et dans sa zone de visibilité, ces identificateurs deviennent des littéraux de ce type. Ils peuvent être écrits suivant la notation «identificateur étendu» (§2.6.2 page 13) ou encore être aussi des caractères littéraux (§2.7.5 ci-dessous).

```
type couleur is (bleu, blanc, rouge);

-- ici bleu est un littéral du type couleur.
```

2.7.5 Caractères littéraux

Un caractère littéral est un caractère entre deux simples apostrophes. Rappelons qu'un caractère littéral n'est pas forcément du type CHARACTER, mais peut être assigné à n'importe quel type de *genre caractère*, voir §2.7 ci-dessus – le type BIT est de genre caractère par exemple. Les majuscules et minuscules sont différenciées. Tous les caractères littéraux appartiennent à des types énumérés et sont de ce fait également des énumérés littéraux.

Exemples: 'A', 'x', '1'.

2.7.6 Chaînes de caractères littérales

Une chaîne de caractères est une séquence de caractères encadrée par deux doubles apostrophes ("). On peut remplacer la double apostrophe par le pour-cent (%) mais ce n'est pas vraiment conseillé, ni l'usage. [1] Ce trait va disparaître lors d'une prochaine révision du langage.
Rappelons qu'une chaîne de caractères littérale n'est pas forcément du type STRING, mais peut être assignée à n'importe quel type de *genre chaîne de caractères*, voir §2.7. Par exemple, le type BIT_VECTOR est de ce genre.
Si l'on désire mettre une double apostrophe dans la chaîne, il faut la doubler.
Les majuscules et minuscules sont différenciées. Les blancs-soulignés éventuels font partie de la chaîne.
Exemples: "toto", "abc""def", %abc%, "0010010". "V_H_D_L"

2.7.7 Chaînes de bits littérales

Lorsque la chaîne de caractères ne contient que des nombres de 0 à 9, des lettres de A à F et des blancs soulignés, il est possible de l'interpréter comme une séquence de bits exprimés en base 2, 8 ou 16, en la préfixant par B (binaire), O(octal), ou X (heXadécimal). Dans ce cas le compilateur va simplement traduire en chaîne de caractères littérale en faisant la substitution attendue et les erreurs éventuelles (inadéquation entre les digits et la base). La longueur de l'objet, si on la demande ensuite, se comprend après la transformation, c'est-à-dire que `"001_001"'LENGTH` peut valoir 6 si c'est une chaîne de bits (équivalente à `B"001_001"'LENGTH` donc à `"001001")` et 7 si c'est une chaîne de caractères. Telle qu'elle, l'expression ne compilera pas ; dans ce cas, il faudrait qualifier avec le type pour klever l'ambiguïté, VHDL ne prend pas d'option par défaut (voir §5.1 page 31).
Exemples

```
B"001_001" devient "001001"
O"123" devient "001010011"
X"A_B" devient "10101011"
```

`"001_001" devient "001001"` (le préfixe est B par défaut) si l'expression attendue est BIT_VECTOR de façon non ambigüe.

[1] D'ailleurs si on le fait, l'usage du " devient interdit dans la chaîne, pour bien marquer que ce n'est qu'un palliatif pour des claviers improbables n'ayant pas le ".

3 Bibliothèques, Entités, Architectures, etc.

La chose compilable s'appelle une unité de conception (*design unit*). Il en existe de 5 sortes: les déclarations d'entités §3.2.1, les corps d'architecture §3.2.2, les déclarations de configuration §3.2.3 et les paquetages, déclaration §3.3.1 et corps §3.3.2.

VHDL en tant que langage ignore absolument la notion de fichier ; les compilateurs (de tous langages) par contre ont l'habitude de compiler des fichiers. On peut donc mettre toutes les unités dans le même fichier, ou mettre une unité par fichier (mais pas moins). Le tout est de les compiler dans le bon ordre. Attention: certaines implémentations font, elles, de fortes suppositions sur les noms de fichiers et leur relation avec les unités contenues, voir la documentation de votre système.
Dans la «vraie vie» néanmoins, on sera quasiment obligé de mettre une unité par fichier pour minimiser les recompilations: en effet les règles de dépendances font que l'on est obligé de recompiler tout ce qui dépend d'une unité recompilée. Si cela n'a pas d'importance dans le cadre d'un exercice, cela peut être dramatique si l'on oblige à faire recompiler sans raison 100000 lignes de code dont certaines aux antipodes, simplement parce que pour corriger un bug dans une architecture on a recompilé sans raison l'entité du même fichier dont tous les collègues dépendent.

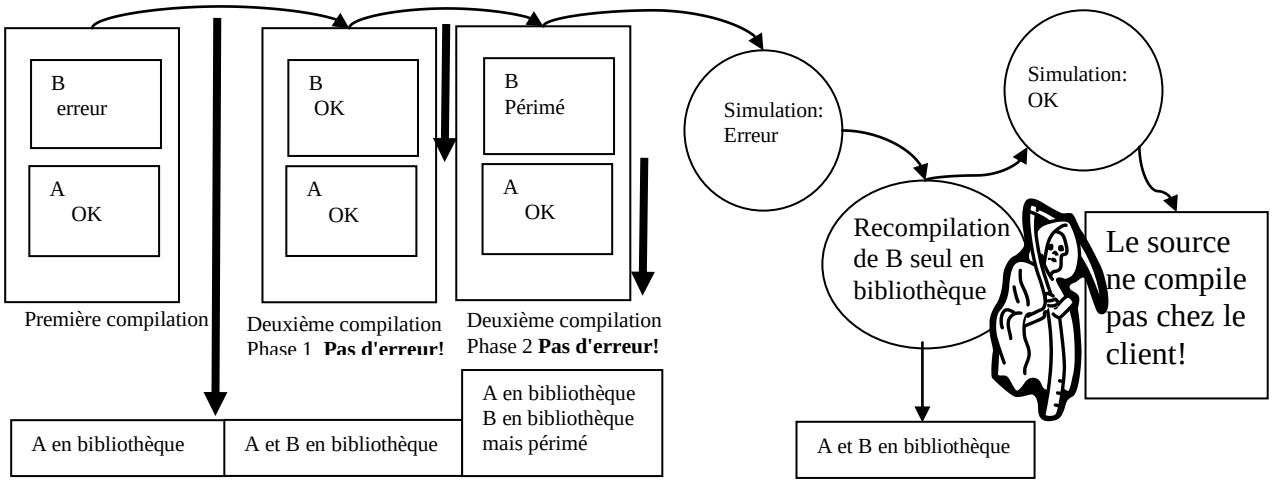


Figure 1 Ennuis dus à l'utilisation d'un seul fichier pour plusieurs unités

Un autre effet pervers de la facilité « tout dans un fichier » illustré par la figure ci-dessus est que si l'on se trompe dans l'ordre des unités (B dépend de A mais se trouve avant dans le même fichier), on aura une belle erreur à la première compilation, mais pas à la seconde –bien des concepteurs essayent une deuxième fois au cas où ☺- : A ayant été compilé après une erreur sur B, B pourra donc être recompilé ensuite puis A le sera de nouveau – on n'a pas le

choix, c'est dans le même fichier- rendant de ce fait B compilé mais périmé ce qu'on ne verra qu'à la simulation.
On sera alors tenté de pousser le bouton «recompiler tout ce qui est périmé» qui va probablement utiliser des copies locales des unités et compilera seulement la copie de B. Et cela marchera…jusqu'au moment de la recette chez le client où ça ne compilera pas; et qui au moment de faire le chèque ne comprendra probablement pas pourquoi il faut compiler deux fois puis appuyer encore sur «recompiler» avant de simuler.
Cet exemple simple ne suppose que 2 unités, le parcours du combattant peut être beaucoup plus long dans un cas réel.

3.1 Visibilités et rythmes du langage; library, use, alias

VHDL est un langage *à la* Ada: un objet déclaré est visible directement dans une certaine région (la sienne) et visible indirectement (par une notation pointée du genre *work.paquet.objet.champ)* dans une région plus large; il est complètement invisible de certaines régions.
Une région est une zone de VHDL qui commence par certains mots-clés (**entity, architecture, package, configuration, procedure, function, block, generate, loop**) et qui se termine par le **end** correspondant.
Attention à la syntaxe: La fin d'une structure se marque par un **end**, suivi *obligatoirement ou optionnellement,* selon les cas, du rappel de ce qu'était l'objet, suivi *optionnellement* du nom de l'objet:
`end; end entity; end BLA; end entity BLA;` sont tous des marqueurs de fin légaux.
Parfois le **end** tout seul suffit: pour les entités, architectures, packages, sous-programmes...
Souvent il ne suffit pas: **`end process, end component, end protected body, end record, end if, end case, end loop.`** Si on ne veut pas s'encombrer l'esprit avec les exceptions, toujours mettre au moins le rappel de la nature de l'objet: le nom de l'objet est, lui, toujours optionnel.
La règle de base est qu'un objet déclaré est visible «en dessous» dans sa propre région. Si une région plus interne est ouverte, un autre objet de même nom peut masquer sa visibilité directe mais pas sa visibilité indirecte.

```
architecture A of E is
signal S : BIT;
begin
  -- ici S est A.S, A.B.S n'est pas visible
  B:block
     signal S: BIT;
  begin
     -- ici S est A.B.S; A.S est visible uniquement sous ce nom.
  end block B;
  -- ici S est A.S; A.B.S n'est pas visible
end architecture A;
```

Cette règle explique la façon *à la* Ada de déclarer les types récursifs: voir §5.4 page 38: on ne référence jamais que des objets déclarés au dessus, fût-ce incomplètement.

Pour rendre visible tout le contenu d'une bibliothèque, il faut préfixer l'unité que l'on compile par :

```
library LIB;
```

Toutes les unités de la bibliothèque deviennent visibles dans toute l'unité courante et, s'il s'agit d'une entité ou d'une déclaration de paquetage, dans ses corps d'architectures ou de paquetage. Leur contenu devient accessible s'il est public. Toutefois la notation pointée (*LIB.PACK.OBJET*) est nécessaire. En particulier, si le paquetage contient des surcharges

d'opérateurs (voir les opérateurs §6.2 page 46 et la surcharge §6.2.5 page 49), ils ne sont accessibles que sous la forme LIB.PACK."+"(A,B) et non par A+B.
Pour rendre visible directement ce qui est déjà visible par notation pointée, on peut dire
`use LIB.PACK.OBJET;` -- rend OBJET directement visible sous son nom simple
`use LIB.PACK.all;` -- pareil pour toutes les déclarations de PACK
La deuxième forme est bien plus utilisée. Cette clause **use**, contrairement à la clause **library** qui doit être en tête d'unité, peut être mise partout dans une zone déclarative. Son effet sera «en dessous».

Une autre façon de changer les visibilités est d'utiliser les alias. Un alias donne un autre nom (local à la région et qui épouse ses règles de visibilité) à un objet déjà visible même indirectement. Cela peut s'avérer très utile quand le même objet (un bus par exemple) doit être tantôt vu comme un entier de 32 bits, et tantôt comme un code opératoire dont certains champs ont des significations logiques n'ayant rien à voir avec des entiers.

```
signal bus32: BIT_VECTOR(31 downto 0);
alias codeop: BIT_VECTOR(0 to 4) is bus_32 (31 downto 27);
```

On voit sur cet exemple que l'alias peut tout chambouler: l'index, la direction des tableaux, etc. tant que l'on peut sans ambiguïté assigner chaque objet scalaire à son alias scalaire. Ici codeop(1) est le même objet que bus32(30).

Lorsqu'il y a ambiguïté pour cause de surcharge (voir §6.2.5 page 49), elle peut être levée par la définition d'un profil: il s'agit de l'énumération des types des arguments dans l'ordre de déclaration, suivie du type de la valeur retournée dans le cas des fonctions, le tout entre crochets []:

```
alias "or" is STD.STANDARD."or" [STD.STANDARD.BIT,
                                 STD.STANDARD.BIT
                                 return STD.STANDARD.BIT];
```

Et les ambiguïtés? À force de rendre visible directement des objets qui ont des noms étendus différents, on peut finir par rendre directement visibles deux objets de même nom «simple» et de même profil. Dans ce cas VHDL a une réponse simple également: pas d'option par défaut, pas de priorité, les deux objets deviennent invisibles directement mais restent visibles par leur nom complet. Cela peut créer des surprises quand on (pense qu'on n')ajoute (qu')une clause **use** à une unité qui compilait parfaitement, et qu'on n'a pas l'impression d'avoir fait grand mal.

Attention: les clauses de contexte ne «marchent» que pour l'unité courante et ses corps. S'il y a plusieurs unités dans un fichier (voir Figure 1 page 17 à ce sujet), penser à répéter les clauses de contexte devant chaque unité où elles sont nécessaires.

3.2 Entité de conception (design entity)

L'entité de conception est l'association d'une déclaration d'*entité* (vue externe, voir ci-dessous §3.2.1) et d'une *architecture* ou *corps d'architecture* (voir ci-dessous §3.2.2) de cette entité. C'est un ensemble simulable pour autant que toutes ses références à d'autres entités via les configurations (ci-dessous §3.2.3) soient également simulables.
Attention: par raccourci abusif, la majorité des concepteurs y compris l'auteur appellent «architecture» le «corps d'architecture» (*architecture body*), ce qui n'est pas grave; et, plus grave, «entité» (*entity*) la seule déclaration d'entité (*entity declaration*). Alors que le manuel

de référence utilise le mot *entity* pour désigner une association entre la déclaration et une des ses architectures.

3.2.1 Déclaration d'entité (entity declaration)

Il s'agit là de décrire la vue externe d'un objet physique. *A contrario*, il ne s'agit pas de dire ce que ça fait! Une déclaration d'entité a un nom:

```
entity BLA is
  [generic (…);]
  [port (…);]
  [declarations]
[begin
  instructions passives ]
end [entity] [BLA];
```

Dans une déclaration d'entité, on va trouver quatre zones:

- La généricité: les déclarations d'objets qui seront vus comme des constantes dans l'entité et ses architectures, mais dont les valeurs seront données plus tard: au moment de l'instanciation, ou au moment de la configuration, ou enfin au moment de l'exécution (simulation, synthèse). Voir 14.1 page 111.
- Les interfaces, les ports: ce sont les signaux ou des objets analogiques (terminaux, quantités) qui «traversent» le boîtier. Ce sont des signaux ou objets exactement comme les autres, avec *en plus* un mode (direction ou contrainte de buffer) quand c'est pertinent. Voir §10.4 page 78.
- Une zone de déclaration d'objets qui seront vus par l'entité et toutes les architectures associées. En sus des items purement déclaratifs (types) on peut y trouver des signaux, des objets analogiques ou des variables partagées, ainsi que des déclarations et des corps de sous-programmes.
- La zone d'instructions passives: c'est une zone où l'on peut mettre tout le code exécutable qu'on veut *à condition qu'il n'affecte pas de signaux*. Autrement dit c'est du code qui ne participe pas au fonctionnement du modèle, il ne peut être que comportemental (processus, appel concurrents de procédures sans signaux autrement qu'en mode **in**, **assert**) et sa seule action ne peut donc être que de faire des messages d'erreur ou d'avertissement (supervision des délais par exemple) sur console ou dans fichier.

3.2.2 Corps d'architecture (architecture body)

C'est l'endroit où l'on décrit le comportement, c'est là qu'on chasse le plus souvent le bug. Une architecture a un nom et fait référence au nom de son entité.

```
architecture TOTO of BLA is …
  [declarations]
begin
  instructions
end [architecture] [TOTO];
```

La première ligne associe TOTO à la déclaration d'entité BLA. Plus tard, on fera référence au couple dans le langage et dans les outils par la syntaxe BLA(TOTO). Il peut donc y avoir plusieurs architectures par entité, ce qui est une des forces de VHDL. Le corollaire est que, lorsqu'on simule une entité, ou lorsqu'on branche une entité sur un composant, il faut toujours se préoccuper de savoir quelle architecture va venir avec car c'est bien l'architecture qu'on traite (simulation, synthèse). Et pour le malheur du concepteur c'est le seul cas de figure où

VHDL prend une option par défaut en l'absence d'information: l'architecture par défaut est la dernière qui a été compilée.

On y trouve:

- une zone déclarative où les objets restent locaux: on ne les voit pas depuis l'extérieur, exception faite des composants qui sont visibles depuis les configurations (§3.2.3 ci-dessous)
- une zone d'instructions qui sont concurrentes: leur ordre n'a pas d'importance. Parmi ces instructions, l'une est le processus qui englobe, lui, des instructions séquentielles: l'ordre y a de l'importance.

3.2.3 Configuration

La configuration est une unité qui regroupe toutes les informations disant «quelle entité et quelle architecture va où» au moment de mettre ensemble toute la construction hiérarchique. La configuration s'applique à une entité et depuis cette entité on «descend» dans la structure pour dire quelles autres entités on branche où. Voir §10.8.3.4 page 83.

```
configuration C of E is
   [declarations]
   for A – une architecture qu'on ouvre pour configurer ses composants
     [clauses uses pour régler les visibilités]
     for composant1: COMP1 use LIB.X1(Y1) generic map(…) port map(…);
     …
     end for;
     for composant2: COMP2 use LIB.X2(Y2) generic map(…) port map(…);
      for Y2 – on descend dans Y2
       for composant_de_Y2:COMP3 use LIB2.Z(2) generic map(…) port map (…);
       …
       end for;
       …
      end for;
     …
     end for;
   end for;
end [configuration] [C];
```

3.3 Paquetage (package)

Un paquetage est un ensemble de ressources décrites dans une partie spécifications (la déclaration, §3.3.1 page 22) et une partie implémentation (le corps, §3.3.2 page 22). L'informaticien qui écrit en C peut imaginer le paquetage comme la réunion d'un *x.h* et d'un *x.c* où il n'y aurait pas de *main()*. Le paquetage peut avoir des fins très diverses: définir un type logique, rassembler des primitives mathématiques, ou d'entrées-sorties, ou des constantes utiles…

3.3.1 Déclaration de paquetage (*package declaration*)

```
package PACK is
 déclarations
end [package] [PACK];
```

La déclaration de paquetage peut contenir tous les items déclaratifs du langage, à l'exception des variables non partagées. On y trouve des types, des constantes, des signaux, des variables partagées, des déclarations de sous-programmes, des déclarations de composants, etc. La déclaration de paquetage est bien mieux définie par ce qu'on n'y trouve pas: des instructions! La déclaration de paquetage a un nom. Elle ne contient pas de **begin**, on ne saurait quoi mettre après, toute l'unité est déclarative.

3.3.2 Corps de paquetage (*package body*)

```
package body PACK is
 déclarations avec corps de sous-programmes et valeurs différées
end [package body] [PACK];
```

Le corps de paquetage va contenir tout ce qui est nécessaire au bon fonctionnement des objets dont on a fait la publicité dans la déclaration de paquetage. Contrairement au mariage polygame des entités avec les architectures, le paquetage est monogame: un seul corps de paquetage par déclaration, de ce fait le corps n'a pas de «deuxième» nom.
On y trouvera donc: des déclarations diverses, mais qui resteront locales au corps de paquetage, n'étant pas publiées, et donc pas de signaux ni de composants (dont on ne saurait que faire[2]). On y verra aussi des fonctions et procédures non publiées, et enfin le corps des sous-programmes publiés, ainsi que les valeurs des constantes différées (voir §4.2 page 23). Il n'y a pas non plus de **begin**.

[2] Les procédures peuvent certes affecter des signaux, mais seulement 1/s'ils leur sont passés en arguments, ce qui serait impossible si le signal est invisible de l'extérieur, ou 2/si le sous-programme est dans la zone déclarative d'un processus ce qui ne peut pas être le cas ici. Quant aux composants, ils ne peuvent pas être utilisés hors d'une architecture et doivent donc être visibles.

4 Boîtes à Valeurs : Signaux, Variables, *Quantités*, etc.

Ce chapitre va détailler tout ce qui concerne la déclaration et la manipulation d'objets « valués » c'est-à-dire porteurs de valeurs, en VHDL. Les valeurs ont des types (voir chapitre 5 page 31) ; elles sont déposées dans des containeurs que nous allons voir ici. On verra que VHDL est un langage extrêmement typé (héritage d'Ada) et qu'en sus des containers triviaux que sont les constantes et les fichiers, il y a deux containers qui participent à la simulation –signaux, quantités- et qui sont d'un genre très différent. Cela pourra surprendre l'habitué de l'informatique qui ne connaît que les variables.

4.1 Initialisations et valeurs par défaut

- Les constantes (**constant**) doivent avoir une valeur initiale, qui peut être donnée de façon différée. Voir §4.2 ci-dessous.
- Les signaux et les variables (**signal**, **variable**) peuvent avoir une valeur initiale explicite lors de leur déclaration :
  ```
  variable V: INTEGER :=3;
  ```
  Si on ne met pas d'initialisation explicite, les signaux et les variables sont initialisés par la valeur par défaut du type concerné : la plus à gauche du type s'il est scalaire (exemple: INTEGER'LEFT = -2^{31}), ou par la valeur composite par défaut construite en prenant transitivement les valeurs par défaut de chacun de ses sous-éléments. Si le type est un type accès, la valeur par défaut est **null**.
  ```
  variable V: INTEGER;
  ```
  équivalent à
  ```
  variable V: INTEGER := -2^31; -- si l'entier est sur 32 bits
  ```
- *[AMS] La quantité (**quantity**) est, elle, initialisée à 0.0 par défaut si on ne met pas d'initialisation explicite. Cette valeur n'est qu'une valeur proposée au solveur pour démarrer ses heuristiques et l'aider à converger, elle changera très probablement.*
- Comme on ne peut pas affecter des fichiers (**file**), la question de la valeur initiale ne se pose pas pour cette classe d'objet. Son statut est ouvert ou pas-ouvert suivant le type de déclaration.

4.2 Constantes

Une constante est un objet nommé d'un certain type, dont la valeur est donnée avant la simulation et qu'on ne peut pas changer pendant (voir le cas particulier de l'indice de boucle). On peut en distinguer quatre catégories :

- **Attributs** les constantes n'ont pas d'attributs (§6.2.6 page 50) en tant que telles, seulement les attributs provenant de leur type:

- Les constantes « simples » dont la valeur est donnée sur le champ :
  ```
  constant PI : REAL := 3.141592 ;
  ```

- Les constantes différées, seulement dans les paquetages, dont la valeur est donnée plus loin dans le source, en général dans le corps de paquetage alors que le nom est visible dans la déclaration de paquetage.

```
constant N : INTEGER ;
…
constant N : INTEGER :=25 ;
```

- Les constantes venant d'arguments génériques, déclarées dans l'entité et dont la valeur sera donnée lors de l'utilisation de cette entité (instanciation directe, configuration, ou simulation).

```
entity E is
generic (N : INTEGER) ;
…
for X : COMP use entity work.E(A) generic map(25) ;
```

- Les indices de boucles (**loop**, **generate**) qui sont vus comme des constantes à l'intérieur de la boucle (on ne peut pas les changer ni les passer à un sous-programme qui les changerait.)

```
for I in 1 to 10 loop
-- ici I a le statut d'une constante dont
-- la valeur ne peut changer que par passage dans le for
end loop ;
```

4.3 Signaux

Les signaux sont des animaux complexes et leur structure est détaillée dans un chapitre spécifique (voir chapitre 7 page 53). Ce sont les principaux vecteurs d'information pendant une simulation, le seul canal autorisé de messages entre processus. Disons simplement ici que les signaux peuvent être de n'importe quel type sauf les types accès (**access**) et les types fichiers (**file**). Et bien entendu on ne peut pas tricher en leur donnant un type composite dont un élément serait accès ou fichier. La raison de cette « limitation » est que toute la sémantique de simulation est basée sur la notion d'événement, c'est-à-dire « changement de valeur ». Pour cela il est nécessaire que la valeur du signal soit clairement définie et ne soit pas l'objet de multiples indirections ou pire, partagée avec d'autres objets.

Attributs (§6.2.6 page 50): Voir leur liste pour les signaux avec les détails chapitre 7 page 53.

```
S'DELAYED
S'DELAYED(T)
S'STABLE
S'STABLE(T)
S'QUIET
S'QUIET(T)
S'TRANSACTION
S'EVENT
S'ACTIVE
S'LAST_EVENT
S'LAST_ACTIVE
S'LAST_VALUE
S'DRIVING
S'DRIVING_VALUE
[AMS]
S'RAMP(…)
S'SLEW(…)
```

4.4 Variables

4.5 Variables en contexte séquentiel

- **Attributs** les variables n'ont pas d'attributs (§6.2.6 page 50) en tant que telles, seulement les attributs provenant de leur type.

La variable en VHDL a exactement la même sémantique qu'une variable d'un langage informatique : elle contient une et une seule valeur, cette valeur peut être de n'importe quel type contrairement aux signaux, et cette valeur peut être lue aussitôt qu'affectée, encore contrairement aux signaux. La variable peut être interprétée par un synthétiseur comme un

élément matériel, mais elle peut aussi être la partie d'un pur modèle comportemental, sans aucune implication matérielle.

En contexte séquentiel, la variable se déclare dans des processus et dans des sous-programmes :

```
process
variable V : INTEGER ;
begin
…
procedure P (arg:INTEGER) is
  variable V : INTEGER ;
begin
…
function F (arg: BIT) return BIT is
  variable V : INTEGER ;
begin
…
```

Elle s'affecte avec le symbole « := » :

```
V1 := V2;
```

Ce symbole spécifique et différent de l'affectation de signaux permet de distinguer les affectations variables/signaux au premier coup d'œil et sans aller chercher les déclarations. En effet la sémantique de l'affectation est si différente entre signaux et variables que l'inspection locale du code sans savoir la classe des objets ne serait d'aucun secours.

4.6 Variable partagées et moniteurs

Introduite avec VHDL'93, la variable partagée se déclare avec le mot-clé « **shared** » et dans un contexte concurrent, par exemple un paquetage ou une zone déclarative d'architecture (là où l'on déclare les signaux). Elle est donc visible et accessible depuis plusieurs processus, contrairement à la variable en contexte séquentiel qui dépend d'un et un seul processus.
Son utilisation casse une propriété essentielle de VHDL: la concurrence.
En effet, en écrivant dans une variable partagée depuis un processus, et en la lisant depuis un autre, on peut déterminer l'ordre d'exécution des processus, chose qui est impossible avec les signaux puisque la mise à jour de ces derniers ne se fait qu'une fois tous les processus arrêtés.

La déclaration se fait ainsi:

```
shared variable V : INTEGER;
```

ou

```
shared variable V : INTEGER := 2.0;
```

L'utilisation de tels objets est intéressante dans quelques cas de figures, comme les simulations stochastiques (création de nombres pseudo-aléatoires, observation de collisions plus ou moins aléatoires sur un bus) ou la collecte d'information pendant la simulation en vue d'une trace. En tout état de cause, on ne doit pas s'en servir si l'on prétend faire un modèle synthétisable ou ayant une sémantique matérielle.
Même quand leur utilisation est légitime, il peut rester le problème de la cohérence des données alors que plusieurs processus indépendants ont accès au même objet. Une structure introduite dans une version encore postérieure de VHDL (2002, voir [IE1] page 169) le **type protégé**, permet de garantir l'atomicité de l'accès à une variable partagée, c'est à dire que un processus **au plus** a accès à certains objets et certaines zones de code **à un instant donné**.

Voici par exemple un cas tiré de la documentation, permettant de gérer un compteur partagé par plusieurs processus, chacun d'eux ayant un accès exclusif quand il le modifie :

Déclaration et definition:

```
type CompteurPartagé is protected
                procedure incrementer (n: integer);
                procedure decrementer (n: integer);
                function valeur return integer;
end protected CompteurPartage;
        ....
type CompteurPartagé is protected body
    variable valeur_compteur: integer := 0;
    procedure incrementer (n: integer) is
     begin
       valeur_compteur:= valeur_compteur + n;
    end procedure increment;
    procedure decrementer (n: integer) is
     begin
       valeur_compteur:= valeur_compteur - n;
    end procedure decrement;
    function valeur return integer is
     begin
       return valeur_compteur;
     end procedure value;
 end protected body CompteurPartagé;
```

Utilisation:

```
shared variable Compteur : CompteurPartagé; -- ici la variable partagée

exemple: process is
                          ...
          Compteur.incrementer(5);
                          ...
          Compteur.decrementer(i);
                          ...
          v := Compteur.valeur;
          if (v = 0) then
             ...-- ici le compteur peut avoir changé
             ...-- car d'autres processus y ont accès
          end if;
end process exemple;
```

La révision 2001 de VHDL a apporté quelques modifications dans ce domaine: les compilateurs respectant cette révision (une minorité, pour ne pas dire aucun à cette date) doivent maintenant vérifier qu'une variable partagée est toujours d'un type protégé ; quant aux opérateurs (+, -, etc.) définis dans le type protégé, ils doivent être déclarés avec un argument de moins que nécessaire. L'argument manquant étant la variable elle-même.

4.7 Fichiers

- **Attributs** les fichiers n'ont pas d'attributs.

Le fichier est une structure de file associée à un élément de stockage externe, en général le disque dur.
L'habitué des langages informatiques connaît bien l'utilisation des fichiers, mais en VHDL il y a plusieurs traits spécifiques dont l'essentiel est qu'un fichier peut s'ouvrir automatiquement lors de sa déclaration, et il vaut mieux ne pas le fermer. Chose qui était d'ailleurs impossible avant VHDL'93. Ceci n'est pas une limitation, c'est le signe que les fichiers ne doivent pas

servir à communiquer entre processus. On ne doit pas, en principe, écrire dans un fichier, le fermer puis l'ouvrir depuis un autre processus. Si l'implémentation le permet-en fait c'est surtout du ressort du système d'exploitation- le résultat sera catastrophique au mieux (au moins a-t-on la trace du problème avant d'aller chez le client), imprévisible au pire.

Supposant la déclaration du type:

```
type MON_TYPE is file of INTEGER ; --voir §5.5 page 39.
```

La déclaration des objets peut être:

```
file F1: MON_TYPE; -- devra être explicitement ouvert avec FILE_OPEN
file F2: MON_TYPE is "bla.bla"; -- ouvre le fichier bla.bla en lecture
file F3: MON_TYPE open READ_MODE is "bla.bla"; -- pareil
file F4: MON_TYPE open WRITE_MODE is "bla.bla"; -- ouvre en écriture
```

Contrairement aux autres porteurs de valeurs, le fichier ne peut être associé dans les passages d'arguments et associations d'arguments qu'à des objets de type fichier (**file**). Pour l'utilisation et la déclaration des objets de ce type, on se reportera au paragraphe 5.5.

4.7.1 [AMS]Quantités

Les quantités sont toujours d'un type réel ou sous-type d'un type réel. Ce sous-type peut être contraint par une tolérance, qui indique au noyau de simulation la précision espérée lors de ses calculs et n'a pas d'autre signification dans le langage (voir 13.1 page 105).

*Elles représentent des variables au sens mathématique du terme (et donc pas au sens du mot-clé **variable** en VHDL). Leur valeur est déterminée par la résolution d'équations ou par quelque générateur: Le concepteur peut leur attribuer une valeur, mais elle ne sera que la valeur initiale utilisée par le solveur d'équations.*

*Elles contiennent les valeurs analogiques lors d'une simulation analogique ou mixte, et elles peuvent être déclarées «**across**» c'est-à-dire tensions pour le monde électrique, ou «**through**» c'est-à-dire courants dans le même monde, subissant ainsi les contraintes des équations prédéfinies par les lois de Kirchhoff. Elles peuvent être aussi libres ou des générateurs dans le domaine fréquentiel. Contrairement aux autres porteurs de valeurs, les quantités sont initialisées par défaut à 0.0. Hors domaine fréquentiel, les générateurs valent 0.0.*

4.7.1.1 [AMS]Quantités libres

Une quantité libre est, comme son nom l'indique, une quantité déclarée qui n'est liée

***Attributs** les quantités ont un jeu d'attributs (§6.2.6 page 50) en tant que telles, en sus des attributs provenant de leur type:*

- *Q'DOT: quantité du type de Q; dérivée par rapport au temps.*
- *Q'INTEG: quantité du type de Q; intégrale de 0 au temps courant.*
- *Q'DELAYED(T): quantité du type de Q; Q décalée de T.*
- *Q'ZOH(T[,initial_delay]) : quantité; (échantillonneur idéal de période T avec maintien). initial_delay: délai initial (0.0 par défaut).*
- *Q'LTF(num,den) : quantité; transformée de Laplace de Q. num et den sont REAL_VECTOR, les coefficients des polynômes).*
- *Q'ZTF(num,den,T[,initial_delay]): quantité; transformée en Z de Q. NUM et DEN sont REAL_VECTOR (coefficients des polynômes).*
- *Q'SLEW([SR[,SF]]) quantité du type de Q; sa valeur suit celle de Q mais avec une rampe définie par les pentes de montée et de descente donnés. Si SF omis, SF=SR. Si SF et SR sont omis, un break implicite est notifié au noyau.*
- *Q'TOLERANCE: string; nom du groupe de tolérance de Q*
- *Q'ABOVE(E): signal booléen; TRUE si Q > E, FALSE sinon.*

*par aucune équation implicite (au contraire des quantités de branches **through** et **across** (§4.7.1.3 ci-dessous).*
Une telle quantité se déclare dans une zone déclarative concurrente, comme les signaux.

```
quantity Q : REAL := 0.0;
```

4.7.1.2 [AMS]Quantités source

*Ce sont des quantités auxquelles on attache un générateur dans le domaine fréquentiel – le signal DOMAIN -§8.3.1page 63- est alors sur FREQUENCY_DOMAIN et la simulation n'est pas mixte. Il y en a de deux sortes: le spectre (**spectrum**) pour une analyse AC petits signaux, auquel il faut une magnitude et une phase, et le bruit (**noise**) auquel il faut une « puissance ». Les types des expressions doivent être compatibles avec celui de la quantité, aussi bien en VHDL, où l'on se ramène à des (sous)types de réels, que physiquement parlant (si l'on ajoute du bruit à un courant, ce bruit sera à calculer en ampères).*
Dans les déclarations de ces quantités et seulement là, la fonction prédéfinie FREQUENCY peut être appelée. Hors du domaine fréquentiel, les valeurs sont calées à 0.0.
Exemples:

```
quantity Q: REAL spectrum magnitude, phase;
quantity Flns: REAL noise K;
```

4.7.1.3 [AMS]Quantité de branche

*Elles sont attachées à deux terminaux ou jeux de terminaux si elles sont organisées en tableaux, et peuvent être **across** ou **through**. Quand un seul terminal est mentionné, l'autre est la référence de la nature à laquelle il appartient, voir §5.7page 41.*

```
quantity I1, I2 through T1;
--définit deux courants entre T1 et la référence.
quantity V1, V2 across T1;
--définit deux tensions entre T1 et la référence, qui sont donc
identiques.
quantity V across I through T1 to T2;
-- définit la tension et une branche de courant entre T1 et T2.
```

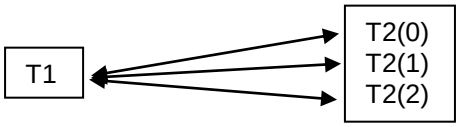


Figure 2 Quantité entre un terminal scalaire et un terminal vecteur
*Si un des deux terminaux est un vecteur, et pas l'autre, la définition des **across** et **through** créera des vecteurs de quantités, autant que d'éléments de vecteurs, allant de chaque élément du tableau vers le point unique de l'autre terminal.*

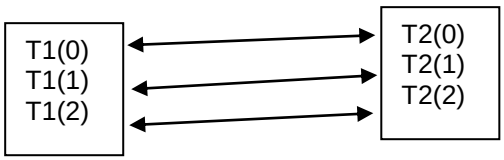


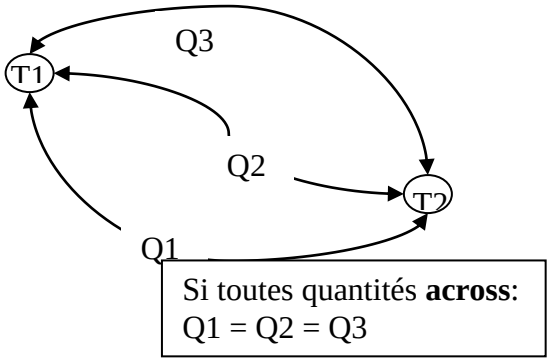
Figure 3 quantité entre deux terminaux vecteurs
*Si les deux terminaux sont des vecteurs (de même taille) alors tout se passe comme si on mettait chaque élément de vecteur «face à face» et on aura deux vecteurs de N **across** et N **through** ayant tous leurs deux éléments de terminal de même indice.*

***Attributs** les terminaux ont un jeu d'attributs (§6.2.6 page 50) en tant que tels, en sus des attributs provenant de leur nature:*

- *T'CONTRIBUTION: quantité **through**; somme des contributions des quantités de branche **through** incidentes au terminal T.*
- *T'REFERENCE: quantité **across**. Entre T et la référence de la nature de T.*
- *T'TOLERANCE: string; nom du groupe de tolerance de T.*

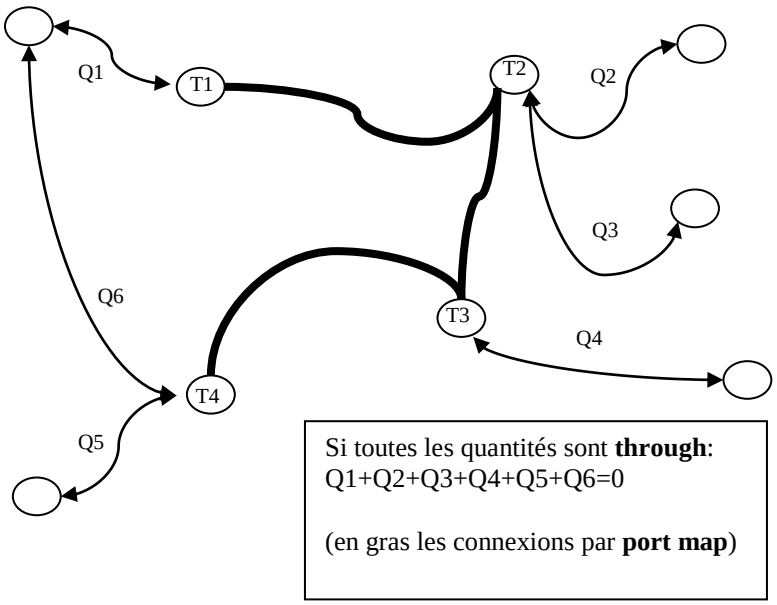
4.7.2 [AMS]Terminaux

*Bien que ce ne soient pas des «porteurs de valeurs», plutôt des transmetteurs, les terminaux sont les objets du monde analogique qui définissent les connexions, les nœuds. Les terminaux sont définis par leur **nature** (§5.7 page 41), qui représente le domaine physique dont on parle. La **nature** est l'association de deux grandeurs, représentant par exemple pour le monde électrique, la tension et le courant. Ou, pour le monde mécanique, la force et la vitesse. Ce sont les terminaux qui portent les lois de conservation: la somme des flux (**through**) qui traverse un jeu de terminaux interconnectés est nulle, l'effort (**across**) entre des terminaux interconnectés et une référence, est unique.*



*Ainsi si un terminal ne porte pas de valeur, par contre entre deux terminaux de même nature il y a une valeur et une seule: la quantité **across**. Déclarer plusieurs quantités **across** entre deux terminaux revient à déclarer plusieurs noms, plusieurs alias, pour la même chose; dans le domaine électrique, cela revient à poser plusieurs voltmètres sur les mêmes bornes.*

*De même, lorsque plusieurs terminaux sont interconnectés (par des **port map**), la somme algébrique de toutes les quantités **through** qui vont sur cette interconnexion, est nulle.*



5 Types, *Natures*

Un type, selon la définition classique, est la réunion d'un jeu de valeurs et d'un jeu d'opérations sur ces valeurs. Ceci est particulièrement pertinent en VHDL qui possède la notion de surcharge, héritée d'Ada. On peut surcharger les valeurs littérales (par exemple '0' peut être donné comme valeur littérale à un objet de type BIT ou un objet de type CHARACTER). On peut aussi surcharger les opérations sur les éléments d'un type, par exemple définir ce que serait un « + » entre deux éléments d'un type défini ad hoc. Tous les types en VHDL ont les opérateurs « = » et « /= « (égal et différent) permettant de comparer des valeurs.

Attributs *les types ont un jeu d'attributs (§6.2.6 page 50)*

T'BASE: le type de base de T
T'LEFT/**T'RIGHT**/**T'HIGH**/**T'LOW**: valeur la plus à gauche (droite, grande, petite) de T
T'ASCENDING: booléen; true si T est defini avec **to**
A'LENGTH[(N)] taille de la Nème dimension de A
A'ASCENDING[(N)]: booléen, **true** si le range de la Nème dimension de A défini avec **to**
T'IMAGE(X): fonction; représentation en string de X
T'VALUE(X): fonction; valeur du type T lue depuis la string X. Erreurs possibles.
T'POS(X): entier; position de X dans le type discret T (début à 0)
T'VAL(X): valeur du type discret T à la position X
T'SUCC(X)/**T'PRED(X)**:: successeur (prédécesseur) de X dans le type discret T
T'LEFTOF(X)/**T'RIGHTOF(X)**: valeur à gauche (droite) de X dans le type discret T
A'LEFT[(N)]/**A'RIGHT[(N)]**/**A'HIGH[(N)]**/**A'LOW[(N)]**:: valeur la plus à gauche (droite, haute, basse)de l'index de Nème dimension du [type/objet] tableau A
A'RANGE[(N)] l'étendue (range) A'LEFT **to** A'RIGHT ou A'LEFT **downto** A'RIGHT
A'REVERSE_RANGE[(N)] idem en inversant **to** et **downto**
[AMS]N'ACROSS: *type du la banche across de la nature N*
[AMS]N'THROUGH: *type du through de la nature*
[AMS]N'REFERENCE: *terminal de référence de la nature N*

5.1 Qualification

Il arrive qu'à cause de la surcharge (6.2.5 page 49 pour les fonctions et §12.7.3 page 104 pour les procédures), certaines valeurs littérales soient ambiguës dans certains contextes : par exemple la chaîne "001001" peut être une chaîne de bits, ou une chaîne de caractères. En général, le contexte permet de faire la différence, par exemple si cette constante est affectée à une chaîne de caractères ; mais parfois ce n'est pas le cas, par exemple si l'on appelle la procédure WRITE du paquetage TEXTIO (§16.1.2 page 117) qui existe en deux exemplaires précisément surchargées sur les types BIT_VECTOR et STRING.

Dans ce cas, on peut indiquer au compilateur quel est le type souhaité en le nommant, suivi d'une apostrophe (on dit « tick »). `BIT_VECTOR'("001001")`. On peut alors écrire : `WRITE(…,BIT_VECTOR'("010")),…)`. Cela s'appelle « qualifier la valeur ».

Ne pas confondre avec la conversion de type[3] : il s'agit juste d'une indication, on dit au compilateur « telle valeur[4] est de tel type » et cette indication lui permet de résoudre les ambiguïtés. Noter que dans ce cas particulier, l'ambiguïté ne serait pas résolue par la simple utilisation de caractères comme 'A' qui n'appartiennent pas au type BIT : la norme en effet dit qu'on ne peut pas regarder les valeurs littérales dans une chaîne de caractères pour déterminer son type.

[3] La conversion de type prend un objet dont le type est connu et fournit une valeur apparentée dans un autre type, sa syntaxe est : NOUVEAU_TYPE(OBJET), voir par exemple §5.2.1 ci-après.
[4] Rappelons qu'une valeur littérale appartient à un « genre de types », pas à un type particulier. cf §2.7 page 13.

5.2 Types scalaires

Un type scalaire est un type dont les éléments sont atomiques : on ne sait pas et on ne veut pas savoir comment ils sont « faits ». Le langage doit donc fournir toutes les opérations nécessaires pour les manipuler, jusqu'aux moyens de les lire et de les écrire lors d'entrées-sorties. Tous les types scalaires en VHDL ont une relation d'ordre et une direction : on peut donc toujours les comparer par les opérateurs ">", "<", ">=", "<=". La direction permet de spécifier dans quel sens on lit le type, il y a **to** et **downto**. La direction sert lorsqu'on utilise des attributs comme RANGE dans la gestion d'une boucle, ou encore pour se conformer à une convention de représentation pour les traces de simulation (vecteurs de bits représentant des entiers avec bits de poids fort à gauche, par exemple).
Exemples : `1 to 10  - 'Z' downto 'A'.`

5.2.1 Types numériques et conversions

Dans l'ensemble des types scalaires, certains ont des opérateurs arithmétiques ("+" , "- ", etc.) : il s'agit des types entiers, des types réels et des types physiques. Il est concevable d'avoir envie, à l'occasion, de les convertir l'un dans l'autre. VHDL permet toujours la conversion entre types numériques ; mais jamais implicitement. Pour convertir, il suffit d'emballer l'objet avec le nom du nouveau type.
Pourquoi cette obligation ? Prenons l'exemple d'un entier codé sur 32 bits : sa valeur sera probablement codée en « complément à deux ». Prenons un réel sur 32 bits : sa valeur sera probablement codée en utilisant le standard IEEE qui attribue un bit pour le signe, quelques bits pour l'exposant et le reste pour la mantisse (voir note 5 page 31). La conversion de l'un vers l'autre, pour la même valeur représentée, n'est absolument pas gratuite et s'il faut la synthétiser cela se terminera par quelques centaines de portes. VHDL oblige donc le concepteur à marquer explicitement ce besoin de conversion, pour qu'au moins elle ne passe pas inaperçue.
Résultat des conversions (on suppose ici que la fs est l'unité de base active du type TIME)

- Entier => réel : conversion « au mieux ». 3 => 3.0
- Entier => type physique : conversion en compte d'unité de base ; 3 => 3 fs
- Réel => entier : conversion au plus proche. 3.6 => 4
- Réel => type physique : conversion « au mieux » en unités de base. 3.2 => 3 fs
- Type physique => entier : l'entier représentant la valeur en unité de base. 2 ps => 2000
- Type physique => réel : le réel représentant la valeur en unité de base. 2 ps => 2000.0

(Voir la note 6 page 35)

5.2.2 Types discrets

La principale caractéristique des types discrets est que chaque élément a un précédent et un suivant, sauf deux s'ils existent: le premier et le dernier ; ou, ce qui les oppose aux réels, qu'entre deux éléments du type il n'y en a pas toujours un troisième. On reconnaît là la caractéristique des nombres entiers, mais il y a aussi les types énumérés comme BIT, BOOLEAN ou CHARACTER qui sont discrets sans avoir d'arithmétique.

5.2.2.1 Types énumérés

Un type énuméré est un type discret pour lequel on énumère toutes les valeurs que pourront prendre les objets de ce type. Ces valeurs sont symboliques, c'est à dire qu'hormis les opérateurs de comparaison venant de l'ordre dans lequel on énumère, il n'y a aucun opérateur prédéfini.

Par exemple, le type BOOLEAN est défini ainsi :
```
type BOOLEAN is (FALSE,TRUE);
```
On voit donc que FALSE est inférieur à TRUE. Là-dessus, pour que ce type se comporte en honnête type booléen, le paquetage où il est défini (STD.STANDARD §16.1.1 page 115) définit aussi tout le jeu des fonctions logiques comme
```
function "and" (LEFT,RIGHT :BOOLEAN) return BOOLEAN ;
```
Et c'est seulement en voyant la définition des deux valeurs et tout le jeu de fonctions logiques que l'on peut dire qu'il s'agit là d'un type booléen.
Les valeurs symboliques peuvent prendre deux formes : des identificateurs comme FALSE, \bla\ ou TOTO, et des caractères littéraux comme 'A', '0'.
Ainsi, dans le paquetage standard, on trouvera le type BOOLEAN, mais aussi le type BIT défini avec les deux valeurs '0' et '1' et le même jeu d'opérateurs logiques (attention : il n'y a pas d'opérateur logique prédéfini entre un BIT et un BOOLEAN).
On trouve aussi le type CHARACTER, défini par ses 128 positions dans l'ordre du standard ASCII et qu'on peut lire aussi dans le paquetage STANDARD (§16.1.1 page 115). De même, le type STD_LOGIC du paquetage STD_LOGIC_1164 (§16.1.3 page 118) est un type énuméré à 9 états.
On peut même les mélanger dans la même définition, mais cela n'est ni utile ni recommandé.
Exemple : `type BIZARRE is (PIERRE, '2', PAUL, 'Z') ;`
À part les types prédéfinis sus-mentionnés, il est intéressant de définir son propre type énuméré pour rendre abstraits les états d'une machine à états finis, et ainsi programmer l'algorithme avant de s'occuper du codage des états :

Exemple :
```
type type_etat is (reset, init0, init1, fetch0, fetch1,..) ;
signal etat: type_etat;
```

5.2.2.2 Types entiers

Il s'agit de représenter le concept mathématique d'entier, qui se définit « abstraitement » de différentes façons (*Peano : 0 est un entier naturel, tout entier naturel a un successeur, aucun entier naturel n'a 0 pour successeur. deux entiers naturels ayant même successeur sont égaux et* $\mathbb{N}$ *est l'ensemble qui contient 0 et le successeur de chacun de ses éléments.*) Mais on ne sait représenter efficacement les entiers que sur des mots machine –en général sur 8, 16, 32 ou 64 bits) et là où le mathématicien connaît « un » entier abstrait, l'informaticien et ici le concepteur VHDL doit en général faire avec « des » entiers qui seront en fait des intervalles de l'entier mathématique. En VHDL, un type entier se définit par deux bornes et une direction :
```
type T1 is range 1 to 10 ;
```
Un type entier particulier suffit au bonheur de la plupart des concepteurs, c'est le type INTEGER qui est défini dans le paquetage STANDARD (§16.1.1 page 115). Le plus généralement, ce type est défini sur 32 bits et va de -2^{31} à $2^{31}-1$. On obtient ces valeurs en invoquant les attributs INTEGER'HIGH et INTEGER'LOW.
Si toutefois on a envie de définir ses propres types entiers, ce qui n'est pas très fréquent, il faudra se souvenir du fort typage de VHDL : deux types entiers ne sont pas compatibles entre eux même s'ils sont identiquement définis ! Mais ils sont convertibles, voir §5.2.1 page 32.
Ainsi :
```
type T1 is range 1 to 10 ; signal S1: T1;
type T2 is range 1 to 10 ; signal S2: T2;
```
`S1 <= S2` est interdit; Mais `S1 <= T1(S2)` compile.

Un intérêt possible de ce genre de définition est de séparer la représentation de l'objet de sa manipulation abstraite. Les objets du type **type** T3 **is range** 200 **to** 203 ; seront codables sur 2 bits alors que l'algorithme les manipulant fera intervenir les valeurs 200, 201, 202 et 203, ce qui, dans un langage comme C, demanderait 8 bits.
Tous les types entiers ont une arithmétique avec les opérateurs +,- etc. Encore une fois, à cause du typage, cette arithmétique rend toujours un élément du même type. Si le résultat « tombe » en dehors de l'étendue du type, ce sera une erreur à l'exécution.

5.2.3 Types réels

Le type réel mathématique a pour propriété qu'entre deux réels on peut toujours en trouver un troisième, en faisant la moyenne par exemple. Le problème, général en informatique, pas seulement en VHDL, est que la représentation finale du réel se retrouvera forcément sur un nombre de bits fini, et que donc il n'y a qu'un nombre fini de réels représentables[5] La représentation des réels sur un ordinateur est une représentation discrète d'un concept qui ne l'est pas.
Ce triste état de faits a plusieurs conséquences : l'arithmétique est approximative (il faut toujours retomber sur un des réels représentés) et la comparaison de deux réels (égalité et différence) est toujours suspecte si elle intervient dans un algorithme.
Comme pour le type entier, il existe un type réel prédéfini qui fera le bonheur de la plupart des concepteurs. Il est défini dans le paquetage STANDARD (§16.1.1 page 115).

```
type REAL is range -1.0E308 to 1.0E308;
```

Si toutefois on a envie de définir ses propres types réels, ce qui, comme pour les types entiers, n'est pas très fréquent, il faudra encore se souvenir du fort typage de VHDL : deux types réels ne sont pas compatibles entre eux même s'ils sont identiquement définis ! Mais ils sont convertibles, voir §5.2.1. Ainsi :

```
type T1 is range 1.0 to 10.0 ; signal S1: T1;
type T2 is range 1.0 to 10.0 ; signal S2: T2;
```

`S1 <= S2` est interdit; Mais `S1 <= T1(S2)` compile.

La révision 2001 de VHDL, encore peu implémentée, oblige les outils à implémenter les réels sur une représentation de 64 bits conforme à un des standards IEEE.

5.2.4 Types physiques

Les types physiques en VHDL relèvent du fantasme d'ingénieur : ils ont été définis en comité par souci de généralisation du type TIME qui, lui, est nécessaire. On peut donc en VHDL définir des types « à dimension » comme type LONGUEUR ou type MASSE. Et les objets de ces types vérifieront les dimensions, c'est-à-dire qu'on ne pourra pas additionner un mètre et une seconde. Ces types physiques n'ont même pas été repris en VHDL AMS, là où précisément on aurait pu imaginer qu'ils auraient pu être utilisables.

Le seul type physique qui ait été réellement utilisé est celui du paquetage STANDARD (§16.1.1 page 115) est prédéfini, donc aucun concepteur normal ne devrait avoir à définir un type physique lui-même.

[5] D'une façon quasi-universelle, les réels sont représentés par la norme IEEE 754, ce qui donne pour 32 bits :
- Le signe : 1 bit
- L'exposant : 8 bits (on ajoute 127 à l'exposant pour ne pas avoir à gérer en sus des entiers signés.)
- La mantisse : 23 bits.

```
type TIME is range -2147483647 to 2147483647
      units
            fs;
            ps = 1000 fs;
            ns = 1000 ps;
            us = 1000 ns;
            ms = 1000 us;
            sec = 1000 ms;
            min = 60 sec;
            hr = 60 min;
      end units;
```

On peut donc écrire des valeurs de ce type partout où le type TIME est requis :

```
S <= valeur after 5 ns;
```

L'espace entre le 5 et le « ns » est en principe requis, mais ce n'est qu'un message de niveau avertissement la plupart du temps. Reste que, quand on compile, il vaut mieux ne pas encombrer la console avec des centaines de messages de ce genre.

La plupart des simulateurs modernes codent en fait le temps sur 64 bits. Mais il est possible de trouver des implémentations qui se restreignent à 32 bits comme dans l'exemple ci-dessus : en ce cas il suffit de compter les zéros pour voir qu'en comptant en femto-secondes, on ne pourrait simuler que quelques microsecondes de temps.

La solution, qui est dans la norme VHDL, est que les simulateurs –même ceux qui codent sur 64 bits- doivent pouvoir, à l'exécution, changer l'atome de temps de la simulation par un moyen quelconque. Il y a forcément dans un menu de configuration de votre simulateur la possibilité de dire que tous les temps se compteront en nano-secondes, par exemple. Le résultat est une troncature à la nano-seconde de toutes les opérations impliquant du temps. Ceci est nécessaire : penser à la conception d'un circuit gérant une montre digitale pour laquelle l'unité de temps n'est pas plus petite que le dixième de seconde, mais qui doit gérer les années bissextiles sur 4 ans. Mais attention : si l'on porte un modèle, la nouvelle plateforme va raboter à zéro tout ce qui est en dessous de l'unité de base locale : « `T1=T2` » peut être vrai ou faux suivant l'unité de base, si T1 et T2 diffèrent d'une quantité inférieure à l'unité de base du simulateur X et supérieure à l'unité de base du simulateur Y.

Les types physiques ont une arithmétique interne (la somme de deux temps rend un temps) et externe (un temps divisé par un temps rend un entier, un entier multiplié par un temps rend un temps). Les types physiques sont des types numériques, donc convertibles vers les autres types numériques (entiers, réels). Chaque fois que la question se pose, la valeur est convertie en l'unité atomique, la femto-seconde en général. Ainsi INTEGER (5 ns) vaut 5000000. [6]

Si, par vice ou curiosité, on veut créer ses propres types physiques, il suffit de suivre le modèle du type TIME : déclarer un atome de la grandeur concernée, et toutes les unités en fonction de cet atome. Ces unités peuvent être arbitraires (pas forcément des puissances de 10, elles peuvent se recouvrir et on pourrait déclarer dans le même type des yards et des mètres). Si enfin on veut pouvoir écrire des formules impliquant ces différentes unités, il faut surcharger les opérateurs "*", "/" de façon à pouvoir diviser des longueurs par des temps pour obtenir des vitesses. L'explosion combinatoire limitera vite les velléités de l'apprenti-sorcier.

5.3 Types composites

Un type composite, au contraire du type scalaire, est un type dont les valeurs sont structurées d'une manière connue. Le langage doit fournir le moyen de les assembler et d'atteindre leurs éléments, lesquels peuvent à leur tout être scalaires ou composites. Il y a deux classes de types composites : les tableaux (*arrays*) et les enregistrements (*records*).

[6] Ceci ne dépend pas de l'atome de temps choisi pour la simulation (voir au dessus). Quand bien même on déciderait de simuler avec une résolution de 1 ns, la conversion en entier de 5 ns donnerait encore 5000000.

5.3.1 Tableaux (*arrays*)

Un tableau est une structure indexée dont les éléments, qui ont tous le même type, sont adressables par un indice ou un jeu d'indices ; en VHDL il n'y a pas de limite au nombre d'indices, et chacun est donné en spécifiant les bornes basse et haute, et sa direction.
Ces indices doivent être d'un type discret (donc pas nécessairement entier). Ainsi un tableau peut être indexé par une tranche d'un type énuméré, un sous-type ou le type lui-même :

```
type T0 is array(1 to 10, 0 to 5, CHARACTER) of INTEGER;
type T1 is array ('Z' downto 'A') of INTEGER ;
subtype MAJ is CHARACTER range 'A' to 'Z';
type T3 is array(MAJ) of BIT;
type T2 is array (BOOLEAN) of BIT;
```

L'indexation se fait avec des parenthèses, et des virgules s'il y a plus d'une dimension.

```
Ex. : MAT(3,4) ;
```

[AMS] On verra au §5.7.2 page 42 que les natures peuvent aussi être composites de genre tableau.

5.3.2 Tableaux non contraints

Il est possible de déclarer incomplètement un type tableau, en donnant son nom, le type de ses éléments, le type de ses index mais pas les bornes qu'on remplace par une « box » : <>.
Exemple tiré du paquetage STANDARD (§16.1.1 page 115) :

```
type STRING is array (POSITIVE range <>) of CHARACTER ;
```

Ces types « non contraints » permettent de décrire des algorithmes ne dépendant pas des bornes des tableaux traités. Ainsi, on peut facilement faire un algorithme qui « passe en minuscule » une chaîne de caractères sans que ses bornes soient des constantes. On va utiliser STRING qui est précisément un type non contraint.

```
function MIN (chaine: in STRING) return STRING is
  variable res:STRING(chaine'range); -- res est "taillée" comme chaine
begin
  for I in chaine'range loop    -- on parcourt les index de chaine
     res:=minuscule(chaine(I)); -- minuscule est à écrire par ailleurs[7].
  end loop ;
  return res;
end;
```

Néanmoins tout objet déclaré doit avoir des bornes connues du compilateur : les types non contraints ne peuvent servir que comme arguments de sous-programmes (étant entendu que les objets passés, eux, auront des bornes déterminées) et comme « types de types » pour déclarer des sous-types, eux, contraints.
Ex. : ici V1 et V2 sont du même type (STRING) avec la même contrainte, elles sont compatibles.

```
subtype S20 is STRING(1 to 20) ;
variable V1: S20;
variable V2: STRING(1 to 20);
```

[AMS] On verra au §5.7.2 page 42 que les natures peuvent aussi être composites de genre tableau non contraint

[7] La façon « simple » de passer en minuscule est de jouer sur le codage ASCII (32 positions d'écart entre majuscules et minuscules) ; ici
if (chaine(i)>='a') **and** (chaine(i)<='z') **then** res :=character'val(character'pos(chaine(i))-32) **end if**;
Pour se rendre indépendant du codage ASCII, ou l'oublier, on peut remplacer 32 par character'pos('a')-character'pos('A')

5.3.3 Indexations, tranches de tableau, concaténation

Si un objet est de type tableau, par exemple :

```
type TABLEAU is array (1 to 10) of BIT ;
variable V : TABLEAU ;
```

On peut

- L'indicer : `V(2)` rend le BIT en position indexée par 2. On peut le mettre en partie droite ou gauche d'une affectation, le passer à un sous-programme. Attention, contrairement à bien des langages, l'indexation se fait avec des parenthèses simples.
- Le trancher (slicing), si c'est un tableau à une dimension : `V(3 to 5)` rend un tableau de 3 éléments qui sont les mêmes objets que ceux du tableau initial (pas des copies donc). On peut mettre la tranche (slice) en partie droite ou gauche d'une affectation. Ainsi : `V(3 to 5) := "010";` va remplacer 3 bits du tableau V. On peut aussi le passer en argument à un sous-programme qui espère un tableau de 3 bits. Si cet argument est **out** ou **inout**, c'est effectivement une tranche du tableau initial V qui va être passée. On peut utiliser plusieurs tranches du même tableau dans la même instruction, le langage garantit que la copie est faite proprement (c'est-à-dire en utilisant un buffer): `V(1 to 9) := V (2 to 10);`

Un opérateur spécifique, le &, permet de concaténer des vecteurs ou d'abouter à un vecteur un élément à droite ou à gauche, ou enfin de concaténer deux éléments pour faire un vecteur de taille 2. Si une STRING est attendue dans le contexte:

`"AB" & "CD"` rend `"ABCD"`
`"AB" & 'C'` rend `"ABC"`
`'A' & "BC"` rend `"ABC"`
`'A' & 'B'` rend `"AB"`
`'A' & ""` rend `"A"`.

5.3.4 Enregistrements (*records*)

Un enregistrement est la réunion de valeurs de types potentiellement différents, désignées par le nom de leur champ. Cela s'appelle **record** en VHDL, Ada, Pascal, et **struct** en C. Il faut bien dire que l'utilisation d'enregistrements en VHDL est plutôt rare, sauf aux niveaux système.

```
  type R is record
    champ1 : INTEGER ;
    champ2: CHARACTER;
  end record;
  type ID is
    nom: STRING(1 to 10);
    age: INTEGER;
  end record;
  signal S:R;
begin
  S.champ1 <= 12;
  S.champ2 <= 'X';
```

[AMS] On verra 5.7.2 page 42 que les natures peuvent aussi être composites de genre record.

5.3.5 Agrégats

En VHDL on peut construire une valeur de n'importe quel type composite ; c'est encore un héritage d'Ada. Pour ce faire, on utilise une notation fort simple à base de parenthèses et de virgules, reproduisant la structure du type concerné, fût-il transitivement composite. De même que dans les questions d'association (§10.3 page 75), on a toujours le choix entre la notation positionnelle et la notation nommée dans laquelle l'ordre des champs n'a pas d'importance. Voyons sur quelques exemples :

```
type T1 is array (1 to 5) of INTEGER;
variable V1:T1;
…
V1:=(9,4,f(…), x+y, 3); -- les valeurs peuvent être calculées
V1:= (2=>3, others=>0);
type T2 is record nom:STRING(1..14); age:INTEGER;end record;
variable V2: T2;
V2:=("NABUCHODONOSOR", 50);
V2:=(age=>10, nom=>('T', 'O', 'T', 'O', others=>' '));
```

Inutile de dire que le compilateur VHDL vérifie tout: le nombre, le type, la complétude et l'unicité des champs. On notera sur les exemples que les chaînes de caractères n'ont pas de statut spécial, ce sont des tableaux ; seule la notation de leurs littéraux entre "" est spécifique, c'est une simple commodité. Ce sont des tableaux avec des indices, et quand on les affecte tous les éléments doivent être pris en compte (il n'y a pas dans le langage de chaînes de caractères « flexibles » sauf à passer par un paquetage spécifique).
Cette notation entre "" est donc simplement une notation particulière d'agrégat qui s'applique à tout objet d'un genre de type « chaîne de caractères » comme STRING, BIT_VECTOR, STD_LOGIC_VECTOR, etc. Écrire "TOTO" est absolument équivalent à ('T','O','T','O') ou, s'il s'agit du type STRING dont les indices commencent à 1 : (4=>'0', 2=>'O', 1=>'T', 3=>'T')

5.4 Types accès (access)

Les types **access** sont de la famille des types « adresse » de C ou pointeurs de Pascal. Ce sont des types permettant de déclarer des objets sur lesquels on peut faire des allocations dynamiques de mémoire, des structures de données récursives. Leur utilisation dans le monde de la conception électronique est très marginale : à un niveau d'abstraction élevé, très loin de l'implémentation (ces objets ne sont pas synthétisables).
Le type accès se déclare sur un autre type déjà déclaré, même incomplètement :

type P_int **is access** INTEGER ;

L'allocation de mémoire se fait avec l'opérateur **new** sur le type visé:

variable V : P_int ; -- initialisation par défaut à **null**
begin
V :=**new** INTEGER ;

La déallocation se fait par une procédure DEALLOCATE qui est implicitement définie pour tous les types accès créés. Ici `DEALLOCATE(V) ;`
Le déréférencement se fait par l'appel d'un pseudo-champ V.**all** . Ainsi dans notre exemple, V.all est un entier, et peut être mis en place d'un entier à gauche ou à droite d'une affectation.

Faire des types récursifs (listes chaînées, arbres, etc.)

1. Déclarer un type incomplet (on ne déclare que son nom)
2. Déclarer le type accès dessus
3. Compléter le type qui peut faire référence au type accès.

Exemple :

```
type GRANULE ;
type P_GRANULE is access GRANULE ;
type GRANULE is record
  valeur : INTEGER ;
  suivant : P_GRANULE ;
end record ;
```

Attention: une facilité héritée d'Ada fait qu'il n'est pas nécessaire d'écrire le pseudo-champ **.all** quand son omission n'introduit pas d'ambiguïté. Si P est une variable de type P_GRANULE, nous pouvons écrire P.**all**.valeur ou P.valeur, cela sera synonyme. Un examen du manuel de référence montre que cette abréviation est toujours possible quand le champ .**all** n'est pas en bout de chaîne du nom de l'objet (y compris les éventuels attributs).

5.5 Types fichiers (file)

Les fichiers sont vus en VHDL comme des files d'objets (on lit ou on écrit au bout d'un tuyau, pas d'accès direct aux éléments d'un fichier. Leur sémantique de synthèse est à peu près nulle, et leur intérêt réside essentiellement dans trois usages :

- Le chargement au temps zéro de la simulation de structures programmables, ROM, PLA, etc.
- La trace d'événements pendant la simulation, pour examen ultérieur, ou symétriquement l'entrée de stimulus.
- L'écriture de messages à la console, ou l'entrée de paramètres depuis le clavier.

Là-dessus, les concepteurs du langage ont craint que les fichiers ne servent à un concepteur pervers pour communiquer entre processus (le seul canal autorisé étant le signal ou, au pire, la variable partagée). En effet, avec un gestionnaire de fichiers classique, on pourrait imaginer qu'un processus ouvre un fichier, écrive dedans et le ferme pendant qu'un autre peut aller y lire et donc déterminer l'ordre d'exécution des deux processus, ce qui n'est pas grave, mais aussi rendre la simulation dépendante de cet ordre, ce qui est grave et réservé à l'usage des variables partagées.

Pour tenter d'éviter cette catastrophe, le système de fichiers en VHDL ne prévoyait avant 1993 ni ouverture ni fermeture de fichier : un fichier était ouvert à sa déclaration, et jamais fermé. Bien des modèles sont écrits dans ce style. Il y a aujourd'hui les primitives d'ouverture et de fermeture, qui ne devraient servir que pour gérer les messages d'erreur qui s'y attachent

Donc si on déclare:

```
file F1: MON_TYPE;
```

Le fichier devra être explicitement ouvert avec FILE_OPEN

Autres exemples de déclarations:

```
file F2: MON_TYPE is "bla.bla"; -- ouvre le fichier bla.bla en lecture
file F3: MON_TYPE open READ_MODE is "bla.bla"; -- pareil
file F4: MON_TYPE open WRITE_MODE is "bla.bla"; -- ouvre en écriture
```

Le paquetage TEXTIO est, dans l'immense majorité des cas, le seul usage que le concepteur aura des fichiers : on s'y reportera au §16.1.4 page 117. À noter un très utile paquetage STD_LOGIC_TEXTIO -§16.1.4 page 120- qui n'est pas standard mais souvent disponible et de toutes façons accessible en source un peu partout, et qui fournit sur STD_LOGIC les primitives d'entrées-sorties sur BIT que l'on trouve dans TEXTIO.

Définir son propre type de fichier :

```
type MON_TYPE is file of INTEGER ;
file MON_FICHIER : MON_TYPE open READ_MODE is "c:\mon-chemin\le_nom.dat" ;
```

(il existe READ_MODE, WRITE_MODE et APPEND_MODE, éléments d'un type énuméré de STD.STANDARD)

La création d'un type fichier crée automatiquement les procédures READ, WRITE et la fonction ENDFILE suivantes:

```
procedure READ(   file F : MON_TYPE ;
                  VALUE : out INTEGER) ;
procedure READ(   file F : MON_TYPE ;
                  VALUE : out INTEGER;
                  LENGTH: out NATURAL);
procedure WRITE(  file F : MON_TYPE ;
                  VALUE : in INTEGER) ;
function ENDFILE( file F: MON_TYPE) return BOOLEAN;
procedure FILE_OPEN(file F: MON_TYPE;
                  EXTERNAL_NAME: in STRING;
                  OPEN_KIND: in FILE_OPEN_KIND:=READ_MODE);;
procedure FILE_OPEN(STATUS: out FILE_OPEN_STATUE;
                  file F: MON_TYPE;
                  EXTERNAL_NAME: in STRING;
                  OPEN_KIND: in FILE_OPEN_KIND:=READ_MODE);
procedure FILE_CLOSE(file F:MON_TYPE);
```

Le statut rendu par une des surcharges de FILE_OPEN vient du paquetage STANDARD, c'est un type énuméré qui comprend les valeurs `OPEN_OK`, `STATUS_ERROR`, `NAME_EROR`, `MODE_ERROR`.

- `STATUS_ERROR` : par exemple fichier fermé alors qu'on le suppose ouvert.
- `NAME_ERROR` : le nom donné ne va pas sur le système d'exploitation
- `MODE_ERROR` : écriture d'un fichier ouvert en lecture ou inversement.

5.6 Sous-types (subtypes)

Le sous-type est l'association d'un type et d'une contrainte. Un objet appartenant à un sous-type appartient « complètement » au type parent, mais il doit en plus vérifier dynamiquement la contrainte.

Exemple trivial tiré du paquetage STANDARD : les entiers positifs sont des entiers (le type) qui doivent être positifs (la contrainte). On écrit :

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH ;
```

Une variable de ce type sera complètement compatible avec des objets entiers ou des objets d'autres sous-types d'entier (par exemple `NATURAL` qui va de `0` à `INTEGER'HIGH`). La vérification du type se fait à la compilation, celle de la contrainte à l'exécution (sauf optimisations du compilateur). L'usage des sous-types ralentit la simulation tout en apportant un gain de sécurité, et c'est pourquoi la plupart des simulateurs offrent la possibilité de « débrancher » la vérification des contraintes une fois le modèle validé.

Même dans le cas pathologique suivant où les contraintes sont disjointes, eh bien les éléments de ces deux types seront compatibles (on pourra compiler des affectations d'un objet d'un type avec la valeur de l'autre). Mais l'erreur est garantie à l'exécution, un peu comme le serait une division par zéro.

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH ;
subtype NEGATIVE is INTEGER range INTEGER'LOW to -1 ;
V1: POSITIVE;
V2: NEGATIVE;
…
V1:=V2; -- légal, plantera à l'exécution.
```

L'utilisation de sous-types explicitement écrits par le concepteur peut se justifier en association avec l'instruction **case** ou **select** qui exigent que chaque valeur du (sous-)type soit traitée une fois et une seule. Supposons par exemple qu'on veuille faire un case ou un select sur les combinaisons de 3 bits :

```
process
  subtype PERTINENT is BIT_VECTOR (1 to 3) ;
…
begin
…
case PERTINENT'(bit1 & bit2 & bit3) is
              -- utilisation de la qualification, voir §5.1 page 31
   when "000" =>…
   when "001" =>…
   when "010" =>…
   when "011" =>…
   when "100" =>…
   when "101" =>…
   when "110" =>…
   when "111" =>…
end case;
…
end process;
```

En l'absence du sous-type, d'après les règles VHDL le case aurait dû comprendre une branche **others** vide, qui n'aurait aucun sens matériel ; ce qui pourrait chagriner le concepteur.

Une autre application importante des sous-types est leur utilisation pour définir le comportement de signaux en cas de conflits. Cet usage est rarement laissé au concepteur qui utilisera plutôt des sous-types prédéfinis à cet usage, tous les détails sont §7.5 page 57.

[AMS] Enfin, dans le monde analogique, il est possible de contraindre un (sous)type en lui attribuant une tolérance: c'est une chaîne de caractères sans autre signification dans le langage, qui sera utilisé par le noyau de simulation pour régler ses heuristiques au mieux. Exemple: `subtype courant is real tolerance "tolerance_courant";`

Toutes les quantités déclarées de ce sous-type partageront la même tolérance.

5.7 [AMS] Natures

VHDL-AMS permet de résoudre des systèmes d'équations différentielles entre deux points de simulation digitale (voir §8.2 page 62). Il se trouve que, dans les domaines qui intéressent le concepteur et qui impliquent des descriptions

***Attributs** (§6.2.6 page 50):*

- *[AMS]N'ACROSS: type du la banche across de la nature N*
- *[AMS]N'THROUGH: type de la branche through de la nature N*
- *[AMS]N'REFERENCE: terminal de reference de la nature N.*

hiérarchiques, il y a toujours la notion de «effort entre» et de «flux dans», par exemple dans le domaine électrique, la tension et le courant. Dans le domaine thermique, la température et le flux de chaleur. En mécanique, la vitesse et la force, en hydraulique la pression et le flux de fluide. Avec ces deux notions viennent des équations classiques: entre deux terminaux il y a un et un seul effort, et dans un terminal la somme algébrique des flux est nulle.

Cela donne, dans le domaine électrique qui nous intéresse le plus, les lois de Kirchhoff (la tension entre deux équipotentielles est unique, la somme des courants dans une équipotentielle est nulle.) Mais VHDL-AMS peut mélanger différents domaines physiques, par exemple thermique et électrique, ou mécanique. La nature est en quelque sorte le genre de type des terminaux (§4.7.2 page 29). Il n'y a pas de nature prédéfinie, par contre il existe des paquetages écrits en VHDL les définissant: voir les §16.4.1 à 16.5.6 pages 137 à 148.

5.7.1 [AMS]Natures scalaires

Une nature scalaire est définie par

- *Son nom*
- *Le type ou le sous-type de la quantité «effort», le* ***across*** *(forcément du genre réel, en général un sous-type de REAL)*
- *Le type ou le sous-type de la quantité «flux», le* ***through*** *(forcément du genre réel, en général un sous-type de REAL)*
- *Le nom d'un terminal de cette nature qui sert de référence.*

BNF simplifiée:

```
nature nom-nature is
    nom-(sous-)type across
    nom-(sous-)type through
    nom-terminal reference ;
```

Par exemple:

```
subtype I is real;
subtype v is real;
nature electrical is
      v across
      i through
      masse reference;
```

5.7.2 [AMS] Natures Composites

Une nature peut aussi être définie comme un composite, de genre tableau ou enregistrement. Les éléments de ce composite seront d'autres natures composites ou, in fine*, des natures scalaires.*

```
nature vect is array (1 to 10) of electrical;
nature rec is record x: thermic; d: vect; end record;
```

Une nature peut encore être d'un type tableau non contraint:

```
nature vect1 is array (integer range <>) of electrical;
```

Dans ce cas, de même que les types tableaux non contraints, elle peuvent servir à déclarer d'autres natures, ou pour définir le type d'un argument de sous-programme, mais pas pour déclarer des terminaux.

5.8 [AMS] Subnatures

BNF (voir définition page 9) :

```
subnature_declaration ::=
     subnature identifier is subnature_indication ;

subnature_indication  ::= nature_mark [ index_constraint ]
       [ tolerance string_expression across
                   string_expression through ]
```

Une sous-nature, de même qu'un sous-type, est une nature plus une contrainte. Comme pour les types et sous-types, les objets de natures et de ses différentes sous-natures sont compatibles.

*Une sous-nature peut spécifier un index pour une (sous-)nature parente non-contrainte. Elle peut aussi spécifier des tolérance sur les aspects **across** et **through** sous la forme de chaînes de caractères.*

6 Expressions et Fonctions, Attributs, Groupes

Les valeurs peuvent provenir de quatre structures:

- Les littéraux (§2.7 page 13)
- Les références à un porteur de valeur : variable, signal, *quantité*, constante (chapitre 4 page 23).
- Les attributs qui rendent une valeur prédéfinie (chapitre 15 page 113) et déclarés par l'utilisateur, que nous allons voir dans ce chapitre au §6.2.6 page 50.
- Les résultats de fonctions dont font partie les expressions, que nous allons voir ci-après.

Attributs *les expressions, fonctions, etc. n'ont d'autres attributs que ceux hérités de leurs types.*

6.1 Toute expression est un arbre de fonctions

Tous les opérateurs comme + ou – sont définis, en VHDL, comme étant des fonctions ayant deux arguments et rendant un résultat. Par exemple, l'opérateur + entre deux entiers est défini dans STD.STANDARD:

```
function "+" (LEFT,RIGHT: INTEGER) return INTEGER;
```

Nonobstant le fait que son nom est encadré par deux doubles apostrophes et qu'elle va être employable sous forme dyadique (A+B), cette fonction est exactement de même nature qu'une fonction sinus:

```
function SIN(ARG:REAL) return REAL;
```

Par exemple, on peut l'écrire soi-même sur n'importe quel type:

```
function "+" (A,B: CHARACTER) return STRING is
 variable RESULTAT:STRING(1 to 2);
begin
  RESULTAT(1):=A;
  RESULTAT(2):=B;
  return RESULTAT;
end function "+";
```

Nous pouvons ensuite écrire 'A' + 'B' et le résultat vaut "AB" [8]. Le "+" appelé sera bel et bien celui qui a été écrit ci-dessus.
Par conséquent, toute expression, quelle que soit sa complexité, peut se ramener à un arbre d'appels de fonctions. Par exemple, l'expression suivante

```
Résultat:=X + SIN(Y) – (3*(A+B))
```

peut se représenter suivant l'arbre ci-contre. Elle est équivalente à:

```
Résultat:="-"("+"(X,SIN(Y)),"*"(3,"+"(A,B)))
```

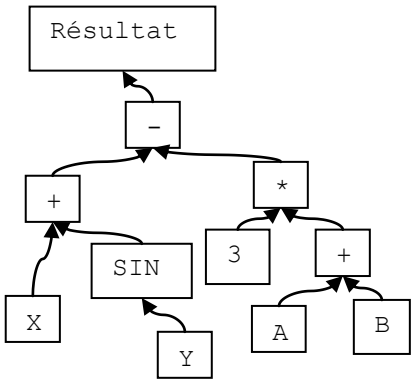


Figure 4 Arbre d'évaluation d'une expression

[8] L'opérateur "&" prédéfini fait déjà ça très bien.

Cette expression passera très bien la compilation: c'est la même expression avec les mêmes opérateurs appelés dans le même ordre. Elle produira le même code exécutable.

Ainsi toute expression en VHDL se ramène à deux questions; comprendre l'ordre d'évaluation des opérateurs en l'absence de parenthèses (la priorité) et comprendre le mécanisme d'appel de fonction car les opérateurs sont des fonctions.

6.2 Les opérateurs et leur priorité

Les opérateurs peuvent être classés en:

- **arithmétiques**: + - * / ** **abs mod rem**,
  - + et – ont une variété modadique (comme dans -4) et dyadique (2-1).
  - + - * / sont classiques
  - ** est la puissance (le terme de droite doit être du type prédéfini INTEGER)
  - **abs** la valeur absolue, sur tout type numérique
  - **mod** est le modulo sur types de genre entier (8 **mod** 3 = 2, 8 **mod** -3 = -2)
  - **rem** : reste de la division sur types de genre entier (8 **rem** 3 = 2, 8 **rem** -3 = 2)
- **logiques: and or nand nor xor xnor** ont le sens habituel. Ils fonctionnent de façon prédéfinie sur les types BOOLEAN, BIT et BIT_VECTOR. Les opérateurs **nand** et **xnor** ne peuvent pas être cascadés (comme dans A **and** B **and** C).
- **relationnels:** = /= (différent) > >= < <= ont leur sens classique et fonctionnent sur tous les types scalaires ainsi que sur les vecteurs de types scalaires. Ils ne peuvent pas être cascadés, cf infra.
- **shift et rotation: sll srl sla sra rol ror** n'existent que depuis VHDL'93 et s'appliquent sur des vecteurs (tableaux à une dimension) pour l'opérateur de gauche, et demandent le type prédéfini INTEGER comme opérateur de droite. La première lettre signifie **s**hift ou **r**otate. Dans le cas des shift, la seconde lettre signifie **l**eft ou **r**ight et la troisième est **l**ogique ou **a**rithmétique. Pour les **ro**tate, il y a **l** pour **l**eft et **r** pour **r**ight.
  - shift: décalage de N positions.
  - rotate: rotation de N positions, les positions libérées d'un côté étant remplacées par les positions poussées de l'autre.
  - logique (shift): les positions ouvertes sont remplacées par la valeur par défaut du type de l'élément du tableau, donc T'LEFT. Exemple: à force de faire des shifts logiques sur un vecteur de bits, un vecteur sera entièrement rempli de '0'.
  - arithmétique (shift): les positions ouvertes sont remplacées par la duplication de la valeur de la première position ouverte. Exemple: à force de faire des **sla** sur un vecteur de bits, un vecteur sera entièrement rempli par la valeur du bit le plus à droite.
- **Le concaténateur &:** il permet de concaténer des tableaux à une dimension (de tous types), ou de concaténer un tel tableau avec un élément (sa taille augmente de 1), ou enfin de concaténer deux éléments pour faire un vecteur de taille 2.

Comme dans tous les langages, les opérateurs de VHDL ont une priorité respective qui détermine l'ordre d'exécution des fonctions en l'absence de parenthèses.
Hormis la priorité elle-même, les opérateurs de comparaison : **=, /=**, etc. ont une particularité héritée d'Ada: ils ne sont pas cascadables[9]. Alors que l'on peut écrire 2+3*5+4, on ne peut pas

[9] un opérateur non associatif comme **nand** non plus.

écrire `A=B>C` pour une raison que le lecteur avisé comprendra certainement d'autant mieux qu'il comprendra péniblement l'expression citée, surtout s'il a déjà programmé en C où cela y est autorisé. En VHDL il faudra parenthéser: `(A=B)>C` pour autant que cela veuille dire quelque chose, ici C doit être du type Booléen sauf redéfinition des opérateurs.

Pour le reste, voici la priorité des opérateurs, du plus fort au plus faible. Quand la priorité est identique, c'est leur emplacement qui fait la différence et le plus à gauche dans l'expression passe avant: dans « 2+3-4 », le « + » est exécuté avant le « – » qui verra « 5-4 ».

Plus prioritaire

- ** **abs not**
- * / **mod rem**
- **+ -** modadiques (comme dans +3 ou -5)
- **+ -** dyadiques (comme dans 2+3) **&** (concaténateur)
- **sll srl sla sra rol ror**
- = /= < <= > >=
- **and or nand nor xor xnor**

Moins prioritaire

Attention: si l'on peut surcharger les opérateurs existants (écrire des fonctions qui seront appelées sous le nom de l'opérateur), on ne peut ni changer leur priorité respective, ni les restrictions d'appel (cascader des opérateurs **nand** est interdit) ni en créer d'autres que ceux qui sont dans le lexique.

6.2.1 Fonctions

Une fonction est un objet qui rend une valeur de n'importe quel type, et une seule; elle est appelée avec des arguments. Elle ne peut avoir que des arguments de mode **in**, de classe constante, signal et fichier (**file**) - le mot clé **constant** est le défaut-. Elle peut être récursive. C'est une des deux sortes de sous-programme, l'autre est la procédure détaillée en §11.8.2.1.2 page 100 pour ce qui est de son utilisation en contexte concurrent, et en §12.7 page 103 pour ce qui est de son utilisation en contexte séquentiel.

6.2.2 Pureté

6.2.2.1 Fonctions pures

Une fonction pure a pour caractéristique de ne pas avoir d'effets de bord. Le résultat de la fonction ne dépend que de ses arguments, et l'effet de la fonction n'est que de rendre une valeur. Une fonction mathématique comme SINUS est une fonction pure. Les fonctions pures sont préfixées du mot-clé **pure**, lequel est le défaut (si on l'omet, la fonction est considérée comme pure.)
Quand une fonction est ainsi marquée **pure**, le compilateur vérifie qu'elle n'accède pas en écriture des objets hors de sa région déclarative, sauf pour appeler d'autres fonctions pures.

Exemple:

```
pure function plusun (arg: in integer) return integer is
begin
  return arg + 1;
end function plusun;
```

6.2.2.2 Fonctions impures

Une fonction impure est une fonction qui peut avoir des effets de bords (elle peut aussi accidentellement ne pas en avoir). Le résultat de la fonction peut dépendre du contexte, et son exécution peut le modifier. Une fonction de comptage (qui rend le nombre de fois où elle a été appelée, nécessitant donc la modification d'une variable externe), ou les fonctions NOW[10] du paquetage STANDARD qui rendent l'heure qu'il est dans le monde simulé, et donc une valeur différente à chaque appel, sont des fonctions impures (§16.1.1 page 115). Les fonctions impures sont préfixées du mot-clé **impure.**

Exemple:

```
variable vcompteur : integer:=0;
impure function compteur return integer is
begin
  vcompteur:=vcompteur+1;
  return vcompteur;
end function compteur;
```

6.2.3 Déclaration et Définition

BNF (voir définition page 9) :

```
[pure|impure] function designator [(formal_parameter_list)]return type_mark
```

Dans la syntaxe de la déclaration, on déclare donc:

- la pureté (défaut: **pure**)
- le nom de la fonction. Il peut être un identificateur, ou un opérateur entre doubles apostrophes comme "+".
- la liste de ses arguments. La forme de la liste des arguments est vue §10.2 page 75. On ne peut utiliser que le mode **in** qui est le défaut. On peut passer des objets de toutes classes sauf **variable**.
- le type de la valeur retournée.

Exemple:

```
function TOTO (signal S: in BIT; V1,V2: in INTEGER) return BOOLEAN
```

- **Déclaration seule**: si l'on veut faire une déclaration seule, on termine par un point virgule[11]: la fonction est alors visible pour ses éventuels clients; son code devra bien entendu être écrit plus tard dans une définition. On la déclarera ainsi dans une déclaration de paquetage. La déclaration seule n'est nécessaire que dans ce cas et dans celui où l'on voudrait faire de la récursivité croisée: F1 appelle F2 qui appelle F1. Quand la déclaration seule n'est pas nécessaire, elle est néanmoins possible.
  Exemple:

  ```
  function TOTO (signal S: in BIT; V1,V2: in INTEGER)
           return BOOLEAN ;
  ```

- **Déclaration et définition**: quand on veut la définir (en écrire le code) on termine la déclaration par **is** et ensuite on peut déclarer des objets locaux, mettre le mot-clé **begin** et écrire l'algorithme jusqu'au **end**. Si la définition fait référence à une déclaration antérieure, par exemple si on écrit le code de la fonction dans un corps de paquetage alors qu'elle est déclarée dans la partie visible, les deux déclarations doivent être des copié-collé l'une de l'autre.

[10] Les fonctions NOW seront probablement déclarées pures, contre toute intuition, lors de la prochaine version du langage, de façon à pourvoir les utiliser dans des contextes statiques.

[11] L'oubli de ce point virgule n'est souvent remarqué que bien plus loin par le compilateur. En effet les items qui suivent pourraient bien être, pour lui, des items déclaratifs de la fonction qu'il croit en train d'être déclarée. Les messages d'erreurs sont alors surprenants.

Exemple, ici TOTO compare V1 et V2 et si égalité rend TRUE si S est en événement.:

```
function TOTO (signal S: in BIT; V1,V2: in INTEGER)
         return BOOLEAN is
begin
  if (S'EVENT) and (V1 = V2)
      then return TRUE;
  end if;
  return FALSE;
end function TOTO;
```

6.2.4 Instructions

L'intérieur d'une fonction appartient au monde séquentiel (comportemental). Les fonctions peuvent contenir toutes les instructions séquentielles énumérées §12.1 page 97, saut le **wait** – une fonction se déroule à temps de simulation strictement nul- *[AMS] et sauf aussi le **break** dans le monde analogique*. Une fonction doit contenir au moins une instruction **return** (12.5.3 page 102), c'est celle qui rendra la valeur finalement calculée. Elle peut en contenir plusieurs. Cette instruction doit être exécutée pour sortir de la fonction, si l'exécution rencontrait le **end** de la fonction cela signifierait qu'elle n'aurait pas rencontré de **return** et ce serait une erreur à l'exécution. L'effet de l'instruction **return** est de la catégorie «frein à main»: on sort immédiatement de la fonction et on rend la valeur mentionnée.

6.2.5 Appel et surcharge

L'appel d'une fonction se fait par son nom dans une expression. Les arguments sont donnés entre parenthèse, selon les différentes formes détaillées §10.3 page 75. S'il n'y a pas d'argument comme dans le cas des fonctions NOW (§16.1.1 page 115), on ne met pas la paire de parenthèse. Si une valeur par défaut a été spécifiée dans la déclaration, on peut omettre cet argument à l'appel.

```
X := plusun(Y) * Z;
```

Si la fonction rend un type accès (§5.4 page 38), elle peut se trouver à gauche d'une affectation:

```
ref(X).all := 23;
```

La surcharge est le fait que l'identification de la fonction peut se faire par son nom et son contexte: ses arguments (leur nombre et leur type), et le type de retour. Cette facilité est partagée avec les littéraux des types énumérés (§2.7.4 page 14) qui sont vus, pour l'occasion, comme des fonctions sans argument rendant leur propre valeur. Les procédures peuvent aussi être surchargées entre elles, mais elles n'apparaissent pas en position de valeur, donc elles ont leur monde de résolution à elles.

Attention : les valeurs utilisées ne participent pas à la résolution de la surcharge, seulement leur type ou leur genre de type le cas échéant. Pour prendre l'exemple d'une procédure qui est dans les annexes §16.1.2 page 117, TEXTIO.WRITE("XYZ") est ambigu car il y a deux WRITE acceptant des types de genre chaîne de caractère (STRING et BIT_VECTOR). Le fait que 'X','Y' et 'Z' ne soient pas des valeurs du type BIT n'est pas considéré, contre toute intuition. Il faudra qualifier (§5.1 page 31) l'expression littérale pour que ça passe la compilation : STRING'("XYZ")

La surcharge utilisée malicieusement peut rendre un texte incompréhensible[12]. Voyons par exemple:

```
F0(X).all := F1(3) + F2(Y,Z,T)
```

Tout peut être surchargé: les noms F0, F1, F2 peuvent exister plusieurs fois avec différents jeux d'arguments ou différents arguments spécifiés par défaut. Mais X, Y, Z, T aussi, soit en tant que fonctions sans argument (comme `function X return TIME ;`) , ou fonctions avec des arguments ayant une valeur par défaut (comme `function Y(ARG : BIT :='0') return BIT ;` appelable sans argument), soit en tant que littéraux énumérés partagés par plusieurs types. Et le "+" lui-même existe sur tous les types numériques et certains autres. Il est facile, et cela a été un jeu de programmeurs pour Ada, de construire des programmes relativement courts *ad-hoc* où l'explosion combinatoire est telle que le compilateur prend une heure pour la résoudre et trouver la solution (le jeu consiste à ce qu'il y en ait une).

Mais dans la vaste majorité des cas, c'est une commodité très utile, reprise d'Ada et qui existe dans d'autres langages.
C'est ainsi que les opérateurs comme + ou – sont lourdement surchargés sur tous les types numériques ainsi que sur des types définis à base de vecteurs de bits dans des paquetages standard (§16.2 page 121). Et l'utilisateur peut en rajouter voir l'exemple du §6.1 page 45.
On peut voir aussi dans le paquetage STANDARD (§16.1.1 page 115) qu'il y a deux fonctions NOW qui ne se distinguent que par leur type de retour.
`X:= NOW` sera donc l'appel de l'une ou de l'autre selon le type de X.

Et les ambiguïtés? La règle est simple: pas d'option par défaut. Si le compilateur découvre que deux fonctions ou deux littéraux énumérés conviennent à un endroit donné, il ne compile pas.

Exemple:

```
B : boolean := '0' < '1';
```

Ici '0' et '1' sont des éléments de deux types énumérés: BIT et CHARACTER. La fonction "<" rendant un booléen existe sur les deux comme elle existe sur tous les types scalaires: le compilateur ne sait pas laquelle appeler. Alors même que, dans cet exemple, le résultat serait identique dans les deux cas; le langage est complexe et le compilateur est intelligent mais ne va pas jusque là.

6.2.6 Attributs préféfinis

VHDL hérite d'Ada la possibilité de «demander» au compilateur des informations sur pratiquement tous les objets du langage, ou de créer des objets prédéfinis (par exemple S'DELAYED(5 ns) crée une copie de S décalée de 5 ns.)

La notation est NOM ' ATTRIBUT ou NOM ' ATTRIBUT (argument)

---

[12] Voici par exemple un extrait de texte VHDL traité par un programme d'"obfuscation" pour le rendre illisible: en sus du fait que les identificateurs sont à base de l (L minuscule), 1 (un) , et I (i majuscule), les fonctions ont toutes le même nom quand c'est possible et jouent sur la surcharge. La distinction se faisant par le contexte (exemple de Krypton Synopsys tiré d'un rapport de l'ESA).

```
prOceSS BEgIN waiT ON lllllllll ; If lllllllll = '1' anD lllllllll'eVENt AnD
lllllllll'LaST_VAlue = '0' ThEN iF ( lllllllll = '1' ) ThEN lllllllll <=
lllllllll ; END IF ; lllllllll <= lllllllll ; lllllllll <= lllllllll ; ……
```

En franglais, par opposition à l'apostrophe doublée qui encadre les caractères, cette apostrophe unique se dit, dans ce cas, «tick» et cela se lit donc «NOM *tick* ATTRIBUT».

Un attribut peut rendre une valeur (il sera vu comme une fonction) ou un signal, ou un type ou encore une étendue (*range*). *[AMS] Ou une quantité ou un terminal*. Les choses rendues par un attribut peuvent elles-mêmes être attribuées selon leur genre:
S'DELAYED(5 ns) ' TRANSACTION
[AMS] Q ' DOT ' DOT.

La liste complète des attributs est au chapitre 15 page 113. Ils sont mentionnés en encadré avec leur description, chaque fois que c'est pertinent, au fil de ce manuel.

Comme règle aide-mémoire, on pourra retenir que, chaque fois que le concepteur a l'impression que tel renseignement sur tel objet devrait être disponible, il l'est probablement sous la forme d'un attribut –et, *a contrario*, si l'information n'est pas disponible c'est probablement que l'information n'est pas si clairement établie que ça au moment de compiler l'unité, pour le compilateur.

Comme autre règle aide-mémoire, on peut se dire que la majorité des attributs qui rendent une fonction ne font que rendre une information qui est déjà quelque part dans le code, et que leur utilisation ne fait que rendre le code plus résistant à un changement de plate-forme. Par exemple, utiliser INTEGER'HIGH ne fait que remplacer par 2^{31}-1 (probablement) et cette valeur est en dur dans le paquetage STANDARD (§16.1.1 page 115). Utiliser T'RANGE alors qu'on a soi-même défini T avec une étendue de 1 à 10 est équivalent à utiliser «1 **to** 10». Mais l'indirection apporte la sécurité de n'avoir qu'un emplacement à gérer si l'on veut changer l'étendue.

6.3 Attributs définis par l'utilisateur

Il est possible de définir des attributs sur quasiment n'importe quel objet VHDL; de tels attributs, une fois définis, seront vus comme des fonctions sans argument dans le reste du modèle, selon les règles de visibilité bien entendu. Pour ce qui est de leur appel, ils partagent la syntaxe générale des attributs et vont donc s'ajouter à la liste des attributs prédéfinis (chapitre 15 page 113). On peut ainsi attribuer des objets ou des ensembles d'objets: les groupes. L'usage de ces fonctionnalités est assez anecdotique: dans le langage ce n'est qu'une façon compliquée de définir des constantes. Hors langage certains outils peuvent en tirer de l'information contextuelle.
On définit un attribut par son type, ensuite on l'attache aux objets concernés:

```
attribute TOTO: INTEGER;
```

Après quoi on peut attacher cet attribut en lui donnant une valeur à quasiment tous les objets du langage:

```
attribute author: STRING;
…
attribute author of E:entity is "Nabuchodonosor";
```

À partir de là, l'appel de E'author rendra la chaîne de caractères prédéfinie, sous la forme d'une constante.
Lorsqu'il y a ambiguïté pour cause de surcharge (voir §6.2.5 page 49), elle peut être levée par la définition d'un profil: il s'agit de l'énumération des types des arguments dans l'ordre de

déclaration, suivie du type de la valeur retournée dans le cas des fonctions, le tout entre crochets []:

```
attribute author of "+" [STD.STANDARD.INTEGER,
                         STD.STANDARD.INTEGER
                         return STD.STANDARD.INTEGER]: function is "IEEE";
```

On peut aussi remplacer le nom de la l'objet attribué par **all** ou **others**. L'attribut s'appliquera alors à tous les objets de ce nom pas encore attribués (**others**) ou à tous (**all**).

Un attribut de ce genre est défini dans STD.STANDARD (§16.1.1 page 115): FOREIGN, sous la forme d'une chaîne de caractères; cet attribut permet de faire le lien avec des objets non-VHDL, par exemple pour indiquer au compilateur que le corps de tel sous-programme n'est pas en VHDL mais disponible sous forme quelconque (binaire par exemple) dont on donne le nom pour une édition de liens..

6.4 Groupes

Un groupe est l'association de plusieurs objets du langage, pratiquement tous les objets peuvent être ainsi groupés. Une fois un groupe défini, on peut lui donner un attribut. On peut même faire des groupes de groupes.

```
group PIN2PIN is (signal, signal);
group ENSEMBLE is (label <>); -- <> pour «tout nombre de».

…

group G: PIN2PIN (S1,S2);
group INTERESSANT: ENSEMBLE (L1,L2,L3,L4);
```

On peut évidemment attribuer un groupe, ce qui est la seule façon de le faire intervenir dans l'algorithme plus tard, en utilisant cette valeur:

```
attribute AUTHOR of INTERESSANT is "jr";
```

7 Signaux, Pilotes, Résolution, Inertie

Les signaux sont un des « porteurs de valeurs » en VHDL, ceux qui sont censés représenter du matériel. Ce sont des structures complexes et cela justifie ce chapitre séparé. Les autres porteurs de valeurs vus chapitre 4 page 23 sont les constantes, à la sémantique triviale, les variables (§4.4 page 24), plutôt utilisées pour la modélisation comportementale, les fichiers (§4.7 page 39), plutôt accessoires de la gestion de la simulation ou nécessaires lors des initialisations *[AMS] et enfin les quantités, variables au sens mathématiques du terme et utilisées en simulation analogique ou mixte.*

Un signal se déclare dans une zone concurrente donc une zone déclarative d'une architecture, d'un bloc ou d'un package. **signal** S : BIT ; S'il sert dans un interface d'entité, il est aussi un **port**.

Le type d'un signal peut être quelconque, scalaire ou composite, à l'exception d'un type **access** ou **file** (ou contenant transitivement un tel type s'il est composite.)

Un signal peut avoir un genre **bus** ou un genre **register**. Le mot-clé se met après le type, et dans ce cas le type doit être résolu, voir page §7.5 page 57. Cette fonctionnalité n'est pas d'un usage très courant. Voir le détail §11.8.2 page 93.

Attributs les signaux ont des attributs (§6.2.6 page 50): spécifiques, en sus de ceux qui viennent de leur type:

S'DELAYED: signal de même type que S; rend la copie de S retardé de un delta (un cycle, voir §8.1 page 61).
S'DELAYED(T): signal de même type que S; rend la copie de S retardé du temps T (T doit être une constante)
S'STABLE: signal booléen; rend TRUE si S n'a pas changé de valeur pendant le dernier cycle delta (§8.1 page 61).
S'STABLE(T): signal booléen, rend TRUE si S n'a pas changé de valeur pendant le temps T qui doit être une constante.
S'QUIET: signal booléen, rend TRUE si S n'a pas été affecté pendant le dernier cycle delta.
S'QUIET(T): signal booléen, rend TRUE si S n'a pas été affecté pendant le temps T qui doit être une constante.
S'TRANSACTION: signal bit; change de valeur chaque fois que S est affecté qu'il change de valeur ou pas.
S'EVENT: fonction booléenne; rend TRUE si le signal a changé de valeur pendant le dernier cycle delta (voir §8.1 page 61).
S'ACTIVE: fonction booléenne; fonction booléenne; rend TRUE si le signal a été affecté pendant le dernier cycle delta (voir §8.1 page 61).
S'LAST_EVENT: fonction TIME; rend le délai depuis le dernier événement sur S.
S'LAST_ACTIVE: fonction TIME; rend le délai depuis la dernière affectation de S.
S'LAST_VALUE: fonction du type de S; la valeur de S avant le dernier événement.
S'DRIVING: fonction booléenne; rend TRUE si S a un driver dans le processus (équivalent) courant qui ne contient pas null (pas déconnecté, voir $11.6.2 page 91). Rend FALSE sinon.
S'DRIVING_VALUE: fonction du type de S; rend la valeur proposée pour le driver du processus (équivalent) ou sous-programme courant. Crée une erreur s'il n'y en a pas (déconnexion, ou port de mode **in** par exemple).
[AMS] S'RAMP[(TR[,TF])])] *: quantité du type de S; sa valeur suit celle de S mais avec une rampe définit par les temps de montée et de descente donnés. Si TF omis, TF=TR. Si TF et TR sont omis, un break implicite est notifié au noyau.*
*[**AMS] S'SLEW[(SR[,SF])])]** : pareil que 'RAMP mais SR et SF sont des pentes*

7.1 Synthèse

Un signal apparaissant dans un modèle destiné à la synthèse peut avoir plusieurs destins :

- Il s'agit d'un signal interne (déclaré localement) et rien ne dit qu'il « existera » sur la matériel final ; en effet les optimisations peuvent le faire disparaître, et le « mapping » sur une bibliothèque de composants disponibles pour la synthèse peut complètement changer l'organisation du modèle à fonctionnalité égale. Toutefois une inspection de la façon dont ce signal est mis à jour peut donner l'intention du concepteur si la synthèse est naïve, ou faite à la main: s'il n'y a pas d'utilisation des

attributs sur événements ('EVENT, 'STABLE ou les fonctions qui s'en servent) et si le mot-clé « **unaffected** » n'apparaît pas dans les équations flots-de-données le concernant, on peut penser qu'il s'agit d'une connexion reliée à de la logique combinatoire. Sinon, c'est plutôt un registre.

- Il s'agit d'un port de mode **in** ou **out** : son destin sera probablement celui d'un registre d'entrée ou de sortie.
- Il s'agit d'un port de mode **inout** ou **buffer**: ce sera sans doute un fil ou une nappe de fils suivant le type, donc une connexion.

7.2 Simulation

Le signal sert d'unique[13] canal de communication entre les processus, et les processus sont, après transformation de toutes les instructions en leur équivalent (voir §9.1 page 67), l'unique instruction ayant « un comportement » défini.
Le signal a une structure complexe dont voici la forme générale dans le cas d'un type résolu (cas du STD_LOGIC) ; dans cette illustration, on suppose que le signal est affecté en trois endroits du modèle, qu'à l'instant courant les pilotes proposent une valeur nouvelle (pilote du haut) et deux valeurs anciennes (les deux pilotes du bas.) Dans 10 ns, il y aura deux valeurs nouvelles (en bas) et la valeur du haut n'aura pas changé. Dans 30 ns, seule la valeur du milieu aura changé. Chaque fois qu'une des trois valeurs change, la fonction de résolution est appelée par le simulateur avec les trois valeurs disponibles (*driving values*) (anciennes et nouvelles) et elle calcule et rend la valeur unique qui sera la valeur dite « réelle » (*actual value*), celle qu'on lit quand on met le signal à droite d'une affectation.

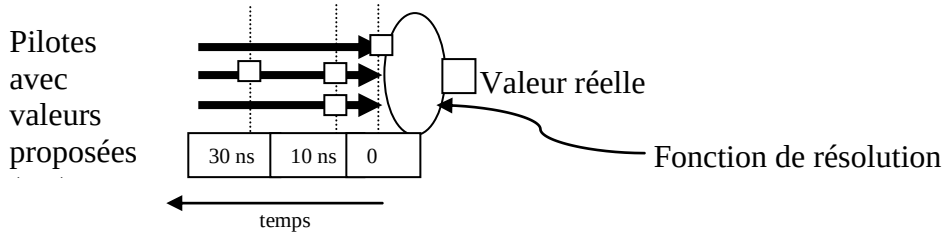


Figure 5 Un signal avec trois pilotes et leurs contributions, sa fonction de résolution et sa valeur réelle.

- Les pilotes (*drivers*) sont des files de valeurs qui contiennent les valeurs *datées* proposées ici et là pour le signal (*driving values*), lorsqu'il est à gauche d'une affectation ; dans l'instruction `X <= S ;` c'est un des pilotes de X qui va être modifié avec la valeur réelle de S pour l'instant 0. si on écrit ensuite `X <= S after 10 ns ;` dans un environnement concurrent (flot-de-données), c'est un autre pilote qui va recevoir la valeur mais avec le décalage 10 ns.
- La fonction de résolution est une fonction écrite en VHDL qui prend en entrée un vecteur (tableau à une dimension) de valeurs proposées par les différents pilotes pour l'instant courant, et calcule la valeur réelle qui sera donc celle qui sera lue si on la demande.
  Dans le cas particulier où le signal est de genre **bus** ou **register**, la contribution d'un pilote peut être supprimée de la liste des contributions à la fonction de résolution. Voir signaux gardés §11.8.2 page 93 et blocs gardés §11.8.1 page 92.

[13] À deux exceptions près : les variables partagées, voir §4.6 page 25. Et les fichiers s'ils sont utilisés de façon malicieuse, voir §5.5 page 39.

- La valeur réelle (*actual value*) est la valeur que l'on lit quand le signal est à droite d'une affectation ; dans l'instruction `x <= s ;` c'est la valeur réelle de S qui va être utilisée.

Si le type n'est pas résolu (cas du type BIT) la structure est plus simple : il n'y a qu'un pilote et pas de fonction de résolution. La valeur du pilote qui se présente à l'instant courant est celle qui va devenir valeur réelle.
VHDL est construit de telle sorte que le compilateur puisse déterminer, au plus tard juste avant la simulation (élaboration), quels sont les processus qui écrivent sur un signal et quels sont ceux qui le lisent. Même s'il y a des appels de procédures impliqués –écriture via un port de mode **out** ou **inout**- ou des signaux globaux (dans des paquetages). À ce moment là, il se crée **un pilote par signal et par processus** : chaque processus ou processus équivalent (voir page 9.1) écrivant sur un signal se voit attribuer un pilote vers ce signal et un seul, même s'il écrit plusieurs fois. Tous les processus qui lisent un signal lisent par contre **la même valeur réelle**.
Attention : le pilote est créé à l'instant zéro de la simulation avec une valeur dedans (valeur initiale ou la valeur par défaut du type, il y en a toujours une). Donc même dans un cas pathologique où l'affectation de signal serait « emballée » par un test toujours faux (voir exemple), il y aurait quand même un pilote et une valeur proposée par le processus pour ce signal. Ceci est une source constante de soucis pour les concepteurs qui oublient de mettre explicitement 'Z' (haute impédance) sur un signal avant même (croient-ils) de s'en servir.
Exemple, supposant que S est du type STD_LOGIC :

```
process
begin          Le process contribue à S avec la valeur 'U'
…              dès le début
if 1 = 2 then – test toujours faux
 S<= '0'; -- l'affectation crée
end if;    -- néanmoins le pilote
…
end process ;
```

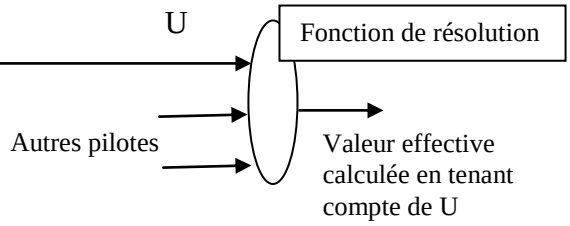


Figure 6 Un processus contribue à un signal dès lors qu'une affectation y apparaît.

Le jeu des affectations de signal pendant la simulation implique deux activités :

*7.3 Délais : l'édition des pilotes (*driver editing*)*

Comme il y a un pilote par signal et par processus, il peut arriver qu'un processus écrive sur le même signal deux valeurs potentiellement contradictoires :

```
begin
…
S <= '1' ;
S <= '0' ;
…
end process ;
```

Ce qui peut prendre une forme plus compliquée en jouant avec le temps:

```
begin
…
S <= '1' after 10 ns ;
wait for 5 ns;
S<= '0' after 5 ns;
…
end process ;
```

Il peut arriver également qu'après avoir assigné une valeur à un certain délai, le processus en assigne une autre à un délai plus proche.

```
begin
…
S <= '1' after 10 ns ;
…
S <= '0' after 5 ns ;
…
end process ;
```

La règle assez commode est que "c'est le plus proche qui gagne". Toute valeur proposée à un temps T efface les valeurs différentes proposées pour des délais supérieurs ou égaux à T, avec un incrément correspondant à l'inertie (voir §7.4 page 56). Dans les trois exemples ci-dessus, 1/ S vaudra '0' au delta suivant 2/S vaudra '0' dans 5 ns et 3/ S vaudra '0' dans 5 ns et ne vaudra pas '1' ensuite.
La règle parle des valeurs « différentes ». Le fait qu'on n'efface pas les valeurs identiques n'aura aucune conséquence pour la simulation tant qu'on ne s'intéresse pas aux transactions (affectation sans changement de valeur) via les attributs qui les gèrent. Ce qui n'est pas d'un usage très courant.

7.4 Inertie : rejection, transport

Il y a deux façons de parler du temps quand on spécifie du matériel.

- Soit on spécifie effectivement un temps, par exemple dans une ligne à retard, et si l'on dit 64 ms c'est que 63 ou 65 ne font pas l'affaire. C'est le délai spécifié.
- Soit on décrit le comportement d'un élément de bibliothèque et ses délais capacitifs, et si l'on dit 6 ns, on serait probablement plus heureux de pouvoir dire 3 ns. C'est le délai subi.

Spécifier un délai (ligne à retard) est une chose que l'on fait rarement ou alors à des niveaux systèmes. Par contre introduire dans un modèle les délais capacitifs après la synthèse –la rétro-annotation- est une activité courante. Dans ce cas là, VHDL gère le fait qu'il y a inertie, et en plus de propager les signaux avec le délai demandé, le simulateur va effacer les « glitchs » inférieurs à un temps que l'on peut spécifier.
Exemple :

```
S <= reject 2 ns A and B after 5 ns ;
```

Dans ce cas, le résultat de A **and** B va être propagé sur S avec un retard de 5 ns, mais s'il y a des mouvements plus rapides que 2 ns, ils seront éliminés, par un des mécanismes de l'édition de pilotes (voir §7.3 page 55).
Quand la clause **reject** est omise, le comportement par défaut est que la réjection se fera avec le même temps que celui spécifié dans la clause **after**, la première s'il y en a plusieurs.

```
S <= valeur1 after temps1, valeur2 after temps2 ;
```

équivalent à

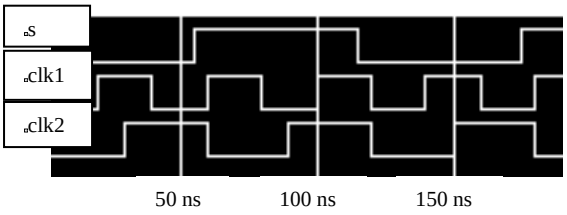
```
S <= reject temps1 valeur1 after temps1, valeur2 after temps2 ;
```

Si enfin on veut spécifier un délai (ligne à retard) et inhiber la réjection, on utilisera le mot-clé **transport** (équivalent à **reject** 0 ns):

```
S <= transport A and B after 5 ns ;
```

Exemple typique qui peut surprendre si l'on oublie l'inertie, car c'est le mode **par défaut**:

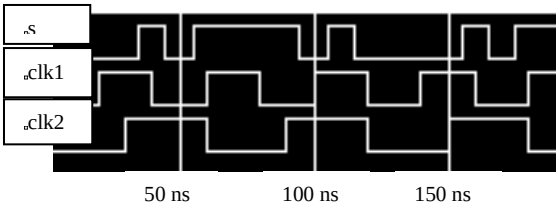
```
entity horloges is
  port (S:out bit);
end;
architecture inerte of horloges is
  signal CK1,CK2: BIT;
begin
  CK1 <= not CK1 after 20 ns; -- une horloge de période 40 ns
  CK2 <= not CK2 after 30 ns; -- une horologe de période 60 ns
  S <= CK1 xor CK2 after 15 ns;
end architecture inerte;
```



Les 15 ns de l'affectation à S vont raboter tous les pulses de 10ns venant du calcul du **xor**.

Si par contre on rajoute le mot-clé transport, les pulses seront transmis:

```
architecture transp of horloges is
  signal CK1,CK2: BIT;
begin
  CK1 <= not CK1 after 20 ns; -- une horloge de période 40 ns
  CK2 <= not CK2 after 30 ns; -- une horloge de période 60 ns
  S <= transport CK1 xor CK2 after 15 ns;
end architecture transp;
```



Ici S transmet fidèlement le résultat du calcul décalé de 15 ns.

7.5 La résolution de conflit

7.5.1 Cas du paquetage STD_LOGIC_1164

Le type défini dans le paquetage en question a neuf états.

- U : le premier, 'U', signifie « *uninitialized* » ; c'est la valeur par défaut de tous les objets de ce type (VHDL dit que la valeur par défaut est pour les signaux, variables et constantes[14] toujours la valeur « la plus à gauche » du type par exemple -2^{31} pour un entier sur 32 bits, '0' pour BIT et FALSE pour BOOLEAN). Son intérêt essentiel réside dans le fait que, si l'on trouve des signaux à 'U' après quelques heures de simulation, il y a probablement un défaut de couverture dans le test ou une mauvaise initialisation des circuits.
- 0 1 X et L H W : deux types logiques « fort » et « faible » sont symbolisés par '0' et '1' (états forts) et 'L' pour *low*, 'H' pour *high* (états faibles). Si on se restreint aux valeurs 0 et 1, ou aux valeurs L et H, toute la logique conventionnelle marche parfaitement : 'L' **and** 'H' rend 'L', par exemple, de même que '0' **and** '1' rend '0'. Si l'on fait un conflit

[14] Pour les quantités, la valeur par défaut est 0.0.

entre '0' et '1', la fonction de résolution rendra 'X' :conflit fort. Si l'on fait un conflit entre 'L' et 'H', la fonction rendra 'W' (weak-conflict) : conflit faible. Tout l'intérêt de cette duplication est évidemment qu'un conflit entre une valeur forte et une valeur faible rendra la valeur de la valeur forte, avec une force « forte ». Ainsi 'W' *contre* '0' rend '0'. Ceci permet de modéliser des systèmes au niveau *switch* ou à drain ouvert, par exemple. Le paquetage STD_LOGIC_1164 définit ainsi la résolution entre toutes les combinaisons de ces neufs valeurs, ainsi que toute la logique et les opérateurs correspondants. .Une inspection du corps de paquetage correspondant (dans la bibliothèque IEEE de votre simulateur, ou sur Internet, ce serait trop gros pour ce manuel) montrera que certaines combinaisons ont un sens, d'autres…moins (que rendre si on fait un **and** entre 'Z' et 'W' ?). Néanmoins le paquetage doit évidemment rendre une valeur pour toute combinaison et il le fait par des tables de vérités qu'on peut consulter ; au concepteur de ne pas faire de bêtises.

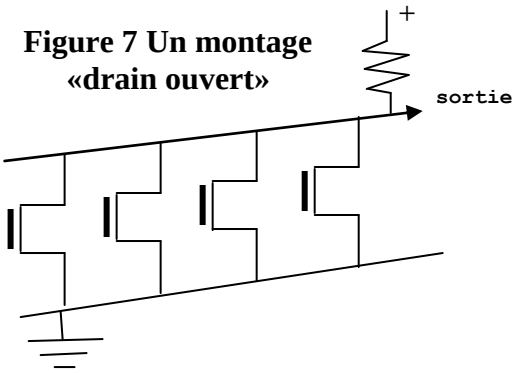
- Z : la valeur 'Z' est la haute impédance. Tout conflit entre 'Z' et une autre valeur rend l'autre valeur.
- - : *don't care*. La valeur '-' ne sert qu'en synthèse (de même que 'U', 'X', et 'W' ne servent qu'en simulation). Cette valeur permet au concepteur de ne pas sur-spécifier son modèle et permet de dire que tel signal, à tel moment, peut prendre n'importe quelle valeur que le synthétiseur jugera optimale. Par exemple, dans certains codes opérations, il arrive souvent que des bits soient inutilisés. Les spécifier d'autorité à 0 ou à 1 obligerait le synthétiseur à faire de la logique inutile. Mettre « don't care » lui donne l'opportunité d'optimiser.

7.5.2 Cas général

Le concepteur « ordinaire » qui ne se sert quasiment que du type STD_LOGIC venant du paquetage STD_LOGIC_1164 (voir §16.1.3 page 118) se sert de ce fait de la fonction de résolution qui s'y trouve et, tel monsieur Jourdain, fait de la résolution sans le savoir. Depuis la standardisation du paquetage STD_LOGIC_1164, rares sont les concepteurs qui ont écrit leur propre fonction de résolution. Ce qui suit est donc destiné au curieux qui veut savoir « comment ça marche » et au concepteur chevronné ou concepteur système qui veut écrire ses propres fonctions de résolution. Ou, évidemment, au malheureux qui doit maintenir un modèle écrit avant le paquetage STD_LOGIC_1164.

Chaque processus écrivant sur un signal passe donc par ce tuyau qu'est le pilote. Pour calculer la valeur réelle du signal à l'instant courant, le simulateur prend toutes les valeurs proposées par les différents pilotes si l'une d'elles a changé (événement), en fait un vecteur (un tableau à une dimension) et passe ce vecteur à une fonction *ad hoc* écrite en VHDL –qui doit être **pure** (cf §6.2.2.1 page 47) et mentionnée dans le type du signal. La fonction est écrite pour rendre une valeur du type, et c'est cette valeur qui devient la valeur réelle du signal à l'instant donné.

Figure 7 Un montage «drain ouvert»



Par exemple, si, en partant du type BIT, nous voulons modéliser la résolution « drain ouvert » (où il suffit d'une contribution à '0' pour que le résultat soit '0' (et donc il faut que toutes les contributions soient à '1' pour que le résultat soit '1' : c'est le ***et câblé :*** figure ci-contre.

1. On declare quelque part (déclaration de paquetage) le sous-type résolu et sa fonction de résolution (pure) :
```
function fdrainouvert(arg : bit_vector) return bit ;
subtype bit_drain_ouvert is fdrainouvert bit ;
```
2. On écrit dans le corps (body) le comportement espéré pour la resolution:
```
function fdrainouvert(arg : bit_vector) return bit is
begin
 for I in arg'range loop
   if arg(i)='0' then return '0'; end if;
 end loop;
return '1';
end function fdrainouvert;
```
3. On peut se servir du sous-type résolu :
```
signal S : bit_drain_ouvert;
…
S <= valeur1 ;
…
S <= valeur2 ;
…
S <= valeur3 ;
```

On voit ci-dessus que l'écriture de cette résolution se fait par la définition d'un sous-type spécifique, qui récupère donc les valeurs et les opérations du type parent, ici BIT, qui est compatible avec le type BIT, mais a la fonctionnalité supplémentaire –la fameuse contrainte des sous-types, cf. §5.6 page 40- que, si un signal est de ce type, toutes les contributions de ses différents pilotes passent par la fonction de résolution, laquelle fournit la valeur effective du signal.

On notera aussi que la fonction de résolution n'a aucun sémantique matérielle : c'est un algorithme censé représenter un phénomène physique qui contribue à la simulation, ce n'est pas la description d'un matériel à construire ou synthétiser. Au niveau système on peut aussi écrire des fonctions de résolution très abstraites, par exemple gérant la collision de paquets sur un bus.

Cette fonctionnalité est tout à fait spécifique à VHDL. Il s'agit de dire au simulateur d'appeler une fonction écrite par l'utilisateur sous son seul contrôle, quand il y a affectation d'un signal multi-affecté. La fonction de résolution n'est en principe jamais appelée explicitement dans un modèle, elle est simplement mentionnée comme attribut d'un type.

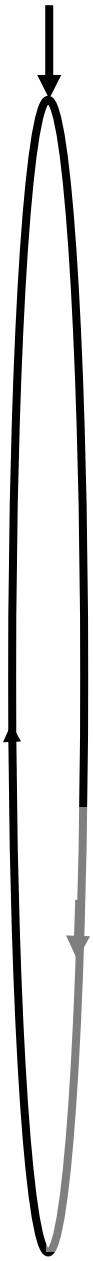
Maux de tête et complications : Il est évidemment possible d'utiliser un sous-type résolu dans un type composite de genre « record », les résolutions s'appliqueront logiquement au niveau où elles sont déclarées.

On peut ensuite déclarer un sous-type de ce « record » et le résoudre à son tour. En cas de confit les fonctions de résolutions seront appelées en cascade et dans un ordre parfaitement défini dans le LRM (manuel de référence). Là où la chose se corse, c'est qu'il peut y avoir conflit sur un seul des champs de l'enregistrement (**record**), qui se trouve donc *partiellement* en conflit. Cette question ne peut intéresser aucun concepteur sain d'esprit, les autres trouveront leur bonheur dans le LRM, voir références chapitre 19 page 169.

8 Le Cycle de Simulation

Le cycle de simulation en VHDL comporte quelques phases distinctes. D'abord toutes les instructions concurrentes sont ramenées à leurs processus équivalents (voir §9.1 page 67). Le *delta* est le cycle élémentaire décrit ci-après.

8.1 Le Cycle



0. **Initialisation** : Les objets sont initialisés, et les attributs prédéfinis, tous les processus sont réveillés et exécutés, les **postponed** en dernier. *[AMS] Le noyau analogique cherche un état d'équilibre (DOMAIN vaut QUIESCENT_DOMAIN, voir 8.3.1page 63) en fonction des valeurs données dans les **break**.* Le temps est mis à 0.0.
1. *[**AMS]** le noyau analogique est exécuté.*
2. **Tous les signaux sont mis à jour** : le simulateur procède à l'édition des drivers (voir §7.3 page 55) qui contiennent des valeurs tout juste déposées ou déposées précédemment avec un délai, aux calculs de résolutions de conflits (voir §7.5 page 57) et à la mise à jour des valeurs effectives des signaux. À la fin, les signaux ont une valeur effective, pas forcément stable pour le temps courant mais qui sera lue par la suite si le signal apparaît à droite d'une affectation.
3. **Réveil des processus** : tous les processus non marqués **postponed** sensibles à un des signaux qui vient de changer sont «réveillés » et sont exécutés dans un ordre quelconque et en principe inobservable (sauf utilisation malicieuse des fichiers ou des variables partagées). Pendant leur exécution, ils affectent des drivers de signaux pour un temps futur, même s'il ne s'agit que de un *delta*: les valeurs effectives des signaux ne sont pas modifiables et ne bougent pas. Ils finissent tous (ils doivent) par tomber sur une instruction **wait**, sinon c'est une erreur grave de conception et le simulateur s'arrête.
4. **Le prochain point de simulation digitale est déterminé**, comme étant le plus proche « où il doit se passer quelque chose » (résultats de clause **after**, **wait for**). Si ce point est au temps courant (résultat d'une affectation de signal à temps zéro, instruction **break**), ce sera un nouveau cycle delta. *[AMS] Le signal DOMAIN est positionné sur TIME_DOMAIN ou FREQUENCY_DOMAIN si on n'est pas dans le cycle d'initialisation.*
5. **Réveil des processus retardés** : si et seulement si le prochain point de simulation marqué sur l'échéancier n'est pas au temps courant (ce ne sera pas un nouveau *delta*) le simulateur exécute tous les processus retardés marqués **postponed** qui sont réveillés, soit qu'ils l'aient été au début des cycles courants, soit qu'ils l'aient été pendant. Chaque processus retardé ne doit s'exécuter qu'une fois et ne doit pas re-déclencher les autres processus : ils ne doivent pas créer de nouveau *delta* ce qui serait contraire à l'hypothèse de leur exécution. Ce sont donc des processus passifs –n'affectant pas de signaux- ou qui affectent des signaux avec un délai non nul -alors le prochain point de simulation est recalculé, voir point 4-. Après leur exécution, la série de cycles *delta* du temps courant est terminée. Tous les signaux ont une valeur effective (*effective value*) qui apparaît sur les chronogrammes et correspond à un état « réel » du circuit simulé.
6. Le simulateur avance le temps si les cycles delta sont finis et si c'est un modèle digital. *[AMS] Si c'est un modèle mixte VHDL-AMS, le temps avancera par pas à la discrétion du solveur numérique pendant l'étape 1.* Puis **retour en 1**

Figure 8 Cycle de simulation

8.2 Le temps en VHDL

Plusieurs « sortes » de temps existent en VHDL :

- Le temps que vivra l'objet qu'on simule : c'est celui qui apparaît dans les chronogrammes. Celui-ci a deux aspects :
  - Le temps du monde digital, marqué par des événements qui s'inscrivent dans l'avenir au fil de la simulation, comme on inscrit des rendez-vous dans un agenda.
  - Le temps du monde analogique, fait de points de résolution décidés à la discrétion du solveur (sauf clause **limit**, voir §13.6 page 109).
- Le temps delta, signifiant la causalité, dans le monde digital : la sortie apparaît « après » l'entrée, même si le temps spécifié est nul.
- Le temps que dure la simulation, qui en général est beaucoup plus long que le premier mais parfois plus court (simulation du système Terre-Lune en analogique, par exemple).

Dans le monde digital, la simulation se fait par événements : comme on le voit dans le cycle de simulation (§8.1 ci-dessus) les processus s'activent, affectent des signaux qui à leur tour activent des processus au delta suivant, etc. jusqu'à ce que rien ne reste actif pour le temps de simulation courant. Pendant cette activité, et pendant les précédentes, des choses ont été inscrites pour l'avenir dans l'échéancier du simulateur. Celui-ci avance donc le temps à la prochaine date où quelque chose a été inscrit, ou où il y a forcément un événement (on affecte un signal qui change de valeur) ou au moins une transaction (on affecte un signal mais il ne change pas de valeur). Et ça recommence. Bien noter qu'avancer dans le temps est gratuit en termes de ressources, tout le travail se fait sur des points de temps fixes.

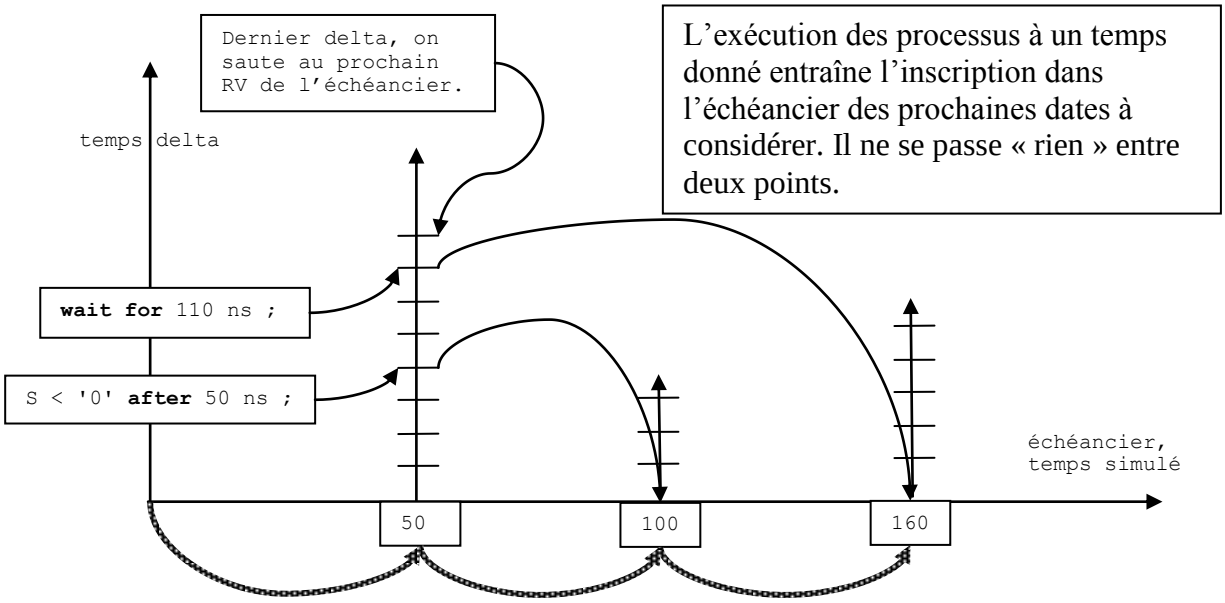


Figure 9 Le temps en VHDL digital pur

Dans le monde analogique pur, il n'y a pas de temps delta : le solveur décide à sa guise des points de résolution, en tenant compte des différents paramètres, limites et précision demandée, etc.

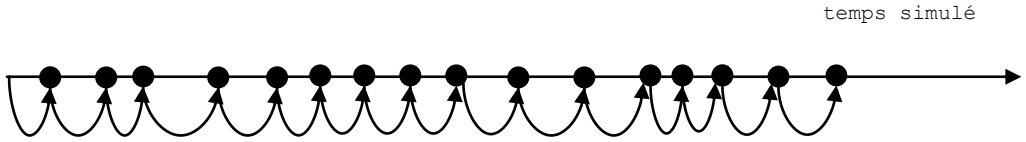


Figure 10 Le temps en VHDL analogique pur

Dans le monde mixte, les deux simulateurs se passent la main respectivement : le simulateur analogique dont les points de calculs ne sont pas nécessairement à intervalles réguliers « s'arrange » pour tomber sur les points d'intérêt du simulateur numérique, au besoin en faisant des allers-retours quand les événements numériques sont créés par le passage d'un seuil analogique et que donc le simulateur numérique n'a pas pu l'anticiper : voir à ce sujet la gestion de l'attribut ABOVE, §8.3.2 ci-dessous.

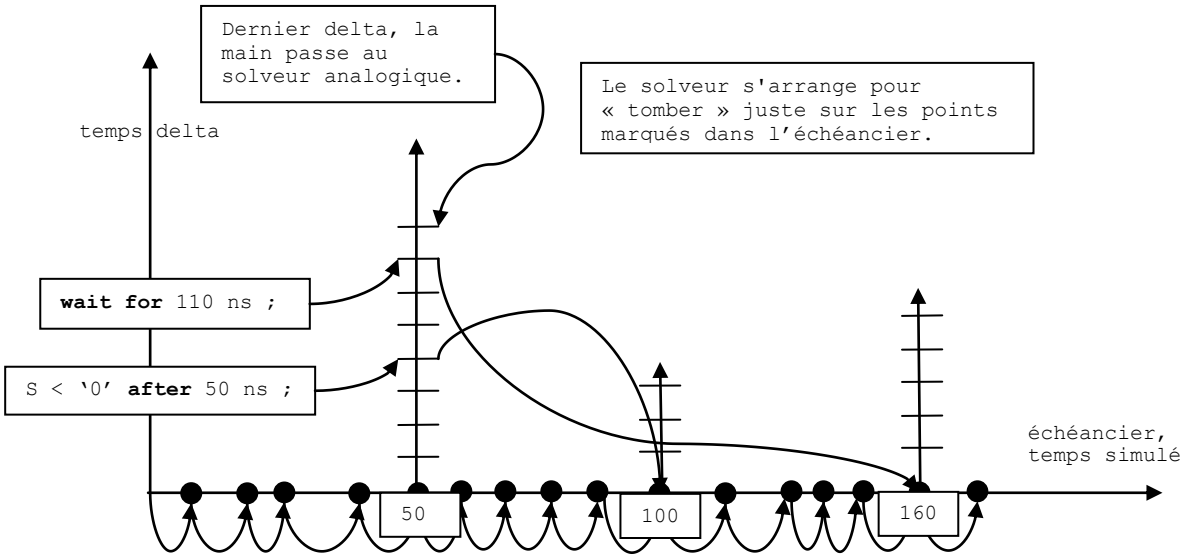


Figure 11 Le temps en VHDL mixte

8.3 [AMS]Interactions entre analogique et numérique

*Contrairement au monde digital, pour lequel un simulateur canonique est décrit dans le LRM, le monde analogique ne définit pas d'algorithme de résolution. Tout ce qu'on en sait, c'est que le calcul se fait par pas (les ASP, Analog Solution Point) à la discrétion du noyau et que ces pas ne sont pas forcément réguliers. Le concepteur a le moyen de forcer certains de ces points (instruction **break**) ou de majorer leur distance (clause **limit**).*

8.3.1 Domain

Un signal particulier, prédéfini, s'appelle DOMAIN et peut prendre une des valeurs d'un type énuméré: QUIESCENT_DOMAIN, TIME_DOMAIN, FREQUENCY_DOMAIN. Ce signal est mis à jour pendant et après la phase d'initialisation (le tout premier cycle de stabilisation). Il peut être utilisé dans le modèle pour rendre des résultats pertinents selon le mode d'analyse. Le fait que ce soit un signal permet de rendre des processus sensibles dessus. Par exemple pour inhiber la simulation digitale quand on passe au FREQUENCY_DOMAIN, ou encore pour distinguer le cas de la recherche du point de fonctionnement continu, et le cas de l'analyse temporelle.

```
process (……,DOMAIN)
begin
  case DOMAIN is
    when QUIESCENT_DOMAIN => actions spécifiques à la recherche
                             du point de stabilisation DC,
                             au temps 0.
    when TIME_DOMAIN => au dernier delta du temps 0, actions
                        spécifiques à l'initialisation du
                        modèle pour l'analyse temporelle
    when FREQUENCY_DOMAIN => idem, pour l'analyse fréquentielle
  end case;
end process ;
```

À noter que certains attributs n'ont de sens que dans un domaine ou dans l'autre: par exemple les transformées de Laplace obtenue par l'attribut 'LTF ou la transformée en Z obtenue par l'attribut ZTF.

8.3.2 ABOVE

Quand le modèle est mixte (VHDL-AMS), l'avance au temps défini peut être écourtée si d'aventure une des quantités dépasse pendant la phase analogique un seuil fixé (threshold) par la grâce d'un attribut 'ABOVE, provoquant alors un événement sur un signal; événement qui n'était pas encore dans les tuyaux et donc n'était pas prévu par le noyau numérique.

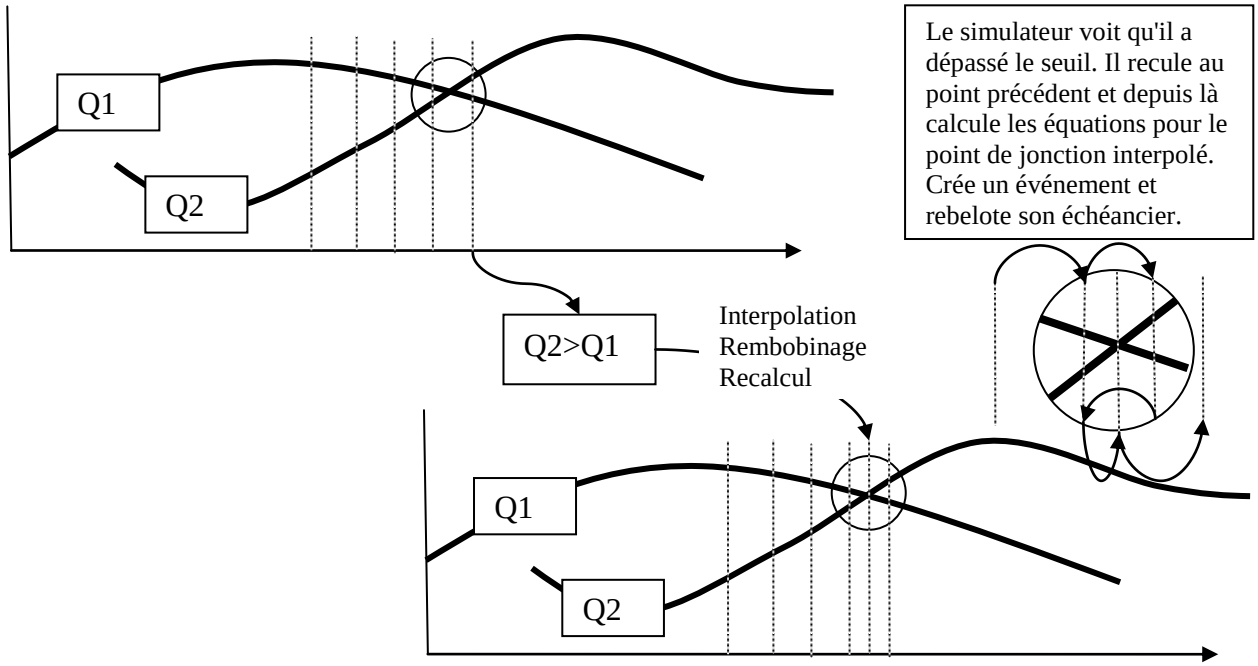


Figure 12 Gestion de ABOVE dans le cycle de simulation

Dans ce cas, le temps de la discontinuité est interpolé (rappelons que le simulateur analogique avance par pas et remarque le dépassement d'une valeur parce que l'un des pas est «avant» et l'autre «après»). Cette interpolation est assez exacte puisque dans la fenêtre forcément étroite qui nous intéresse (le zoom de la figure) les courbes peuvent être assimilées à des segments de droites.

Le temps de simulation est donc rembobiné au temps de calcul antérieur et les valeurs des quantités rétablies dans leur état de ce temps là –ce qui suppose que le simulateur les a conservées d'une façon ou d'une autre, ou qu'il a les moyens de les recalculer- puis avancé juste au temps supposé de la discontinuité qui peut donc être facilement interpolé linéairement avec très peu d'erreur.
Alors le signal implicite 'ABOVE est mis à jour, se trouve être en situation d'événement et les processus qui sont sensibles dessus seront réveillés au prochain cycle.

8.3.3 Break

Symétriquement, un événement sur un signal peut être utilisé pour forcer le simulateur analogique à prendre ce temps là comme point de calcul avec éventuellement de nouvelles conditions localement « initiales ». Ceci est utile quand le modèle présente une discontinuité: modèle de la balle qui rebondit (la vitesse V devient subitement –V) ou modèle de multiplexeur analogique (la sortie qui était branchée sur l'entrée E1 devient brusquement branchée sur l'entrée E2). En effet il n'existe pas d'algorithme connu à ce jour qui permettrait de faire en sorte de notifier automatiquement le noyau dans toutes les conditions baroques qui peuvent survenir. C'est l'instruction concurrente ou séquentielle ***break on*** *qui va servir :*

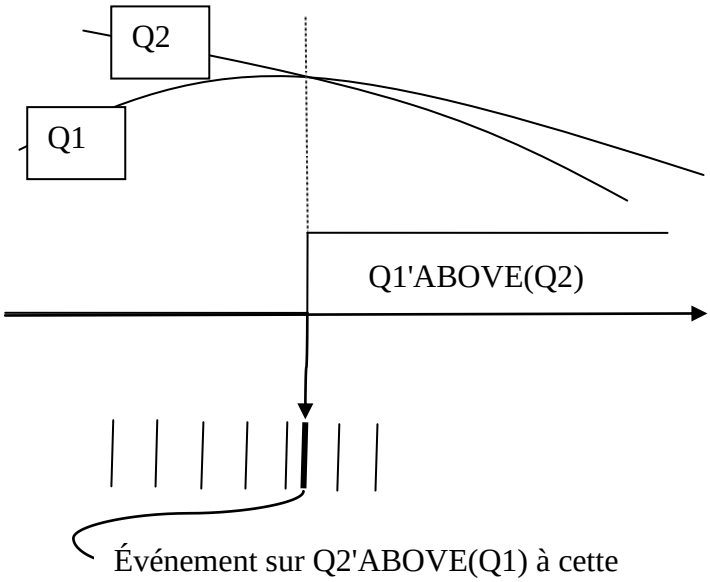


Figure 13 break sur événement digital, à partir de 'ABOVE

```
-- Contexte concurrent, voir équivalences §9.1 page 67
break on S; -- force un point de calcul quand il y a événement sur S
break on Q1'ABOVE(Q2); -- idem quand Q1 passe au dessus de Q2
```

Dans ses versions diverses, l'instruction break permet de spécifier une condition:

```
break on S when condition;
```

S'il y a une condition mais pas de liste de signaux, le break sera activé à chaque événement sur un des signaux de la condition.

Elle permet aussi de forcer des conditions initiales pour le solveur:

```
break q1 => valeur1, q2=>valeur2;
```

(ici pas de liste de sensibilité, donc exécution une seule fois au début)

Les conditions spécifiées sur la quantité de sélection (ici Q) remplacent la condition initiale sur la quantité de break (ici Q aussi); par défaut les conditions initiales sont Q'DOT=0 pour la recherche de l'état de départ (si le signal DOMAIN vaut QUIESCENT_DOMAIN), ou la condition par défaut Q = Q(-t) pour la réinitialisation après discontinuité (si DOMAIN vaut TIME_DOMAIN). Voir §8.3.1 ci-dessus pour le signal DOMAIN.

La forme complète est:

```
break for Q1 use Q2 => expression on S when condition;
```

Ici Q1 est la quantité de sélection (celle dont on remplace la condition initiale Q1'DOT=0), Q2 la quantité de break, pour laquelle on donne une valeur initiale, équation qui remplacera Q1'DOT=0.
*Si la quantité de sélection est omise, par défaut Q1 est identique à Q2 et sa dérivée doit apparaître dans le modèle: soit **parce** qu'elle est une quantité dont la dérivée (Q2'DOT) est explicitement dans le modèle, soit parce que c'est la quantité implicite de l'attribut INTEG: Q'INTEG, Q étant évidement dans le modèle.*
*Certains attributs font un **break** implicite quand on les utilise : RAMP et SLEW, il est donc inutile d'en mentionner un explicite.*

Exemple : la balle qui rebondit (tiré de [CHR] page 169):

```
entity bouncer is
end entity bouncer;


architecture ball of bouncer is
  quantity v: real tolerance "velocity"; -- vitesse
  quantity z: real tolerance "elevation"; --hauteur au dessus du sol
  constant g: real := 9.81; -- gravité
  constant air_res: real := 0.001; -- résistance de l'air, unités m-1
begin
   z'dot == v;
   if v>0.0 use
      v'dot == -g - (v**2)*air_res;
   else
      v'dot == -g + (v**2) * air_res;
   end use;
   break v=> -v when not z'above(0.0); -- lors du rebond
   break v=> 0.0, z => 10.0; -- conditions initiales (pas de when)
end architecture ball;
```

9 Les Équivalences

Bien des constructions de VHDL n'ont de définition que par équivalence à une autre. En simulation (voir §9.1 ci-dessous), c'est le comportement qui est ainsi défini par équivalence à une seule construction (le processus). Le processus est défini lui-même comme l'exécution d'un programme séquentiel, au sens informatique du terme. Le processus est *in fine* la seule instruction de VHDL digital qui ait « un comportement » défini.
Pour ce qui est de la construction hiérarchique (voir §9.2 ci-dessous), c'est tout le modèle défini en termes de composants configurés, après la définition des arguments génériques et le déroulement des instructions de génération de code (**generate**) qui se retrouve éclaté en blocs interconnectés, eux-mêmes contenant des instructions éventuellement transformées pour la simulation en leurs processus équivalents.
Enfin, de façon plus anecdotique, quelques constructions de VHDL sont juste des facilités ou des raccourcis d'autres constructions (§9.3 ci-dessous).

9.1 Simulation

Pour ce qui est de la simulation, l'ensemble d'un modèle peut de façon ultime se résoudre par équivalences en une mer de processus communiquant par des signaux.
Exemple simple : l'instruction flot-de-données, concurrente :

```
S<= A and B ;
```

…a pour équivalent le processus suivant :

```
process
begin
  S<= A and B ;    -- on affecte un pilote de S avec A and B
  wait on A,B ;    -- on attend un événement(changement de valeur)
                   -- sur A ou sur B.
end process ;      -- on revient au début (process = boucle infinie)
```

Et la seule affectation de signal qui soit sémantiquement définie est celle qui est dans le processus, à sémantique séquentielle (on l'exécute chaque fois qu'on « passe dessus ». L'autre (la première, celle qui est concurrente) n'a de sémantique que par équivalence. Ainsi on verra souvent, ici et dans le LRM, décrire la sémantique de simulation d'une instruction par la seule description de son processus équivalent. Il est donc important de bien comprendre comment marche le cycle de simulation, le déroulement des processus et le flux des données dans les signaux. Voici en un tableau d'exemples les équivalences des instructions concurrentes :

***Affectation concurrente de signal** (voir §11.2 page 87 et suivantes): le processus équivalent est la transformation de l'affectation en son équivalent (instruction **if** ou **case**) suivie d'un **wait** sur l'ensemble des signaux apparaissant à droite de l'affectation (y compris dans les calculs de délais et dans le sélecteur).*

`label : S <= expression ;`	`label: process`
	`begin`
	`  S <= expression ;`
	`  wait on S1, S2 , ... ;`
	`end process label;`

`label: S<= expr1 when condition1`
`  else expr2 when condition2`
`  else unaffected ;`
	`label: process`
	`begin`
	`  if condition1 then S<=expr1;`
	`  elsif condition2 then S<=expr2;`
	`  end if;`
	`  wait on S1, S2 , ... ;`
	`end process label;`

`label: with selecteur select`
`  S <= expression1 when valeur1,`
`  expression2 when valeur2,`
`  expression3 when others;`
	`label: process`
	`begin`
	`  case selecteur is`
	`  when valeur1 => S<=expression1;`
	`  when valeur2 => S<=expression2;`
	`  when others => S<=expression3;`
	`  end case;`
	`  wait on S1, S2 , ... ;`
	`end process label;`

***Affectations gardées:** Si les affectations sont gardées (voir §11.6.2 page 91) le processus équivalent inclut un test sur le signal implicite GUARD avant de reprendre une des trois transformations ci-dessus. GUARD participe à la liste de sensibilité. Si le signal est **bus** ou **register** (voir §11.8.2 page 93), la branche **else** de ce test comprend une clause de déconnexion (S <= **null**;) sinon il n'y a pas de branche **else**. Nous donnons ici un seul exemple de transformation (premier cas), les autres s'en dérivent facilement :*

`label: S <= guarded S1+S2 ;`	`label: process`
	`begin`
	`  if GUARD`
	`  then S <= S1+S2 ;`
	`  [else S <= null;`
	`  si S est bus ou register`
	`  voir page 93]`
	`  end if;`
	`  wait on GUARD, S1, S2 ;`
	`end process label;`

***Appel concurrent de procedure** (voir §11.8.2.1.2 page 94): le processus équivalent contient le même appel de procédure suivi d'un **wait** sur l'ensemble des signaux apparaissant dans ses arguments de mode **in** ou **inout**.*

`label:proc(V,S1,S2);`	`label: process`
	`begin`
	`  proc(V,S1,S2);`
	`  wait on S1, S2;`
	`end process label;`

*Le processus avec **wait** implicite est traduit en un processus avec **wait on** explicite, mis à la fin du code et sur les mêmes signaux.*

`label: process(S1,S2)`
`begin`
`…`
`end process label;`
	`label: process`
	`begin`
	`…`
	`wait on S1,S2;`
	`end process label;`

Assert concurrent *(voir §11.8.2.1.1 page 93): le processus équivalent contient le même* ***assert*** *suivi d'un* ***wait*** *sur l'ensemble des signaux apparaissant dans la condition. À noter qu'il n'y a pas d'instruction* ***report*** *concurrente, faute de condition.*

```
label: assert S1=S2
       report "message"
       severity ERROR;
```

```
label: process
begin
   assert S1=S2
   report "message"
   severity ERROR;
 wait on S1, S2;
end process label;
```

[AMS] BREAK ON (VOIR §13.5 PAGE 109): LE PROCESSUS EQUIVALENT CONTIENT LA MEME INSTRUCTION BREAK HORMIS LA BRANCHE «ON», SUIVI D'UN WAIT SUR L'ENSEMBLE DES SIGNAUX APPARAISSANT DANS LA LISTE OU, EN SON ABSENCE ET SEULEMENT DANS CE CAS, DANS LA CONDITION.

```
LABEL1: BREAK FOR Q1
       USE Q2 => 0.0
       ON S1, S2
       WHEN A>B;


LABEL2: BREAK FOR Q1
       USE Q2 => 0.0
       WHEN A>B;
```

```
LABEL1: process
begin
   BREAK FOR Q1
         USE Q2 => 0.0
         WHEN A>B;
   WAIT ON S1, S2;
end process LABEL;

LABEL2: process
begin
   BREAK FOR Q1
         USE Q2 => 0.0
         WHEN A>B;
   WAIT ON A,B;
end process LABEL;
```

[AMS] L'INSTRUCTION PROCEDURAL (§13.3 PAGE 107) EST DEFINIE COMME UNE EQUIVALENCE A UNE INSTRUCTION SIMULTANEE SIMPLE (UNE EQUATION) DONT L'UNE DES BRANCHES EST L'AGREGAT DES QUANTITES AFFECTEES, ET L'AUTRE EST UN APPEL DE FONCTION DONT LE CODE EST LA COPIE DE CELUI DU PROCEDURAL, ET LES ARGUMENTS SONT LA LISTE DE TOUTES LES QUANTITES APPARAISSANT DANS LE CODE. ICI UN EXEMPLE OUTRAGEUSEMENT SIMPLIFIE.

```
BEGIN – DE L'ARCHITECTURE
…
LBL: PROCEDURAL IS
BEGIN
 Q1 := …Q3…
 Q2 := …Q4…
END PROCEDURAL;
```

même code

```
FUNCTION F_LBL(
        QQ1,QQ2,QQ3,QQ4:REAL)
RETURN REAL_VECTOR IS
  VARIABLE Q1:REAL:=QQ1;
  VARIABLE Q2:REAL:=QQ2;
  VARIABLE Q3:REAL:=QQ3;
  VARIABLE Q4:REAL:=QQ4;
BEGIN
  Q1 := …Q3…
  Q2 := …Q4…
  RETURN (Q1,Q2);
END FUNCTION F_LBL;
…
BEGIN – DE L'ARCHITECTURE
…
LBL:(Q1,Q2)==F_LBL(Q1,Q2,Q3,Q4);
```

9.2 Construction hiérarchique

La construction hiérarchique dans un langage de description de matériel a ceci de particulier, par rapport à un langage informatique, que les blocs de code « factorisés » (les sous-programmes en informatique, les instances de composants avec leur configuration en VHDL)

sont de nature très différente : un sous-programme (avec ses variables) n'existe qu'une fois quand on l'appelle, s'il n'y a pas de récursivité, et pas du tout si on ne l'appelle pas. Un composant instancié et configuré va « exister » tout au long de la simulation, dès l'instant zéro, avec son état propre et les valeurs de ses signaux internes. S'il est n-pliqué, il existera N fois avec N états internes. En ce sens, le composant configuré est plutôt similaire à une expansion de macro en C qu'à un appel de sous-programme.
Cette expansion se fait avant la simulation : pendant l'élaboration ; et elle peut elle-même être contrôlée en VHDL : c'est l'objet des instructions de génération : **generate**. On peut ainsi écrire *du code qui génère le code* qui sera finalement exécuté. Ce qui est très utile pour les blocs réguliers ou pseudo réguliers (réguliers avec quelques exceptions).
De plus, certaines constantes du modèle peuvent être données au dernier moment, juste avant de simuler (cas de l'entité tête de hiérarchie) ou d'instancier (cas des entités utilisées dans la hiérarchie). C'est la généricité, qui permet dans la zone marquée **generic** de nommer des constantes dont on ne donne pas la valeur au moment de la compilation. Plus fort, ces constantes différées peuvent être utilisées comme arguments des blocs **generate**, permettant ainsi d'écrire *du code qui produit du code* généré au dernier moment.
Toutes ces transformations s'appliquent évidemment successivement et transitivement jusqu'à ce qu'il ne reste plus rien à transformer.

*Les arguments génériques (voir §14.1 page 111) sont purement et simplement remplacés par leur valeur, y compris s'ils sont utilisés dans des instructions de génération (**generate**).*

```
generic (D:TIME)
…
S <= valeur after D;
```

```
...generic map (5 ns)….
…
S <= valeur after 5 ns;
```

```
generic (N:INTEGER);
…
lbl: for I in 1 to N generate….
```

```
…generic map (128)…
…
lbl: for I in 1 to 128 generate…
```

*Les instructions de génération (**generate,** voir §14.2 page 112**)** créent le code qui sera ensuite soumis à d'autres transformations puis simulé ou synthétisé.*

```
lbl : for I in 1 to 3 generate
  lbl2: if I < 3 generate
     aff: X(I) <= Y(I+1);
  end generate;
inst: comp port map(A(I),A(I+1));
end generate;
```

```
lbl(1).lbl2.aff : X(1) <= Y(2);
lbl(1).inst: comp port map (A(1),A(2));
lbl(2).lbl2.aff : X(2) <= Y(3);
lbl(2).inst: comp port map (A(2),A(3));
lbl(3).inst: comp port map (A(3),A(4));
```

*Les instances de composants avec leur configuration (voir §10.8 et suivants page 79) sont remplacées par le code, éclaté sur place, des entités/architectures appelées. Pour cela, une hiérarchie de blocs (**block**) est construite : un niveau pour le composant, un niveau pour l'entité. À la fin toute factorisation de code a disparu (le code est éclaté autant de fois que nécessaire, ou du moins tout de passe comme si).*

Supposons le contexte suivant:

```
entity E is (P1:BIT;P2:out BIT); end E;
architecture A of E is
-- declarations de l'architecture
begin
-- ici code de l'architecture
end;
```

Dans une autre architecture:

```
component comp (X:BIT;Y:out BIT); end component;
```

Instanciation de composant:
`for C:comp use entity work.e(a);`
…
`C : comp port map (A,B) ;`
	`C:block port (X,Y) port map (A,B);`
	`begin`
	`E : block port (P1,P2) port map (X,Y) ;`
	`-- declarations de l'architecture`
	`begin`
	`-- ici code de l'architecture`
	`end block E;`
	`end block C;`

Instanciation directe:
`C: entity WORK.E(A)`
`port map (A,B);`
	`C:block begin`
	`E : block port (P1,P2) port map (A,B);`
	`-- declarations de l'architecture`
	`begin`
	`-- ici code de l'architecture`
	`end block E;`
	`end block C;`

Noter que, lorsqu'un label est calculé (expansion d'un **generate**) , le nom indexé obtenu ne pourrait pas être écrit directement en VHDL, le texte équivalent n'est donc pas exactement « pur VHDL ». Mais c'est la seule exception.

9.3 Facilités

Quelques constructions sont en fait de simples raccourcis ou facilités. Ou peuvent être vues comme telles.

Pour le lexique (voir chapitre 2 page 11), quelques substitutions permettant de gérer les claviers anciens. Ces équivalence ont été déclarées périmées et vont disparaître lors d'une prochaine révision.

`%abc%`
`%abc"`
`%abc"def%`
`"abs%def"`
	`"abc"`
	interdit
	interdit
	autorisé

`:123:E4` `:123#E4`	`#123#E4` interdit
`case 2 ! 3 => …`	`case 2 | 3 => …`

Pour le **wait**:

wait until condition ;
Supposant que la condition contient des références aux signaux S1 et S2.
	loop
	  wait on S1,S2…
	  exit when condition ;
	end loop ;

wait on S3, S4 until condition ;
Ici les signaux apparaissant dans la condition ne sont pas pris en compte.
	loop
	  wait on S3,S4;
	  exit when condition ;
	end loop ;

10 Hiérarchie: Composants, Maps, etc.

Il est important de pouvoir décrire un système comme une interconnexion de sous-systèmes. C'est ainsi qu'une armoire informatique sera définie comme l'interconnexion de tiroirs, eux-mêmes définis par un ensemble de cartes enfichées, chaque carte étant un assemblage de composants discrets, eux-mêmes étant décrits à partir de blocs existants, etc etc.
Cette décomposition hiérarchique permet :

- La décomposition du travail entre équipes.
- La sous-traitance d'une partie clairement délimitée.
- La réutilisation de code (on écrit et on teste une fois, on appelle plusieurs fois).
- L'utilisation de bibliothèques.

On peut penser à première vue qu'il n'y a là rien que de très habituel en informatique, avec la notion de sous-programme. En fait, si effectivement on retrouve la nécessité de factoriser du code (écrire une fois du code qui sera utilisé plusieurs fois), la grande spécificité des langages décrivant du matériel est que chaque instance du code ainsi factorisé a sa vie propre pendant la simulation, il ne s'agit pas d'objets « appelés », il s'agit d'objets « instanciés ». Et donc l'utilisation du code factorisé s'apparente bien plus à l'expansion d'une macro –ou à le *inline* de C et Ada) qu'à un appel de sous-programme. Il faut bien comprendre cette différence, car VHDL possède *aussi* la notion de sous-programme, et même de sous-programmes concurrents, voir §11.8.2.1.2 page 94.

Attributs *la hiérarchie fait l'objet d'attributs (§6.2.6 page 50) permettant de retrouver dans une chaîne de caractères le nom des instances de façon à faire des traces ou des messages significatifs.*

- `E'SIMPLE_NAME string, le nom de l'objet E`
- `E'INSTANCE_NAME string, le nom hiérarchique de l'objet E`
- `E'PATH_NAME string, le nom du chemin hiérarchique qui mène à E`

Déclaration:	**Déclaration d'entité et d'architecture**

`procedure P (N :integer) is`
`  variable V:integer;`
`begin`
`  …code de P`
`end P;`
	`entity E is`
	`  port (N:integer);`
	`end ;`
	`architecture A of E is`
	`  signal S:integer;`
	`begin`
	`  Code de A`
	`end;`

Séquence d'appels:	**Instances (syntaxe instanciation directe):**
`P(X);` `P(2);`	`C1: entity E(A) port map (X) ;` `C2: entity E(A) port map (Y) ;`
Exécution :	**Code équivalent créé (avant exécution):**

`appel de P`
`exécution de P pour N = X`
`V existe`
`Retour, V n'existe plus.`
`appel de P`
`exécution de P pour N = 2`
`V existe`
`Retour, V n'existe plus.`
	`Copie du code de A avec N <= X`
	`C1.S existe individuellement`
	`Copie du code de A avec N <= Y`
	`C2.S existe individuellement`
	L'exécution concurrente va donc traiter tout le code expansé. Tout ce qui est créé existe pendant toute la simulation.

Figure 14 Comparaison sous-programme - instance de composant

10.1 Associations

L'association consiste à associer un argument réel à un argument formel. Par exemple, si la fonction sin(X) est définie, l'appel avec sin(4) associe 4 à X.
Il y a quatre endroits où l'on déclare des listes d'arguments formels :

- La déclaration de sous-programme (**procedure**, **function**)
- La déclaration d'entité (**generic**, **port**)
- La déclaration de composant (**generic**, **port**)
- Le bloc (**block**) d'usage anecdotique par les concepteurs, quoiqu'il soit massivement utilisé dans le LRM pour décrire la sémantique des autres constructions (voir §9.2 page 69).

Il y a quatre endroits en VHDL où l'on fait des associations :

- L'appel de sous-programme.
- L'instance de composant (association d'arguments génériques et de ports)
- La configuration (association d'arguments génériques et de ports)
- Le bloc (**block**).

Dans ces quatre situations, les règles sont les mêmes à quelques détails près dépendant de la nature des objets passés et de celle des choses appelées (composant, procédure…).

10.2 La partie formelle

Exemples de listes d'arguments formels (ici soulignées) :

```
procedure P(constant X: INTEGER; signal Z in BIT; variable T: out INTEGER);
function F (X,Y: INTEGER) return INTEGER;
component C
    port(V:BIT VECTOR(7 downto 0);S: out BIT; quantity Q: in REAL);
end component;
entity E
     generic (N :INTEGER) ;
     port (A,B: BIT VECTOR (N-1 downto 0):=(others=>'1'); C: out BIT ) ;
end;
```

La partie formelle se présente sous la forme d'une parenthèse encadrant une liste de déclarations séparées par des points-virgules (… `A : BIT ; B : out INTEGER ; …`) . Quand plusieurs noms partagent exactement la même déclaration, on peut les factoriser en les séparant par des virgules : `A, B, C : BIT ;`

- Chaque item a un nom et un type, c'est le minimum à mettre explicitement.
  Ex. : (…A : BIT ;…)
- Chaque item a un mode : **in**, **out**, **inout**, **buffer**[15]. Ne rien mettre équivaut à **in** (défaut).
  Ex. : (…A : **out** BIT ;…). Les arguments de classe **file** n'ont pas de mode. Seuls les arguments de classe **signal** peuvent être **buffer**.
- Chaque item a une classe : **variable**, **signal**, **constant**, **file,** ***[AMS] quantity, terminal***.
  Ne rien mettre équivaut à **signal** pour les composants, les blocs et les entités (*[AMS] avec **quantity** et **terminal*** seuls genres autorisé dans ces cas, et **signal** est le défaut qu'on omet souvent si le modèle est uniquement digital ; ne rien mettre équivaut à mettre **constant** pour les sous-programmes qui admettent aussi les autres classes – la fonction n'admet pas la classe variable -.
  Ex.: (…**variable** A : **out** INTEGER ; **signal** B : **in** BIT ;…)
- Certains items peuvent avoir une valeur par défaut, celle qui sera fournie en l'absence d'association.
  Ex ;: `(…A : in BIT := '0' ;…)`

Certaines combinaisons sont incompatibles, par exemple **buffer** suppose que le paramètre est un **signal**, cf. ci-dessus.

10.3 L'association

L'association se fait dans les «map»: **generic map**, **port map**; ainsi que dans les appels de sous-programmes. On verra le détail du fonctionnement de la généricité §14.1 page 111.

Exemples d'associations reprenant les exemples ci-dessus:

```
P( 23, '0', I) ; -- appel de procédure
INST : C port map (V => vect, S=> clk, Q => QT) ;
       -- instanciation de composant
INST2: entity work.E(A) generic map (2) port map (VECT1, open, S);
       -- instanciation directe
for all : COMP use work.E(A) generic map (N=>2) port map (VECT1, VECT2, S);
-- configuration
```

[15] Un mode **linkage** existe, complètement inutilisé et qui n'est à retenir que parce que c'est un mot réservé.

Positionnelle *vs* nommée : pour désigner quel objet réel va sur quel argument formel, il y a deux solutions plus un mélange des deux:

- L'association par position consiste à mettre les arguments réels dans l'ordre des arguments formels. Le premier sera associé au premier, le second au second, etc. C'est le mode d'association classique des langages de programmation comme C.
  Ex. : …`port map (S1,S2,S3) ;`
- L'association par nom consiste à écrire l'argument formel et le réel pour chaque item : `argument_formel=>objet_réel` , ce qui libère de la contrainte d'ordre ; d'une part cette forme est beaucoup plus claire pour le lecteur, et de ce fait quasiment obligatoire dans les sociétés qui font du VHDL ; d'autre part elle devient nécessaire si l'on veut profiter de certaines fonctionnalités décrites ci-après, comme l'association de sous-élément.
  Ex. : …`port map (X=>S3, Y=>S1, Z=>S2) ;`
- Il est possible de mélanger les deux modes, en commençant par du positionnel, et en finissant par du nommé. Ex. : …`port map (S1,Y=>S3,Z=>S2) ;`…

Ce choix entre association positionnelle et association nommée se retrouve dans la description d'agrégat, voir §5.3.5 page 38.

Scalarité des associations : Même si l'on peut associer « en vrac » des structures composites arbitrairement complexes `(vecteur_formel => vecteur_reel),` à chaque élément scalaire de la liste des « formels » doit *in fine* correspondre un argument scalaire « réel », ou un mot clé indiquant une absence : **open**, ou enfin une valeur par défaut préétablie dans la liste des formels si le mode s'y prête.
Ceci (scalarité) a une implication qui peut être très utile : on peut associer les éléments d'un objet d'un type composite séparément.
Si la déclaration est : `(…X : BIT_VECTOR (3 downto 0) ;…)` l'association correspondante pourra être : `(…X(0) => S1, X(1)=>S2, X(3 downto 2)=> V(20 to 21)…)` où l'on suppose que S1 et S2 sont de type BIT, et V un objet de type BIT_VECTOR dont on prend une tranche.
Évidemment comme rappelé ci-dessus on peut aussi associer « en bloc » des types composites s'ils sont identiques ou compatibles. Le fait que tout soit ramené à des scalaires fait que l'association de tableaux exige des types identiques ou des sous-types compatibles, *mais pas des indices ni des directions identiques*, tant qu'il y a un objet réel par objet formel.
Exemple: Si V1 est un formel BIT_VECTOR (2 **to** 4) et V2 un signal BIT_VECTOR (6 **downto** 4), on peut associer l'un à l'autre, l'élément 6 de V2 sera associé au 2 de V1, etc.
Le type doit impérativement correspondre exactement entre l'argument formel et l'argument réel d'une même association (même type ou sous-types compatibles, mais dans le cas du sous-type pas forcément indices ou direction identique, voir ci-dessus).
Toutefois, si l'on veut associer des objets de types différents mais pour lesquels il existe des fonctions de conversion, implicites ou définies par l'utilisateur, on peut « emballer » l'argument réel ou formel ou les deux par les fonctions correspondantes. Ces fonctions seront appelées au fil de la simulation et selon les hasards de la simulation par événement. Par exemple, si I est un entier, et R un réel et que l'association est **inout**, on écrira :

```
port map (…REAL(I) => INTEGER(R) ,…)
```

Si l'association est **in**, le flux ne pouvant qu'être entrant, on écrira :

```
port map (…I => INTEGER(R) ,…)
```

Si l'association est out, ce sera l'inverse :

```
port map (…REAL(I) => R ,…)
```

À noter que ceci fonctionne aussi sur les associations de sous-éléments scalaires si l'objet est composite (un tableau). À noter surtout que cette fonctionnalité n'a été introduite en VHDL que pour permettre l'interopérabilité de modèles écrits avec des types logiques différents, et que l'intérêt de la chose a singulièrement rétréci depuis la standardisation de STD_LOGIC_1164 ; d'autant que ces fonctions de conversion participent à la résolution de la surcharge et que cela conduit parfois à des imbroglios invraisemblables. Cette fonctionnalité n'est disponible pour les signaux que dans les **port map** donc pas pour les passages de signaux aux procédures. En effet dans une procédure, l'affectation de signal «tape» directement dans le signal passé en argument, il n'y a pas de signal-port faisant tampon, donc pas d'accroche pour une éventuelle conversion.

La classe (**variable**, **signal**, **constant, file,** ***[AMS] quantity, terminal***) doit être compatible (ce qui ne veut pas dire identique). Un port de mode **in** peut être appelé avec une expression, et de l'intérieur on verra l'objet de la classe déclarée –signal, quantité- affecté par cette valeur; hormis ce cas de figure, la question ne peut se poser que dans le cas d'appels de sous-programmes et les règles sont assez intuitives :

- Si un sous-programme déclare un signal, une variable *[AMS] ou une quantité* comme argument formel, on ne peut lui passer qu'un signal ou respectivement une variable *[AMS] ou une quantité* comme argument réel. En effet la procédure « de l'intérieur » a accès aux attributs spécifiques de l'objet et il faut bien qu'ils existent ! Rappelons que les fonctions n'acceptent pas la classe variable.
- Si le sous-programme veut une constante, on peut lui passer constante, variable, signal *[AMS] ou quantité*: Il verra en tant que constante la valeur de l'objet au moment de l'appel.
- Dans le cas du fichier (**file**) on ne peut passer qu'un fichier.

Le mode (**in**, **out**, **inout**, **buffer, linkage**) doit être compatible avec les arguments présentés. Là aussi, les restrictions sont plutôt intuitives : le mode **in** demande une valeur donc éventuellement une expression calculée comme dans sin(X+3) ; si on y passe un objet, c'est sa valeur qui sera transmise. Les modes **out**, **inout** et **buffer** demandent des objets nommés puisqu'ils sont susceptibles d'être écrits. **linkage** ne sert à rien. La sémantique de ces modes est la suivante :

- **in** : l'objet peut être lu dans la structure concernée. Il peut être associé à un autre objet de mode **in** d'une structure plus interne.
- **out** : l'objet peut être affecté dans la structure concernée. Il peut être associé à un autre objet de mode **out** d'une structure plus interne.
- **inout** : l'objet peut être lu ou affecté dans la structure concernée. Il peut être associé à un autre objet de mode **inout**, mais aussi à un objet de mode **in** ou un objet de mode **out** d'une structure plus interne.
- **buffer** : ne peut être qu'un port de genre signal. Le port peut être lu ou affecté comme ci-dessus. Néanmoins le système garantit (par une erreur juste avant la simulation, le cas échéant) que dans tout le modèle le signal correspondant a un seul pilote (*driver*). Si on veut le connecter à d'autres ports, il ne peut être connecté de l'extérieur qu'à un autre port de mode **buffer**, et de l'intérieur qu'à **buffer** ou **in**. Rarement utilisé. La révision de 2001 autorise maintenant **buffer** à être associé à des ports de mode **out** ou **inout**.
- **linkage** : inutilisé. Fut mis là pour des raisons qui sont perdues, probablement pour donner une poignée vers un éventuel simulateur mixte (digital-analogique), chose qui

a été faite autrement depuis avec VHDL-AMS. Ce mode va sans doute disparaître lors d'une prochaine révision.

Absence d'association : si un argument formel a une valeur par défaut et qu'il est de mode **in**, il est possible de ne pas l'associer : la valeur par défaut sera retenue. Il est aussi possible de dire explicitement que tel port n'est pas associé en mettant le mot-clé **open** à la place de l'argument réel. Dans ce cas, si c'est un formel de mode **in** ou **inout**, la valeur lue à l'intérieur sera celle par défaut (explicite ou implicite). Si c'est un formel de mode **out**, il sera juste ignoré.
Lors de l'appel de procédures, les signaux ne peuvent avoir de valeur par défaut.

10.4 Ports et port map

Un port est un signal, un terminal ou une quantité qui se trouve être aussi partie de l'interface de la structure concernée. On trouve des ports

- Sur des entités
- Sur des composants
- Sur des blocs

Les ports, comme tous les objets valués, ont un type qui peut être quelconque sauf les types accès et fichiers, ni tout type composite en contenant. Ce type peut être résolu.
Le **port map** consiste à brancher les ports sur les signaux désignés dans le §10.3 ci-dessus. Ce branchement se fait dans le cas des signaux par le mécanisme de pilotes (drivers) décrit dans le chapitre 7 page 53. En particulier, si un port de mode **out** est connecté à un signal «à l'extérieur», il créera une contribution pour ce signal même si «à l'intérieur», le port est laissé en l'air. dans ce cas le concepteur peut être surpris de voir débarquer une valeur 'U' (cas de STD_LOGIC) alors que, *in fine*, le port n'est pas connecté à un signal. Dans le cas des terminaux, le branchement est « électrique » : les lois de Kirchhoff s'appliquent à toute l'équipotentielle définie par les branchements.

10.5 Generic et generic map

Un argument générique est vu comme une constante nommée dans le modèle qui s'en sert. Le seul détail est que… sa valeur n'est pas connue au moment de la compilation. On verra le détail du fonctionnement de la généricité §14.1 page 111. Il n'y a guère de mystère dans le fonctionnement des génériques: la valeur proposée est remplacée dans le modèle partout où l'argument correspondant apparaît. Le seul souci qu'on puisse avoir, c'est avec les arcanes de la prise en compte des valeurs par défaut: si le composant configuré par une entité a une valeur par défaut différente de celle de l'entité. La meilleure attitude à avoir est d'éviter ce cas là -les valeurs par défaut dans les arguments de composants- qui, bien que documenté (c'est l'entité ou la configuration qui gagne) est illisible et trompeur pour l'utilisateur.

10.6 Arguments et appels de sous-programmes

Les sous-programmes ont ceci de particulier que les arguments formels et réels peuvent ne pas être de la même classe, selon les règles vues §10.3 ci-dessus. Par exemple, un signal passé à un sous-programme qui attend une constante, sera vu dans le sous-programme comme une constante ayant la valeur du signal.

10.7 Blocs(block)

Le bloc est une structure qui permet d'ouvrir une parenthèse dans une région d'instructions concurrentes (une architecture ou un autre bloc, après le **begin**). Cela correspond à peu près à

tracer un rectangle plus ou moins arbitraire sur le plan d'un circuit, à déclarer que tous les fils qui traversent ses frontières sont ses ports, et à décréter que l'intérieur du rectangle est le plan de ce nouveau sous-circuit. L'intérêt de cette seule fonctionnalité est mince pour le concepteur, mais par contre grande pour la définition du langage: en effet toutes les opérations hiérarchiques d'instanciation et de configurations sont définies en termes d'équivalences avec des blocs imbriqués et connectés. Voir §9.2 page 69 pour les détails.

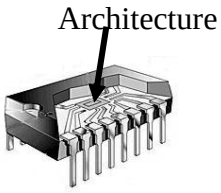
Le bloc a donc à la fois une déclaration de ports (qui seront visibles à l'intérieur) et une association (**port map**) avec l'environnement extérieur. De même, il a à la fois la déclaration d'arguments génériques et leur **generic map**.

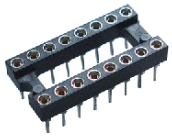
```
label: block
  generic (…)
  generic map (…)
  port (…)
  port map (…)
begin
  -- ici instructions concurrentes.
end block label;
```

10.8 Le composant (component)

Le composant est le support de la connexion. Le mot composant est une traduction hasardeuse de **component** qui lui-même n'est pas très bien choisi. On pourrait dure «support» ou «empreinte».

De même qu'en VHDL on sépare nettement la déclaration d'entité de ses architectures, de même au moment de s'en servir (de l'appeler), on sépare nettement l'utilisation de la vue externe (instance, connectique) de l'appel de son architecture (configuration). C'est en ce sens qu'on peut parler de support: sur une plaque de circuit imprimé, on va souder les supports, leur connecter toutes les pistes nécessaires et ensuite seulement brancher le circuit intégré dessus.





Dans cette allégorie, le support est l'instance de composant, et le circuit intégré est l'entité avec son architecture à l'intérieur; le branchement est la configuration. La référence au catalogue de supports étant déclaration de composant, si l'on veut tirer la comparaison à l'extrême.

Cette fonctionnalité n'a pas d'équivalent dans les langages informatiques et non plus dans les langages concurrents de VHDL. Si le lecteur est familier de C, il peut se représenter cette fonctionnalité comme lorsque, en C, on compile tous les modules d'un programme en incluant les «.h» nécessaires, et on choisit ensuite quel «.o» on va associer à chaque «.h» pour l'édition de liens. On le fait par exemple pour passer d'une version DEBUG à NODEBUG, ou d'une version qui trace des variables à une version définitive, ou encore pour changer la gestion de la mémoire, etc.

Ce qui est possible en C et d'autres langages en jouant avec les *makefile*, l'est en VHDL dans le langage.

10.8.1 Déclaration

BNF (voir définition page 9) :

```
component_declaration ::=
  component identifier [ is ]
              [ local_generic_clause ]
              [ local_port_clause ]
  end component [ component_simple_name ] ;
```

Le mot-clé **is** est optionnel pour une raison de compatibilité avec les modèles écrits avant 1993.

La déclaration de composant se place dans une zone déclarative d'une architecture, ou d'un bloc, ou dans une déclaration de paquetage. Elle est souvent la copie d'une déclaration d'entité pour ce qui est de ses arguments génériques et ports, en effet la fonction du composant est de recevoir une entité configurée.

La liste des arguments génériques est une liste d'items séparés par des points virgules, dont chacun est de la forme:
…NOM: [**in**] TYPE,… (**in** est le seul mode possible).
On peut mettre une valeur par défaut: N : INTEGER := 3;
On peut aussi regrouper en les séparant par des virgules des paramètres génériques qui ont exactement les mêmes propriétés: …N, M:INTEGER :=8;…

La liste des ports est une liste d'items séparés par des points virgules, donc chacun est de la forme:
…classe NOM: mode TYPE;

- classe est à prendre parmi **quantity, terminal** et **signal** (le défaut est **signal**).
- mode est à prendre parmi **in out inout buffer (**le défaut est **in** et **buffer** ne va que pour un signal).

On peut aussi mettre une valeur par défaut si c'est pertinent (mode **in**) et regrouper des objets ayant exactement les mêmes caractéristiques en séparant leurs noms par des virgules.
Exemple:

```
component COMP is
  generic (N: INTEGER:=8);
  port (S: BIT_VECTOR(N-1 downto 0);
        quantity Q: out REAL);
end component COMP;
```

10.8.2 Instanciation de composant

BNF (voir définition page 9) :

```
component_instantiation_statement ::=
  instantiation_label :
              instantiated_unit
                          [ generic_map_aspect ]
                          [ port_map_aspect ] ;
```

L'instanciation de composant se fait dans une zone d'instruction d'une architecture ou d'un bloc (après le **begin**) d'où la déclaration est visible.

Son effet est de créer effectivement le composant à partir de sa déclaration, de lui donner les éventuels arguments génériques dont il a besoin et de connecter ses ports à des signaux déclarés.

Exemples (voir déclaration en §10.8.1 ci-dessus):

```
C1: COMP generic map (5) port map (BUS5, Q1);
C2: COMP port map (BUS8, Q1); -- la valeur par défaut sera prise pour N.
C3: COMP port map ("00100001", Q1);  -- passage d'une constante
```

Les **map** peuvent être assez complexes, on consultera le §10.3 page 75 (l'association) pour le détail des possibilités.

10.8.3 Configuration

La configuration consiste à «brancher» les entités elles-mêmes configurées sur les composants. Pour cela il faut associer à chaque port du composant un port de l'entité, ou une valeur par défaut, ou une indication **open**.
Il y a quatre façons de configurer:

- la solution «propre» consiste à le faire de l'extérieur dans une unité de conception dédiée qui se compile en tant que telle, la déclaration de configuration (détaillé ci-dessous §10.8.3.4 et page 21).
- La solution «économique» consiste à le faire dans la même architecture que celle où l'on a instancié le composant (§10.8.3.3 ci-dessous).
- La solution «trop facile» consiste à instancier le composant directement configuré (§10.8.3.2 ci-dessous).
- La solution «triviale» consiste à ne pas le faire du tout et à compter sur la configuration par défaut (§10.8.3.1 ci-dessous).

Dans les trois derniers cas, le langage offre une multitude de combinaisons dans les branchements des ports les uns aux autres : voir §10.3 page 75 (l'association). Chaque fois qu'on connecte un jeu de ports sur un jeu de signaux, il est possible de permuter l'ordre (en utilisant l'association nommée), de laisser des ports ouverts (mot-clé **open**) et de connecter individuellement des signaux scalaires sur des ports de types composites. On peut aussi changer le type des ports «au vol». Ceci correspond assez bien à la réalité du travail du concepteur de cartes, qui, au moment de chercher dans les tiroirs un circuit à mettre dans le support, va trouver «presque le même» mais dont les pattes sont dans un ordre différent ou qui marche en 3V au lieu de 5V. Et il va être obligé de permuter ou de couper des pattes, ou d'insérer un convertisseur.

Sans préjuger de cas particuliers, on choisira plutôt la première solution en contexte de développement industriel, la seconde pour les modèles à usage interne, la troisième pour des modèles à usage unique ou pour le modèle final englobant transitivement tous les autres et la quatrième pour des exercices ou des essais. Voyons les dans l'ordre de complexité croissante.

10.8.3.1 Configuration par défaut

Lorsque le composant se trouve être absolument identique à une entité déjà compilée (même nom, mêmes génériques, mêmes ports de même mode avec les mêmes noms, la seule différence admise étant les valeurs par défaut), alors les systèmes acceptent de faire une configuration automatique par défaut. L'entité appelée est évidemment celle qui est identique, et tous les arguments génériques et ports sont associés un pour un avec ceux du composant. Le souci vient lorsqu'il y a plusieurs architectures associées à l'entité et il vaut mieux ne pas se

servir de cette facilité dans ce cas là: la norme dit que c'est la dernière architecture compilée qui sera prise et cela peut créer des surprises après des recompilations.

10.8.3.2 Instanciation directe

L'instanciation directe consiste à ne pas passer par l'«étape composant». Dans le parallèle circuit-imprimé/support/circuit intégré c'est comme si on soudait directement le circuit intégré sur le circuit imprimé en se passant de support. La syntaxe est la même que celle de l'instanciation de composant (§10.8.2 ci-dessus) mais à la place du nom du composant on peut mettre

- un couple entité/architecture: **entity** E(A) – si l'architecture est omise, la dernière compilée sera retenue.
- une configuration déjà compilée: **configuration** C(E)

Exemple:

```
C1: entity LIB.BLA(ARC) generic map (6) port map (A,B,C);
```

Pour toutes les options de câblage (les **map)** , voir §10.3 page 75 (l'association).
La révision 2001 de VHDL –peu ou pas implémentée à cette date- dit maintenant que le préfixe « LIB » n'est pas nécessaire si c'est la bibliothèque courante.

10.8.3.3 Spécifications de configuration

BNF (voir définition page 9) :

```
configuration_specification ::=
    for component_specification binding_indication ;

component_specification ::=
    instantiation_list : component_name

binding_indication ::=
    [ use entity_aspect ]
    [ generic_map_aspect ]
    [ port_map_aspect ]
```

La spécification de configuration est un item déclaratif qui se trouve dans la même zone que l'instance de composant. Il permet d'associer une entité et une architecture au composant, en spécifiant dans le détail quelle valeur ou quel objet va sur quel générique ou quel port. Si on ne spécifie pas tout dans le détail, les défauts sont:

- l'architecture compilée la dernière si on l'omet
- les génériques de même nom et type si on ne les associe pas explicitement, ou les valeurs par défaut
- les ports de même classe, nom, mode et type si on ne les associe pas explicitement, ou les valeurs par défaut quand c'est pertinent.

Inutile de dire que, si l'on joue avec les choix par défaut, le nombre de possibilités explose et avec lui la chance que l'on a de trouver une erreur le cas échéant. Il est donc sage de ne laisser les choix par défaut que dans une alternative tout ou rien: si un port doit être explicité, il vaut mieux expliciter aussi tous les autres.
On trouvera §10.3 page 75 (l'association) toutes les subtilités et complications des mécanismes d'association dans les **port map** et **generic map**.

Attention, par exception à une règle générale en VHDL, la visibilité des spécifications de configuration ne descend pas dans les blocs internes y compris ceux définis par les instructions de génération. Il faut donc répéter les spécifications de configuration dans chaque bloc où les composants sont instanciés, et dans le cas des zones **generate** il faut soit utiliser la zone déclarative quand l'implémentation le permet, soit créer un bloc *ad-hoc* juste pour avoir la zone déclarative permettant de caser la spécification de configuration.

Exemples:

```
for all :composant use work.entity E(A) ;
begin
  X: composant port map… -- sera configuré correctement
  G :for I in 1 to 10 generate
     for all :composant use work.entity E(A);
     begin
        Y: composant port map…
     end generate;
```

Utilisation de la zone déclarative du **generate**.
Y sera configuré alors que sans cela il ne l'aurait pas été.

Figure 15 Spécification de configuration dans un generate, utilisation de la zone déclarative

Dans la plupart des implémentations VHDL, la zone déclarative du bloc **generate** n'est pas disponible car c'est un ajout récent à la norme, il faut donc faire un bloc:

```
for all :composant use work.entity E(A) ;
begin
  X: composant port map… -- sera configuré correctement
  G :for I in 1 to 10 generate
    B:block
     for all :composant use work.entity E(A);
     begin
        Y: composant port map…
     end block;
  end generate;
```

block *ad hoc* pour avoir une zone déclarative et pouvoir répéter la configuration.
Y sera configuré alors que sans le **block** il ne l'aurait pas été.

Figure 16 Spécification de configuration dans un generate, utilisation du bloc

Il faut bien faire attention à cette « fonctionnalité » car il peut arriver, par malheur, que l'instance « interne » corresponde exactement à une entité compilée qui n'est pas celle désirée et que le mécanisme de configuration par défaut se mette alors en branle si on a oublié de configurer explicitement.

10.8.3.4 Déclarations de configurations

La déclaration de configuration est une unité de conception, au même titre que les déclarations d'entité, les architectures ou les deux morceaux des paquetages. On la voit à ce titre §3.2.3 page 21.

BNF (voir définition page 9) dont on donne ici seulement la tête car elle serait trop longue, voir le reste chapitre 17 page 149:

```
configuration_declaration ::=
   configuration identifier of entity_name is
               configuration_declarative_part
               block_configuration
   end [configuration] [ configuration_simple_name ] ;

configuration_declarative_item ::=
   use_clause
   | attribute_specification
   | group_declaration

block_configuration ::=
   for block_specification
               { use_clause }
               { configuration_item }
   end for;

block_specification ::=
   architecture_name
   | block_statement_label
   | generate_statement_label [ ( index_specification ) ]

configuration_item ::=
   block_configuration
   | component_configuration

component_configuration ::=
   for component_specification
               [ binding_indication ; ]
               [ block_configuration ]
   end for;
binding_indication ::=
   [ USE entity_aspect ]
   [ generic_map_aspect ]
   [ port_map_aspect ]
```

La configuration référence une entité: **configuration** CONF **of** ENT **is**…
Il peut donc y avoir plusieurs configurations pour la même entité dans la bibliothèque.

Après quoi la configuration est capable d'ouvrir transitivement tous les blocs de la hiérarchie pour y découvrir les composants instanciés dans les architectures, et les configurer avec une autre configuration ou un couple entité/architecture dont l'architecture peut être ouverte à son tour et…ainsi de suite. La déclaration de configuration peut être ainsi un fichier très long et indenté en triangles de profondeurs variables, d'une lisibilité…discutable. Dans ces cas là, les déclarations de configuration peuvent être produites automatiquement par un outil VHDL à l'usage d'un autre outil VHDL, par exemple à l'occasion d'une recette chez le client ou encore par l'utilisation d'un outil de rétro-annotation qui va introduire des délais après synthèse dans un modèle écrit en utilisant les conventions VITAL. Ces conventions associées à quelques paquetages standards permettent d'écrire des modèles « sans temps » et d'insérer les délais de l'extérieur, quand on les connaît, par une configuration ad-hoc. Vital est un standard issu de l'industrie, devenu ensuite le standard IEEE 1076.4.

Par ailleurs il est possible de faire quelques déclarations, principalement des clauses **use** (§3.1 page 18), dans les déclarations de configuration, de façon à ce que les visibilités qu'aura la

configuration en ouvrant la hiérarchie soit la même que la visibilité qu'avait l'architecture au même endroit quand elle a été compilée. Exemples:

```
configuration conf of testbench is
   for arch_test
      for COMP : comp_in_test
         use configuration work.conf1;
      end for;
   end for;
end conf;


configuration conf of E is
     for arch – une architecture de E
         for COMP1 : component1 -- un composant de arch
            use entity E1(A1);
            for A1 – l'architecture de E1
               for COMP2 : component2 – un composant de A1
               use entity E2(A2);
               end for;
            end for;
         end for;
         for COMP3 : component3 -- un autre composant de arch
            use entity E3(A3);
         end for;
      end for;
end configuration conf;
```

11 Flot de Données (*dataflow*) et Concurrence

Il s'agit de décrire un système par des équations reliant les sorties et les entrées. D'une façon assez courante, ce seront des équations *logiques* impliquant le type BIT, STD_LOGIC ou des vecteurs correspondants mais rien n'y oblige dans le langage.

11.1 Syntaxe

Les équations *flot-de-données* se mettent dans les architectures : entre le **begin** et le **end**. Il s'agit d'instructions concurrentes : **leur ordre respectif n'a pas d'importance**. Comme toutes les instructions concurrentes en VHDL, elles peuvent être labellisées (on peut préfixer par un nom suivi d'un deux points « `LBL :` » nom qu'on retrouvera probablement dans les traces de simulation) mais ce n'est pas obligatoire.
BNF (voir définition page 9) :

```
concurrent_signal_assignment_statement ::=
        [ label : ] [postponed] conditional_signal_assignment
      | [ label : ] [ postponed ] selected_signal_assignment
conditional_signal_assignment ::=
      target      <= options conditional_waveforms ;
conditional_waveforms ::=
      { waveform when condition ELSE }
      waveform [ when condition ]
options ::= [ guarded ] [ delay_mechanism ]
delay_mechanism ::=
      transport
      | [ reject  time_expression ] inertial
waveform ::=
      waveform_element { , waveform_element }
      | unaffected
waveform_element ::=
      value_expression [after time_expression ]
      | null [ after time_expression ]
```

11.2 Les affectations de signaux

Elles peuvent prendre trois formes. Les expressions affectées au signal doivent être du même type que le signal, ou alors être le mot-clé **null** indiquant une déconnexion dans certains cas (signaux gardés voir §11.8.2 page 93).

11.3 L'affectation simple

```
S <= [options] waveform_element ;
```

où waveform_element est d'une des deux formes :

- `expression`
- `expression` **`after`** `délai`

Les options sont vues §11.6 ci-dessous.
Et S est un signal qui peut, si son type est composite, être un agrégat de signaux:
Ceci est légal, supposant S1, S2 et S3 du type BIT et V du type BIT_VECTOR(0 to 2):
```
(S1,S2,S3) <= V after 5 ns;
```
À la fin, chaque signal scalaire recevra sa valeur scalaire venant de son vis-à-vis.

11.3.1 Simulation

Si un des signaux de la liste de sensibilité change (tous les signaux apparaissant à droite de l'affectation, y compris éventuellement dans l'expression de délai), l'expression est évaluée et S est affecté (ce qui signifie qu'un pilote de S est affecté et que ce pilote contribuera à S après un certain temps -délai spécifié- ou au moins après un delta).

11.3.2 Interprétation pour la synthèse

D'une façon générale, si les types manipulés sont des BITS, STD_LOGIC, entiers, vecteurs des précédents, etc… et si on n'utilise pas d'attributs de stabilité ('EVENT, 'STABLE, fonction rising_edge, etc.) la synthèse de cette instruction rendra de la **logique combinatoire** représentant l'expression logique après optimisation. Attention à ce qu'en synthèse, la spécification de délai n'a pas de sens. Si par contre on utilise des attributs comme 'EVENT, la synthèse fera intervenir des bascules et des registres.

11.4 L'affectation conditionnelle

```
S <=  [options] waveform_element1 when condition1
           else waveform_element2 when condition2
         … else waveform_element_n;
```

où waveform_element est d'une des deux formes :

- `expression`
- `expression` **`after`** `délai`

Les options sont vues §11.6 ci-dessous.
Ici aussi, S peut être un agrégat de signaux: (S1, S2) <= … est légal, pour autant que les valeurs proposées soient des tableaux à deux éléments. À la fin, chaque signal scalaire recevra sa valeur scalaire venant de son vis-à-vis.

La dernière expression peut être le mot-clé **`unaffected`**[16] et elle ne peut pas être conditionnée. Autrement dit, si un des signaux de la liste de sensibilité change, une *au plus* des expressions sera affectée à S, ou aucune si c'est **`unaffected`** qui est choisi. On comparera utilement cette instruction avec le **if** du monde comportemental, instruction qui est utilisée pour le processus équivalent (voir §9.1 page 67).

[16] Dans les anciennes versions de VHDL, **`unaffected`** n'existe pas. On le remplace par la répétition du signal affecté, ce qui revient *grosso-modo* au même : `S<= expression` **`when`** `condition` **`else`** `S` ; la différence étant qu'il faut mettre S à droite de l'affectation, il ne peut donc pas être un port de mode **out**.

11.4.1 Transformation pour la simulation

```
process
begin
  if condition1
    then S<= waveform_element1;
  elsif condition2
    then S<= waveform_element2;
  … else S<= waveform_element_n;
  end if ;
wait on (signaux apparaissant dans les expressions et les conditions)
end process ;
```

où `waveform_element` est d'une des deux formes :

- `expression`
- `expression after délai`
-

Ici encore, S peut être un agrégat de signaux: Exemple: (S1, S2) <= … est légal, pour autant que les valeurs proposées soient des tableaux à deux éléments. À la fin, chaque signal scalaire recevra sa valeur scalaire venant de son vis-à-vis.

11.4.2 Interprétation pour la synthèse

De deux choses l'une : ou bien le signal est affecté dans toutes les branches il n'y a pas d'utilisation des attributs EVENT et STABLE ou de fonction comme rising_edge, alors toute l'instruction peut être réécrite sous une forme d'équation logique et finir en logique combinatoire. Ou bien il y a utilisation d'attributs de stabilité ou l'une des branches contient « **unaffected** » ou le signal lui-même, et alors il est nécessaire de synthétiser un registre pour retenir la valeur de S entre deux cycles du processus.
Exemple :

```
S <= A when B='1' else C ;
```

est équivalent à : `S <= (A and B)  or  (C and not B);`
Ce sera de la logique combinatoire.

```
Par contre:
S<= A when B='1' else unaffected; --  ou "else S;"
```

n'a pas d'équivalence logique combinatoire: S sera un registre chargé par A quand B vaut 1.
Enfin : `S <= A when CK'EVENT and CK='1' else unaffected;`
Donnera une bascule marchant sur front.

11.5 L'affectation à sélection

```
with sélecteur select s<= [options] waveform_element1 when valeur1 ,
                                   waveform_element2 when valeur2 ,
                                   waveform_element3 when valeur3 ,
                                 … ;
```

où waveform_element est d'une des deux formes :

- `expression`
- `expression after délai`

Les options sont vues §11.6 ci-dessous.
Le sélecteur doit être d'un type discret (entier ou énuméré) ou d'un vecteur[17] de type discret

[17] Vecteur = tableau à une seule dimension.

(par exemple BIT_VECTOR ou STRING) et les valeurs sont du même type. Toutes les valeurs de ce type doivent être présentes une fois et une seule dans la sélection (ce sont donc des constantes !), au besoin en utilisant une jonction |, une étendue (`valeuri to valeurj`) ou le mot-clé **`others`**. On comparera utilement cette instruction avec le **case** du monde comportemental, instruction utilisée pour le processus équivalent (voir §9.1 page 67).

11.5.1 Transformation pour la simulation

```
process
begin
case  sélecteur is
   when valeur1 => S <= waveform_element1;
   when valeur2 => S <= waveform_element2;
   when valeur3 => S <= waveform_element3;
...
end case ;
wait on (signaux apparaissant dans les expressions et le sélecteur)
end process ;
```

11.5.2 Interprétation pour la synthèse

Comme pour l'instruction précédente, de deux choses l'une : ou bien le signal est affecté dans toutes les branches sans utilisation d'attributs EVENT et STABLE ou de fonction comme rising_edge, et alors toute l'instruction peut être réécrite sous une forme d'équation logique et finir en logique combinatoire. Ou bien une des branches contient un attribut EVENT ou STABLE et il y aura des bascules, ou « **unaffected** » ou le signal lui-même, et alors il est nécessaire de synthétiser un registre pour retenir la valeur de S entre deux cycles. Voir l'exemple précédent et le transposer.

11.6 Les options

11.6.1 Les délais

Le mode de gestion des délais de chaque affectation peut être contrôlé, voir §7.4 page 56 :

- Le mode par défaut (on ne met rien mais on peut mettre le mot-clé **`inertial`**) est le mode inertiel.
  ```
  S<= expression after temps;
  S<= inertial expression after temps;
  ```
- Dans le mode inertiel on peut séparer le temps de délai de propagation et le temps de réjection des *glitchs*.
  ```
  S <=  reject temps1 expression after temps2 ;
  S <=  reject temps1 inertial expression after temps2 ;
  ```
- On peut inhiber la réjection en mettant le mot-clé **`transport`** :
  ```
  S <=  transport expression after temps2 ;
  ```

11.6.2 guarded expression

Dans le contexte d'un *bloc gardé* (11.8.1 page 92), chacune des trois formes peut être préfixée du mot-clé **guarded** :

Par exemple : `S <= guarded expression ;`

est équivalent à `S <= expression when GUARD else unaffected ;`

… ou à `S <= expression when GUARD else null;`

... si S est de genre **bus** ou **register**, voir §11.8.2 page 93.

Les deux autres formes (conditionnelle et sélectée) sont aussi exécutées si et seulement si GUARD passe de FALSE à TRUE. Il n'y a pas d'instruction concurrente équivalente, par contre dans la transformation en *processus,* cela revient à emballer la transformation de l'affectation par un test sur GUARD, lequel sera ajouté dans la liste de sensibilité du *processus* équivalent:

```
if GUARD
then S<= expression when…  ou with selecteur select S<= …
     (ici la transformation de l'affectation, voir au dessus)
else S <= unaffected ; -- ou null (si S est bus ou register).
end if ;
```

11.7 postponed

C'est un mot-clé qu'on peut trouver devant toutes les instructions concurrentes pouvant se réduire –par équivalence- à un processus. De ce fait, la seule chose qui soit définie pour la simulation est le «**postponed process**». Par contre, en synthèse, certains outils plus ou moins académiques se servent de ces instructions (l'**assert** le plus souvent) pour faire de la spécification par contrainte.

Ce mot-clé indique que, s'il y a un événement sur l'un des signaux de la liste de sensibilité, l'exécution de l'instruction est différée à la fin du jeu de cycles du temps courant. L'intérêt est de pouvoir exécuter du code après la stabilisation du système et donc s'épargner de coûteux tests de stabilité n'ayant pas de signification physique. Évidemment, si d'aventure l'exécution de l'instruction crée elle-même des événements au même temps de simulation, on est dans un cas d'erreur, puisqu'on détruit la condition d'exécution de l'instruction. Pour faire court, une instruction **postponed** peut être passive (**assert**) mais si une instruction **posponed** affecte un signal, elle doit le faire après un délai non nul (clause **after**); le but du jeu est qu'elle ne s'exécute qu'une fois par temps de simulation, à la fin des cycles d'un temps de simulation donné, avec d'autres instructions **postponed**, le cas échéant. Exemple d'instruction passive, utilisée sur un additionneur combinatoire:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
entity ADDER is
port (A,B: in STD_LOGIC_VECTOR (15 downto 0);
             S: inout STD_LOGIC_VECTOR (16 downto 0));
begin
  postponed assert UNSIGNED(S) = UNSIGNED(A)+UNSIGNED(B)
            report "l'additionneur n'additionne pas"
            severity ERROR;
end entity ADDER;

architecture STRUCT of ADDER is
begin
  -- ici long code implémentant l'additionneur au niveau portes
  -- c'est-à-dire plusieurs dizaines de portes interconnectées
  -- donnant le résultat après glitchs et aléas.
end architecture STRUCT;
```

Figure 17 Exemple d'utilisation d'un «postponed assert».

Dans cet exemple, l'addition se calcule à travers les portes et leurs délais. Avant la stabilisation la sortie présente une quantité de «cheveux», états transitoires dus aux propagations et aux états des entrées (propagation des retenues). Ce qui compte, c'est qu'au bout du cycle de stabilisation, la sortie soit égale à la somme des entrées. Ceci est vérifié par l'instruction **assert** qui ne se déclenche qu'après la stabilisation, et vérifie la qualité du résultat en appelant une fonction de bibliothèque.

Sans le mot-clé **postponed**, l'instruction serait activée une grande quantité de fois avant la stabilisation, et pour s'en prémunir il faudrait ajouter à la condition de **l'assert** un test sur la stabilité de la sortie (par l'attribut 'STABLE par exemple). C'est cette solution qu'on trouvera sur les modèles écrits avant les simulateurs implémentant VHDL'93.

11.8 Choses gardées

11.8.1 Blocs gardés

Bien que d'usage assez restreint en pratique, surtout depuis la standardisation du paquetage STD_LOGIC_1164, le bloc gardé permet d'écrire facilement de la logique synchrone en « factorisant » des événements, par exemple le front montant de l'horloge. Un bloc gardé définit un signal particulier de type BOOLEAN, appelé GUARD, qui sera affecté avec la condition de garde qui doit donc être booléenne. Ce signal existera effectivement et sera utilisable dans toutes les instructions du bloc comme un signal booléen normal. Il sera visible dans les simulateurs. Néanmoins il a une magie spécifique *dans le cas des affectations de signaux* (§11.2 page 87) : toute utilisation du mot-clé **`guarded`** (§11.6.2 page 91) signifie que l'affectation est conditionnée par le passage de ce signal de FALSE à TRUE, et que dans le passage inverse le signal sera déconnecté (on lui affecte **null**) s'il est du genre **bus** ou **register**, voir ci-dessous les définition de ces signaux.

Exemple :

```
nom_du_bloc : block (clk'event and clk='1')
  -- ici déclarations possibles
begin
 S <= guarded expression ;
end block;
```

Ceci est équivalent à :

```
nom_du_bloc : block
  signal GUARD : boolean ;
begin
GUARD <= (clk'event and clk='1') ;
S<= expression when GUARD else unaffected ;
        -- ou else null ; si S est de genre bus ou register
end block;
```

Dans le bloc, certaines instructions *peuvent parfaitement ne pas* être gardées ; les signaux de genre **bus** ou **register**, par contre, *doivent* l'être : voir ci-dessous. Les premières « voient » alors les déclarations locales du bloc, y compris le signal GUARD déclaré implicitement, mais ne profitent pas du mécanisme de garde : la condition automatique.

11.8.2 Signaux gardés (bus, register)

Il est possible de décorer, lors de la déclaration d'un signal, celui-ci par la mention **bus** ou **register**.

```
signal S1 : std_logic bus ;
signal S2 : std_logic register ;
```

Un tel signal qui doit être d'un sous-type résolu -§7.5 page 57- ne pourra ensuite apparaître que dans des affectations gardées -§11.6.2 page 91-. Leur comportement spécifique est le suivant :

Lorsque le signal est déconnecté (affectation de **null** sur toutes ses sources), le signal de genre **bus** prend la valeur telle que rendue par la fonction de résolution avec zéro contribution (donc probablement une valeur « haute impédance »). Le signal de genre **register** garde la valeur telle que calculée *avant* la dernière déconnexion. Il est même possible, avec une clause de déconnexion qui est un item déclaratif (à mettre avant le **begin**), de spécifier le temps au bout duquel le signal prendra cette nouvelle valeur de déconnexion. On peut ainsi écrire dans une zone déclarative où l'on voit les signaux concernés:

```
disconnect S : STD_LOGIC after 10 ns ; -- concerne S
```

ou

```
disconnect all : STD_LOGIC after 10 ns ; -- tous les signaux du type
```

ou

```
disconnect others : STD_LOGIC after 10 ns ; -- ceux pas déjà mentionnés.
```

11.8.2.1 Autres instructions concurrentes

11.8.2.1.1 Assert

L'assertion a une seule partie non optionnelle: la condition.

```
[lbl:] [postponed] assert condition [report "message"] [severity SEVERITE];
```

L'assertion concurrente vérifie «en permanence» la véracité d'une condition. Si cette condition devient fausse par le changement de valeur d'un signal de la condition, l'assertion est activée et le message est émis sur la console. De plus, l'action prédéfinie (sur le simulateur) pour ne niveau de sévérité concerné est activée. Pour le fonctionnement de l'assertion en mode séquentiel, voir §12.3.3 page 99.

Attention: une erreur courante consiste à oublier que l'assertion ne se déclenche qu'en cas de changement de valeur d'un des signaux de la condition. Si cette condition fait intervenir autre chose que des signaux, par exemple une variable partagée (§4.6 page 25), eh bien leur changement ne déclenchera pas l'évaluation de la condition sauf au temps zéro (voir paragraphe ci-dessous).
Son exécution est décrite, pour ce qui est de la simulation, comme l'exécution d'un processus équivalent -voir §9.1 page 67- contenant la même assertion mais sous sa forme séquentielle, suivie d'un **`wait on`** sur tous les signaux apparaissant dans la condition. S'il n'y en a pas, le **`wait`** est «forever» (§12.3.1 page 98). Cela n'est pas forcément pathologique, on peut vouloir vérifier par un **assert** des conditions statiques portant sur les constantes et les génériques, de façon à arrêter la simulation au temps zéro s'il y a incohérence. Exemple:

```
entity E is
 generic (N:INTEGER);
 port(S:BIT_VECTOR(N-1 downto 0));
begin
  assert N<1024 report "N est décidément trop grand" severity FAILURE;
end entity E;
```

11.8.2.1.2 Appel concurrent de procédure

L'appel concurrent de procédure est, comme toutes les instructions concurrentes, défini pour la simulation par une équivalence-processus: on la trouvera dans §9.1 page 67.

On se souviendra que la procédure, en VHDL, peut avoir des arguments de différentes classes: variable[18], signal, constante ou fichier. Le processus équivalent sera sensible sur la liste des arguments de la procédure qui sont des signaux passés à des arguments en mode **in** ou **inout**.

L'utilisation de l'appel concurrent de procédure est adaptée à de nombreux cas de figures, et, il faut bien le dire, trop souvent méconnu. Cela peut être une façon rapide de faire des composants «comportementaux»:

```
procedure PROC (signal A,B: STD_LOGIC; signal S:out STD_LOGIC) is begin…
…
LBL1: PROC(X, Y, Z);
LBL2: PROC (M, N, P);
```

On peut ainsi écrire par exemple une RAM en style comportemental, et l'instancier plusieurs fois. Pour le faire sans passer par une procédure concurrente, il faudrait ou bien jouer du copié-collé et dupliquer les processus, ou bien faire une entité spécifique, son architecture avec un processus, puis son composant et sa configuration; tout cela pour du code qui n'a sans doute pas vocation à être synthétisé.

[18] Dans le cas d'appel concurrent de procédure, la classe variable ne peut concerner que des variables partagées (voir §4.6 page 25), à cause des règles de visibilité.

On peut aussi optimiser une simulation en réduisant les listes de sensibilité à leur minimum fonctionnel. Considérons l'affectation de signal:

```
AFFECT:
 registre<=grand_bus when année_bissextile'event and année_bissextile = '1'
                     else unaffected;
```

Il est clair que le registre ne sera affecté que le 1er janvier des années bissextiles. Néanmoins, l'instruction est sensible à tous ses signaux de la partie droite, et va donc se réveiller chaque fois qu'il y a un événement sur le bus, c'est-à-dire potentiellement dix fois par microseconde, juste pour évaluer une condition quasiment toujours fausse!

Le processus équivalent (cf §9.1 page 67) est donc:

```
AFFECT: process
begin
if année_bissextile'event and année_bissextile = '1'
  then registre<= grand_bus;
end if;
wait on grand_bus, année_bissextile;
end process AFFECT;
```

Cela conduit évidemment à un gaspillage de ressources pour le simulateur. On peut réécrire cette instruction ainsi:

```
AFFECT: affecter (registre, grand_bus, année_bissextile);
```

La procédure sera écrite plus haut, dans la partie déclarative correspondante:

```
procedure affecter (
   signal reg: out BIT_VECTOR(63 downto 0);
   signal gbus: in BIT_VECTOR(63 downto 0);
   signal année_b: in BIT) is
begin
  loop
    reg<=gbus;
    wait until année_b = '1';
  end loop;
end procedure affecter;
```

Maintenant, la procédure entre dans une boucle infinie qui se réveille seulement quand il y a un événement sur le signal intéressant. Évidement on ne va pas réécrire ainsi toutes les affectations de signaux, mais on voit ici que parfois, cela peut considérablement optimiser une simulation. On aurait pu écrire la même chose sous la forme d'un processus: l'avantage de la forme procédurale est qu'elle est réutilisable.

11.8.2.1.3 Le processus (process)

Le processus est une instruction concurrente. Mais il ouvre une région de code admettant des instructions à sémantique séquentielle, et le chapitre 12 est consacré à ces instructions (page 97).

```
[label:] process
begin

end process [label];
```

Le mot clé **postponed** indique que le processus correspondant, ou le processus équivalent de l'instruction marquée, ne s'exécutera qu'à la fin du cycle de simulation et ne contribuera plus au cycle courant.

```
[label:] postponed process
begin

end process [label];
```

Le processus, à la simulation, doit impérativement s'arrêter sur un **wait** (voir le cycle de simulation chapitre 8 page 61). C'est donc une erreur si un processus n'en contient pas (ni d'appel de procédure qui en contiendrait) et le compilateur le signale. Hélas il est possible de passer la compilation, par exemple en écrivant un **wait** et en ne passant jamais dessus à cause de l'algorithme. Ou d'appeler une procédure dont le corps est invisible et dont le compilateur « ne sait pas » si elle contient un **wait**. Dans ce cas, c'est le simulateur qui va râler, après avoir fait tourner le processus un nombre de fois déraisonnable (à régler dans les menus du simulateur)..

Il y a deux façons de prescrire un **wait**: la première est l'explicite: on écrit l'instruction où l'on veut et autant de fois que l'on veut (§12.3.1 page 98). La seconde est l'implicite: on met la liste des signaux intéressés entre parenthèses à côté du mot-clé **process**:

```
[label:] process (S1,S2)
begin
…
end process [label];
```

Le compilateur va alors utiliser l'équivalence suivante (voir §9.1 page 67), et surtout interdire qu'une instruction **wait** explicite apparaisse dans le processus.

```
[label:] process
begin
…
wait on S1,S2;
end process [label];
```

L'écriture avec **wait** implicite est plus contraignante mais préférée des outils de synthèse.

12 Comportemental (*behavioral*): dans le processus, *le procedural,* etc.

Le processus est en lui-même une instruction concurrente: à ce titre sa syntaxe «extérieure» est vue §11.8.2.1.3 page 95 . Il contient des instructions séquentielles. *[AMS] De même, le procédural vu §13.3 page107 contient des instructions séquentielles.* Le processus se comporte comme une boucle infinie, et les variables et fichiers qu'il déclare sont rémanents : il retrouve les mêmes objets dans l'état où il les avait laissés, à chaque tour. *[AMS]À l'inverse, le procedural se comporte plutôt comme une fonction –il est défini par une fonction équivalente- et ses variables locales sont détruites, recrées et réinitialisées à chaque invocation par le noyau analogique.*

12.1 Instructions

12.2 Retour sur la variable

La variable en VHDL a exactement la même sémantique que la variable en C, Pascal ou Ada : c'est un porteur de valeur qui contient une seule valeur, sans histoire ni plans sur le futur (au contraire du signal, donc). Quand elle est affectée, elle change immédiatement (il n'y aurait pas de place pour mettre la valeur « en conserve ». C'est donc la variable qui fait que la séquence des instructions exécutées entre deux **wait** dans un processus est observable.

Preuve : Un processus qui ne ferait qu'affecter des signaux ne ferait qu'écrire des valeurs calculées dans des tuyaux, valeurs qui ne lui deviendraient visibles au mieux qu'après le prochain **wait**. Donc, entre deux **wait**, s'il n'y a pas d'affectation de variable l'ordre des instructions est indifférent. Les deux processus ci-dessous ont un fonctionnement identique:

```
process
begin
 A <= valeur calculée;
 B <= valeur calculée;
 wait on …
end process ;
```

```
process
begin
 B <= valeur calculée;
 A <= valeur calculée;
 wait on …
end process ;
```

De même, plus surprenant, ces deux-ci:

```
process
begin
 A <= valeur calculée;
 B <= valeur calculée;
 if (A=B) then
   C<= valeur calculee;
 end if;
 wait on …
end process ;
```

```
process
begin
 if (A=B) then
   C<= valeur calculee;
 end if;
 B <= valeur calculée;
 A <= valeur calculée;
 wait on …
end process ;
```

Maintenant, si A,B et C sont transformées en variables, la situation change complètement: l'ordre des instructions devient important et le test des derniers processus change de nature;

Nous avons vu qu'il y a deux sortes de variables: les variables locales aux processus, et les variables partagées d'usage plus restreint. Dans le cas très général des variables locales aux processus (et aux sous-programmes), les simples règles de visibilité font qu'elles ne sont accessibles qu'en contexte séquentiel.

12.3 Instructions simples

12.3.1 Wait

L'exécution du **wait** a pour effet d'arrêter le processus et de le rendre sensible à une certaine condition de réveil. Voir le cycle de simulation chapitre 8 page 61.
Il y a quatre formes de wait :

- wait forever: le processus s'arrête définitivement:
  `wait;`
- On attend un événement sur une liste de signaux :
  `wait on S1,S2,.. ;`
- On attend un temps donné :
  `wait for 5 ns ;`
- On attend qu'une condition se réalise :
  `wait until condition ;`
  dans ce cas, l'instruction commence par attendre un événement sur la liste des signaux contenus dans la condition, puis teste la condition. Voir §9.3 page 71.

Ces trois derniers cas peuvent être mélangés :

- `wait on S1,S2 for 5ns ;` on attend un événement mais pas plus que 5 ns.
- `wait on S1,S2 until condition ;` on attend un événement sur S1 ou S2, puis on teste. Attention! Les éventuels signaux de la condition ne sont pas pris en compte.
- `wait on S1,S2 until condition for 5 ns ;` pareil mais pas plus de 5 ns.
- `wait until condition for 5 ns ;` comme **wait until** mais pas plus de 5 ns.

*[AMS] Le **wait** n'est pas autorisé dans une instruction **procedural**.*

12.3.2 Affectation

On peut affecter les signaux et les variables depuis un processus : les deux classes d'objets y sont visibles par la simple conséquence de l'imbrication des begin/end : le processus est déclaré dans une architecture, où l'on déclare aussi des signaux. Le processus peut déclarer des variables non partagées : il peut donc travailler sur les deux.
Attention: il est possible en VHDL d'affecter des agrégats d'objets, c'est-à-dire qu'on met à gauche de l'affectation un agrégat de noms d'objets à affecter, et à droite un type compatible avec l'agrégat. À la fin, chaque objet scalaire recevra sa valeur scalaire venant de son vis-à-vis.

Ainsi ceci est légal: `(C1, C2, C3) := Fonction_rendant_une_STRING;`

…pour autant que la fonction rende trois caractères. C1 recevra le premier caractère, etc.

12.3.2.1 Affectation de variable

```
variable  V ;
…
begin
…
V:= valeur;
```

La variable prend «immédiatement» la valeur soumise. Voir §4.4 page 24 au sujet de la variable en tant qu'objet.

12.3.2.2 [AMS]Affectation de quantités

Dans un «procedural» les quantités visibles peuvent être affectées comme des variables, par le signe ":=". Il s'agit d'une affectation qui sera exécutée sous le contrôle du noyau analogique, et dont la sémantique n'est pas celle d'une affectation de variable mais plutôt celle d'un but à atteindre pour le noyau. Qui, finalement, va activer le procedural autant de fois que nécessaire et avec les valeurs nécessaires pour que l'instruction simultanée équivalente soit vraie (voir §13.3 page 107).

12.3.2.3 Affectation de signal

```
S <= [options] waveform ;
```

Où `waveform` est une succession de «`valeur [`**`after`** `délai]`» séparés par des virgules, et les options sont : **guarded**, **transport**, **reject**. Voir §11.6.2 page 91 pour les expressions gardées, et §7.4 page 56 pour les deux derniers mot-clés.

Voir chapitre 7 page 53 au sujet du signal en tant qu'objet. Cette instruction est utilisée par équivalence des instructions concurrentes du monde *dataflow*. Voir §11.2 page 87 pour ces instructions, et §9.1 page 67 pour les équivalences.
[AMS] L'affectation de signal n'est pas autorisée dans une instruction ***procedural****.*

12.3.3 Assert et report

L'instruction assert a pour seul effet d'écrire un message sur la console utilisateur, et d'interférer avec le contrôle de la simulation d'une façon qui peut être prédéfinie :

```
assert condition report "le message" severity ERROR ;
assert condition ;
assert condition report "le message";
assert condition severity ERROR ;
```

L'assert n'affecte aucun signal: son exécution ne participe donc pas à l'exécution du modèle. Elle n'a *a priori* aucun sens en synthèse, quoique certains outils académiques se servent des assertions pour faire de la spécification par contrainte.
Le message apparaîtra si la condition est *fausse* au moment de l'évaluation. La sévérité a plusieurs valeurs possibles venant d'un type énuméré du paquetage STANDARD (§16.1.1 page 115). Ces valeurs (NOTE, WARNING, ERROR, FAILURE) n'ont pas d'effet documenté, on peut décider dans la simulation que à partir de tel niveau de sévérité on arrête la simulation, ou on la met en pause, qu'on met un point d'arrêt, ou qu'on fait un message rouge clignotant avec fumigènes si la possibilité existe: ce n'est pas normalisé.

Le message doit être du type STRING, ce qui peut causer quelques soucis si l'on veut tracer des valeurs d'autres types. On se souviendra de l'attribut IMAGE qui est bien commode (chapitre 15 page 113): par exemple si la valeur intéressante est du type entier, le message peut être:
`"la valeur de N est:" & INTEGER'IMAGE(N);.`
Message et sévérité sont optionnels: le simulateur fournira un message par défaut: `"assertion violation.";` et une sévérité par défaut: ERROR.

L'instruction **assert** est surtout utile dans sa variante concurrente (§11.8.2.1.1 page 93), dont l'équivalence est donnée § 9.1 page 67. Dans ce cas il est courant que sa condition dépende de signaux, sur lesquels le processus concurrent sera sensible. Si la condition ne dépend pas de signaux, l'instruction **assert** concurrente ne sera exécutée qu'une fois à l'initialisation, ce qui peut avoir un intérêt pour tester des conditions de validités, sur des arguments génériques par exemple.

Cette instruction et sa variante **postponed** permettent de surveiller des conditions qui *doivent* être vraies à tout moment (la variante **postponed** limite la surveillance aux états stables).

L'instruction **report** est identique à une instruction **assert** dont la condition serait toujours fausse. Elle permet de forcer une trace. Dans ce cas la sévérité par défaut est NOTE. Noter qu'elle n'a pas de variante concurrente, dont on ne saurait que faire.

```
report "message":
```
Ou
```
report "message" severity FAILURE;
```

*[AMS] Dans une instruction **procedural**, les **assert** et **report** sont autorisés, mais le nombre de fois qu'ils sont exécutés pendant un cycle d'évaluation n'est pas dit dans la norme.*

12.3.4 Appel de procédure

L'appel de procédure est une instruction séquentielle. La question est vue dans le paragraphe consacré aux procédures, avec la question de la surcharge : §12.7 ci-dessous page 103.

12.3.4.1 [AMS] break

*[AMS] Le **break** n'est pas autorisé dans une fonction ni dans une instruction procedural. L'instruction **break** permet au monde «digital» de forcer le monde «analogique» à recalculer ses valeurs dites initiales. Cela est nécessaire lors que survient une discontinuité engendrée par le changement de valeur d'un signal et qu'il faut faire une analyse du circuit pour trouver le nouveau point d'équilibre.*
Le fonctionnement de cette instruction est détaillé §8.3.3 page 65.

12.4 Instructions composites

12.4.1 If

```
[label:] if condition then sequence d'instruction
           {elsif condition then séquence d'instructions)
           [else séquence d'instructions]
         end if;
```

Cette instruction est triviale: si la première condition est vraie, la première séquence d'instructions est exécutée, sinon on teste la deuxième condition, etc. Une branche **else** peut attraper le fait qu'aucune condition n'a été vraie.
Toutes les branches sont optionnelles sauf la première.
Cette instruction est celle qui est exécutée par équivalence de l'instruction conditionnelle du monde *dataflow*. Voir l'instruction §11.4 page 88, son équivalence §9.1 page 67.

12.4.2 Case

```
[label:] case selecteur is
   when valeur1 => sequence d'instructions
   when valeur2  to valeur 3 => sequence d'instructions
   when valeur4 | valeur5 => sequence d'instructions
   when others => sequence d'instructions
end case;
```

Cette instruction compare le sélecteur qui doit être d'un type discret ou vecteur de discrets, (voir §5.2.2 page 32) avec les valeurs, qui doivent être des constantes du même type. La séquence d'instructions qui correspond est exécutée.
Il y a une et une seule branche concernée: le compilateur vérifie que toutes les valeurs du type sont prises en charge une fois et une seule –c'est pour cela que le type doit être discret et que les valeurs doivent être des constantes-.
On peut spécifier une étendue de valeurs avec **to** et **downto**, et un choix avec la barre verticale. On peut mettre en dernière position un attrape-tout **others**, qui devient quasi-indispensable si l'on veut un sélecteur du type entier (l'autre branche de l'alternative est d'avoir des choix du style `INTEGER'LOW to 0` et `10 to INTEGER'HIGH)`.

Cette instruction est celle qui est exécutée par équivalence de l'instruction sélectée du monde *dataflow*. Voir l'instruction page 89, l'équivalence page 67.

12.4.3 Loop

La boucle contient du code qui va être exécuté répétitivement un nombre de fois qui dépend de sa condition d'arrêt. Elle a trois formes:

12.4.3.1 La boucle infinie

```
[label:] loop … end loop;
```

La boucle infinie, comme son nom l'indique, ne s'arrête jamais (sauf exécution de l'instruction **exit**, voir §12.5.1 ci-dessous). Contrairement aux langages informatiques, il n'est ni rare ni erroné de trouver des boucles infinies en VHDL, après tout le matériel est «permanent».

12.4.3.2 La boucle for

```
[label:] for I in 1 to 10 loop … end loop;
[label:] for I in 10 downto 0 loop … end loop;
[label:] for I in A'RANGE loop … end loop;
```

La boucle est contrôlée par un indice qui lui-même varie selon un schéma donné en tête: il parcourt une étendue (RANGE) qui est donnée explicitement: 1 **to** 10; ou par référence à un type ou un objet: A'RANGE. L'indice ne varie que de 1 en 1.

Attention: l'indice de boucle n'a pas à être déclaré. En fait, si un I était déclaré dans nos exemples, ce serait un autre, accessible par son nom complet dans la boucle (arch.I par exemple). Donc cet indice de boucle est inaccessible à l'extérieur de la boucle. Par ailleurs, il est vu comme une constante pendant l'exécution de la séquence d'instructions: on ne peut pas le modifier par affectation, ni le passer en argument à un sous-programme qui le modifierait.

12.4.3.3 La boucle while

```
[label:] while condition loop … end loop;
```

La boucle est contrôlée par l'évaluation d'une condition. Tant que la condition est vraie, la séquence d'instructions est exécutée. Si la condition est fausse avant d'entrer dans la boucle, la séquence n'est pas exécutée du tout.

12.5 Instructions contextuelles

Certaines instructions n'ont de sens que dans certains contextes: dans une boucle, ou dans un sous-programme. On pourrait y ranger les instructions **wait** et **break** qui ne peuvent pas exister dans une fonction (une fonction s'exécute à temps de simulation strictement nul, même pas un delta) et donc non plus dans un **procedural** puisque sa définition fait appel à une fonction équivalence.

12.5.1 Exit

exit; fait sortir de la boucle la plus interne.
exit `label`; fait sortir de la boucle dont le nom est label. Donc même si c'est une boucle externe.

L'instruction peut être conditionnelle:

```
exit when condition
exit label when condition;
```

On peut ainsi faire l'équivalent de la boucle ***do***…***while*** *condition* d'autres langages: il suffit de mettre le **exit when** `condition` à la fin de la boucle.

12.5.2 Next

next; fait recommencer à son début la boucle la plus interne.
next `label`; fait recommencer à son début la boucle dont le nom est label. Donc même si c'est une boucle externe.

L'instruction peut être conditionnelle:

```
next when condition
next label when condition;
```

12.5.3 Return

```
return valeur;
```

Permet de sortir d'une fonction et de rendre la valeur. C'est la seule façon légale de sortir d'une fonction. Autrement dit, si une fonction arrive sur son **end**, c'est une erreur à l'exécution.
return; (sans valeur)
Permet de sortir d'une procédure. On peut aussi en sortir en passant par le **end**.

Le **return** ne peut donc pas exister dans un processus.

*[AMS] Le **return** ne peut donc pas exister non plus dans une instruction **procedural**.*

12.6 Sous-programmes

Le sous-programme factorise du code qui peut être appelé un grand nombre de fois. Il peut aussi servir à augmenter la lisibilité du code en isolant des algorithmes spécifiques. Il permet enfin de décrire des algorithmes récursifs.

12.7 Procédure

Une procédure est une portion de code comportemental. On l'appelle en contexte

- Concurrent, voir §11.8.2.1.2 page 94, auquel cas pour la simulation l'appel est transformé en processus équivalent : voir §9.1 page 67 et la question se résume à l'appel en contexte séquentiel.
- Séquentiel : dans ce cas la sémantique de l'appel de procédure est la sémantique classique en informatique : les arguments sont passés selon leur mode, le code est déroulé et la procédure se termine après avoir éventuellement changé ses arguments de mode **out** ou **inout**, ou le contexte si elle tape dans des objets qui lui sont extérieurs.

Les procédures en VHDL peuvent être récursives.

12.7.1 Déclaration

Portion de la BNF (voir définition page 9) :

```
 ::= procedure designator [ ( formal_parameter_list ) ]
```

```
Exemple:
procedure P (
             arg1: integer;
             arg2: out BIT;
             signal arg3: STD_LOGIC
             [AMS] ; quantity Q: real
             )
```

Quand il s'agit juste de déclarer l'existence de la procédure, on termine par un point virgule. La procédure devient visible pour ses clients. Il faudra évidemment en écrire la définition plus loin, soit dans la même unité si l'on est dans une architecture ou un corps de paquetage, soit dans l'unité « corps » si l'on est dans une entité ou une déclaration de paquetage.

12.7.2 Définition

Quand l'on veut définir la procédure (en écrire le code) on termine la déclaration par **is** et ensuite on peut déclarer des objets locaux, mettre le mot-clé **begin** et écrire l'algorithme jusqu'au **end**.

```
procedure P (
             arg1: integer;
             arg2: out BIT;
             signal arg3: STD_LOGIC
             [AMS] ; quantity Q: real
             ) is
begin
 -- ici le code de la procédure
end procedure P;
```

Si la définition fait référence à une déclaration antérieure, par exemple si on écrit le code de la procédure dans un corps de paquetage alors qu'elle est déclarée dans la partie visible, les deux déclarations doivent être des copié-collé l'une de l'autre.

12.7.3 Appel et surcharge

La procédure s'appelle par son nom suivi de la liste de ses arguments. Toutes les subtilités de l'association sont au §10.3 page 75.
La surcharge est une fonctionnalité partagée avec les fonctions et les littéraux de types énumérés. On verra §6.2.5 page 49 ce qu'il en est de la surcharge des objets rendant une valeur (fonctions, littéraux énumérés).
Pour ce qui est des procédures, la situation est plus simple : elles ne peuvent être surchargées qu'avec d'autres procédures. Plusieurs procédures peuvent avoir le même nom (voir par exemple les procédures READ du paquetage TEXTIO §16.1.2 page 117). Elles sont distinguées alors par le nombre et le type de leurs arguments. Ceci se complique, comme pour les fonctions, par le fait que certains arguments peuvent avoir des valeurs par défaut et que donc une procédure peut participer plusieurs fois à la résolution de la surcharge. Pour ne rien simplifier, ces arguments peuvent venir eux-mêmes d'expressions dont il faut résoudre la surcharge. À la fin,

- Soit il y a une et une seule solution permettant de déterminer quelle est la procédure qui convient, et c'est celle qui est choisie avec ses arguments éventuellement surchargés dont la surcharge est résolue dans la foulée avec une résolution qui doit aussi être unique
- Soit il y a zéro ou plus d'une solution, et le compilateur refuse de choisir : c'est une erreur.

Attention : les valeurs utilisées ne participent pas à la résolution de la surcharge, seulement leur type ou leur genre de type le cas échéant. TEXTIO.WRITE("XYZ") est ambigu car il y a deux WRITE acceptant des types de genre chaîne de caractère (STRING et BIT_VECTOR). Le fait que 'X','Y' et 'Z' ne soient pas des valeurs du type BIT n'est pas considéré, if faut qualifier le littéral : STRING'("XYZ"). Voir la qualification §5.1 page 31.

12.8 Fonction : voir §6.2.1

La fonction, qui rend une valeur et à ce titre apparaît dans tous les contextes et pas seulement le comportemental, est détaillée au §6.2.1 page 47.

13 [AMS] Simultané

Les instructions simultanées ne sont pas séquentielles et peuvent apparaître partout où l'on trouve des instructions concurrentes. Leur ordre respectif n'a pas d'importance, non plus que leur ordre vis-à-vis des instructions concurrentes. Elles ne sont pas des instructions impératives à exécuter de façon définie, mais des « vérités » que l'on pose.
*Leur effet dans la simulation n'est pas défini par la norme : tout ce qu'on peut raisonnablement supposer c'est que le solveur va faire de son mieux pour les « rendre vraies » en permanence, en ajustant les valeurs des quantités **out**, **through** et libres. On peut –et on doit- l'aider à converger et à gérer les discontinuités par des indications statiques et dynamiques : valeurs initiales, déclarations de tolérances, clause **limit**, instruction **break**.*
Pour ce qui est de la synthèse, les outils sont encore à l'état de projets de recherche.

13.1 Tolérance

L'évaluation des instructions simultanées est sujette à la précision d'un algorithme numérique, laquelle dépend de différents paramètres dont le temps de calcul souhaité. Chaque quantité et chaque instruction simultanée appartiennent à un groupe «de tolérance» et le noyau de simulation peut savoir, pour chaque, quelle stratégie adopter pour résoudre chaque équation ou instruction du monde analogique.
La tolérance est définie en VHDL comme une simple chaîne de caractères, dont on suppose que le simulateur saura quoi faire. Elle n'a pas d'autre effet dans le langage sinon qu'une fois définie, on peut la retrouver par un attribut TOLERANCE.

Pour les quantités, la tolérance est héritée de leur déclaration: le sous-type auquel appartient la quantité peut être contraint par une tolérance (§5.6 page 40). Par défaut, le type REAL a une tolérance anonyme (chaîne vide "").

```
subtype temperature is REAL tolerance "defaut_temperature";
…
quantity T: temperature;
```

13.2 Instructions simultanées

13.2.1 Instruction simultanée simple

BNF (voir définition page 9) :

```
simple_simultaneous_statement ::=
 [ label : ] simple_expression[19] == simple_expression [ tolerance_aspect ] ;
```

[19] *L'utilisation de la règle simple_expression dans la BNF interdit que les premiers opérateurs de l'arbre de cette expression (voir 6.1 page 45) soient des opérateurs logiques ou de shift, quand bien même ils auraient été surchargés pour rendre des réels.*

Ce sont des expressions séparées par le signe double égal : «== ». Les deux termes de l'équation sont permutables. Le type des deux expressions doit être d'un (sous)type de genre réel ou d'un tableau dont les éléments sont d'un (sous)type de genre réel, auquel cas cela revient à poser une équation par couple d'éléments des tableaux.
Au moins une quantité doit apparaître dans une telle équation, qui peut par ailleurs impliquer tous les porteurs de valeurs de VHDL (signaux, constantes, littéraux).
La tolérance est une chaîne de caractères (optionnelle) derrière le mot-clé ***tolerance****. Comme vu ci-dessus, elle n'a pas de signification dans le langage et sert au noyau de simulation pour régler ses algorithmes au mieux.*

Exemples:
```
U == R * I;
EQ: F == m * x'dot'dot tolerance "mecanique";
```

13.2.2 Instruction simultanée conditionnelle

BNF (voir définition page 9) :
```
simultaneous_if_statement     ::=
   [ if_label : ] if condition use simultaneous_statement_part
                  { elsif condition use simultaneous_statement_part }
                  [ else simultaneous_statement_part ]
                  end use [ if_label ] ;

simultaneous_statement_part   ::= { simultaneous_statement }

simultaneous_statement  ::= simple_simultaneous_statement |
simultaneous_if_statement | simultaneous_case_statement |
simultaneous_procedural_statement | simultaneous_null_statement
```

Il s'agit de choisir un jeu d'équations (ou procédural) ou un autre, en fonction d'une condition.

Attention: comme dans l'exemple ci-après, il est possible sinon fréquent que le changement de la condition implique une discontinuité dans une quantité impliquée, ou sur sa dérivée. Dans ce cas, la simulation fonctionnera correctement seulement si cette discontinuité est indiquée explicitement au simulateur, de façon à lui permettre de faire un point de calcul (ASP, Analog Solution Point) à ce temps là et de réinitialiser ce qui a besoin de l'être avec de nouvelles conditions initiales. Pour ce faire, on utilisera l'instruction ***break*** *§8.3.3 page 65.*

Exemple: un système dont le comportement est résistif tant que la tension n'est pas trop forte, et ensuite se comporte en limiteur de courant.

```
écrêteur:  if not V'ABOVE(Vmax) use
             I == V1 / R ;
           else
             I == Imax;
           end use;
-- Attention: comme on veut une discontinuité dans la dérivée de I,
-- il est indispensable de forcer le noyau à recalculer ses
-- conditions «initiales» lors de cette discontinuité:

break on V'ABOVE(Vmax); -- voir §8.3.3 page 65.
```

13.2.3 Instruction simultanée sélectée

BNF (voir définition page 9) :

```
simultaneous_case_statement   ::=
    [ case_label : ] case expression use
                          simultaneous_alternative
                          { simultaneous_alternative }
                     end case [ case_label ] ;
simultaneous_alternative      ::=
                     when choices => simultaneous_statement_part

simultaneous_statement_part   ::= { simultaneous_statement }

simultaneous_statement  ::= simple_simultaneous_statement |
simultaneous_if_statement | simultaneous_case_statement |
simultaneous_procedural_statement | simultaneous_null_statement
```

*Il s'agit de choisir un jeu d'équations (ou procédural) ou un autre, en fonction de la valeur d'un sélecteur qui doit être d'un type énuméré et dont toutes les valeurs doivent être prises en compte. C'est une instruction de la même famille que le **case** §12.4.2 page 101 ou le **select** §11.5 page 89, avec les mêmes contraintes et les mêmes avantages: toutes les valeurs possibles du sélecteur doivent être traitées une fois et une seule. Le sélecteur doit donc être d'un type discret ou vecteur de discrets, et les valeurs mentionnées doivent être des constantes.*

*Comme dans le cas de la sélection, le fait de changer brusquement de jeu d'équations peut provoquer des discontinuités sur certaines quantités ou leurs dérivées. Dans ce cas également, c'est à la charge du concepteur que de signaler la chose au simulateur avec l'instruction **break** (§8.3.3 page 65).*

13.3 Procedural

BNF (voir définition page 9) :

```
simultaneous_procedural_statement   ::=
       [ procedural_label : ] procedural [ is ]
                              procedural_declarative_part
                              begin
                                 procedural_statement_part
                              end procedural [ procedural_label ] ;
```

*C'est une instruction qui se place dans une zone concurrente, comme toutes les instructions simultanées. Mais elle contient des instructions séquentielles, sauf des affectations de signaux, des **wait** et des **break** (ces instructions renverraient au monde événementiel en dehors des canaux prévus).*

On y déclare des variables qui ne sont pas «permanentes» comme dans un processus: leur valeur est réinitialisée à chaque évaluation de l'instruction.
Les quantités externes y sont vues comme des variables. On peut donc les affecter avec le symbole « := »

*La norme décrit la sémantique de cette instruction par une équivalence à la forme «instruction simultanée simple» (§13.2.1 page 105). D'un côté de l'équation on met l'agrégat des quantités affectées dans le **procedural**, de l'autre l'appel à une fonction dont le code est*

la copie de celui qui est dans l'instruction, et à laquelle on passe toutes les quantités mentionnées dans le code.
Exemple:

```
Lbl: procedural is
begin
 Q1 := …Q3…
 Q2 := …Q4…
end procedural;
```

*Pour l'équivalence, on supposera une fonction contenant le même code, et dont les arguments sont la liste de tous les objets nécessaires au **procedural** pour calculer (toutes les quantités, les terminaux avec leurs quantités). On passe par une copie locale pour pouvoir les affecter dans l'algorithme.*

```
function F_Lbl(QQ1,QQ2,QQ3,QQ4:real) return real_vector is
  variable Q1:real:=QQ1;
  variable Q2:real:=QQ2;
  variable Q3:real:=QQ3;
  variable Q4:real:=QQ4;
begin
  Q1 := …Q3…
  Q2 := …Q4…
  return (Q1,Q2);
end function F_Lbl;
```

Cette fonction rend un real_vector, type qui est défini dans le paquetage STANDARD (§16.3.1 page 129).

*L'instruction équivalente est une simple équation simultanées, où l'on décrète l'égalité de l'agrégat des quantités affectées dans le **procedural**, avec l'appel de la fonction:*

```
Lbl:(Q1,Q2)==F_Lbl(Q1,Q2,Q3,Q4);
```

13.4 Condition de solution

*La vieille règle durement apprise à l'école s'applique: le nombre d'inconnues doit être égal au nombre d'équations. VHDL impose cette règle par unité de compilation[20]. Donc: dans une architecture, la somme des quantités libres, des quantités **through** et des quantités de mode **out** doit être égale au nombre d'équations. C'est une chose que peut vérifier le compilateur, et il fera donc une erreur si ce n'est pas le cas.*

*Les quantités dont il est question ne doivent pas forcément apparaître dans des équations explicites: il ne faut pas oublier que les lois de conservation attachées aux terminaux participent à la résolution, et que le simple fait de mentionner une quantité **through** l'attache au système à résoudre, même si on ne s'en sert pas explicitement.*

[20] Au prix de beaucoup de complications, cette règle aurait pu être assouplie et n'exiger l'égalité inconnues=nombre d'équations qu'au niveau du modèle entier après élaboration. Les concepteurs du langage ont choisi la solution sage de la vérification par unité.

Exemple tiré de [TUT] chapitre 19 page 169

```
entity Vdc is
generic (dc: REAL);
port (terminal p, m: electrical);
end entity Vdc;
```

```
architecture Bad of Vdc is
quantity v across p to m;
begin
v == dc;
end architecture Bad;
```

```
architecture Good of Vdc is
quantity v across i through p to m;
begin
v == dc;
end architecture Good;
```

*La première architecture ne compilera pas: il y a une équation et la somme des quantités **through**, de mode **out,** et libres, est nulle. La seconde architecture compilera: il y a maintenant une quantité **through**. Qu'elle n'apparaisse pas dans une équation explicite n'empêche qu'elle participe à la résolution avec des valeurs traçables.*

Attention: le fait que cette condition soit remplie ne signifie pas que le système va converger: il s'agit là d'autres conditions dynamiques et de valeurs initiales qui dépendent du problème à traiter.

13.5 Break

L'instruction break est une instruction concurrente qui interagit avec le noyau analogique, et comme les instructions concurrentes elle a un processus équivalent (§9.1 page 67) qui ramène la question à son équivalent séquentiel.
Elle est vue dans le §8.3.3 page 65 pour ce qui est de son fonctionnement, et listée avec les autres instructions séquentielles §12.3.4.1 page 100.

13.6 Limit

Il peut être nécessaire de dire au noyau de simulation une limite maximum du pas temporel entre deux points de calcul. Autrement dit, on peut ne pas faire confiance à ses heuristiques pour obtenir la précision souhaitée.

BNF (voir définition page 9) :

```
step_limit_specification ::= limit quantity_specification
                             with real_expression ;
```

*La specification de quantité peut être faite par son nom, ou par les mots-clés **all** (toutes les quantités de ce type) et **others** (celles pas encore mentionnées). Cette déclaration de limite ne peut se faire que dans la région où est déclarée la quantité.*

Exemples:

```
limit Q1,Q2: current with 1/10*F;

limit all: voltage with L;
```

14 Généricité et Génération de Code

14.1 Généricité

Il est possible de nommer des constantes dont on ne donne la valeur qu'au dernier moment, c'est-à-dire

- Au moment de la simulation ou de la synthèse
- Au moment de configurer un composant (§10.8.3 page 81). La valeur fournie peut alors elle-même être le nom d'un argument générique qui sera donné plus tard, transitivement. L'important étant qu'au moment de la simulation ou de la synthèse, tous les trous soient bouchés.

La déclaration des génériques se fait dans la déclaration d'entité ou dans la déclaration de composant. C'est la clause illustrée par cet exemple:

```
generic (A,B:INTEGER:=8; C: BIT);
```

On voit qu'on y déclare les noms des arguments, leur type et leur éventuelle valeur par défaut à prendre si aucune autre n'est donnée.
Aussitôt déclarée, cette constante peut être utilisée et en particulier dans la déclaration des ports qui suit immédiatement:

```
generic (N:INTEGER);
port (…S: BIT_VECTOR(N-1 downto 0)..);
```

On fournit les valeurs par la clause **generic map**, qui apparaît dans les instanciations de composants (§10.8.2 page 80) et dans les différentes configurations (§10.8.3 page 81) et dont voici une illustration:

```
… generic map (8) …
```

La manière d'associer des valeurs aux arguments formels suit les règles décrites §10.3 page 75 (l'association). Toutefois il y a une certaine complexité liée à la situation où des valeurs par défaut différentes seraient attribuées au composant générique et à l'entité générique qu'on lui attribue. Bien que ce soit spécifié très précisément dans la norme, la chose est spécialement illisible et il vaut mieux éviter ce cas là en donnant explicitement les valeurs génériques.

L'effet du **generic map** est de remplacer brutalement toutes les occurrences de l'argument générique par sa valeur dans le code qui en dépend. Cela est utile

- pour éviter de mettre dans le texte des constantes périssables
- pour paramétrer des instructions de génération (§14.2 ci-dessous)

On trouvera les constructions équivalentes §9.2 page 69.

Noter que, pour pouvoir faire la rétro-annotation (imposer des délais découverts après synthèse dans le modèle qui a servi à la synthèse, afin de le resimuler), il est possible d'écraser un **generic map** fait dans une configuration par un autre fait dans la configuration du niveau supérieur.
Noter aussi qu'il est possible de ne donner dans un **generic map** que quelques-uns des arguments génériques, et pas les autres. Il faudra cependant que tous les arguments génériques aient une valeur au moment du lancement de la simulation ou de la synthèse. Si par exemple on a une entité avec deux arguments génériques et un composant de même, on peut configurer le composant en fixant un seul des deux génériques, et l'instancier en fixant l'autre. Si l'entité a deux arguments génériques et le composant un seul, on peut configurer le composant en fixant l'argument de l'entité qui n'est pas générique pour lui, et fixer le sien lors de l'instanciation. Toutes les combinaisons sont possibles du moment qu'à la fin, chaque argument générique sache combien il vaut.

14.2 Génération de code

Certaines parties du code VHDL peuvent être crées algorithmiquement. Il s'agit d'algorithmes qui se déroulent avant le temps zéro de la simulation, le code ainsi créé est ensuite régulièrement compilé et traité par l'outil qui s'en sert. Il y a deux instructions de génération qui apparaissent dans un contexte concurrent, c'est-à-dire uniquement dans les zones d'instruction des architectures et anecdotiquement des blocs (après le **begin**). On peut ainsi créer des structures régulières (avec la boucle) ou pseudo régulières (en y introduisant des exceptions avec le test). On peut créer du code conditionnel destiné à disparaître après débogage (voir §2.3 page 11).

Code conditionnel:

```
label: if condition generate
[begin optionnel, versions récentes]
…
end generate;
```

(pas de branche **else**, si on en veut une il faut écrire le test inverse.)

Code répétitif:

```
label: for I in range generate
[begin optionnel, versions récentes]
…
end generate;
```

L'indice de boucle (ici I) est implicitement déclaré par l'instruction, n'est visible ni avant ni après et est vu comme une constante dans la boucle.

Le label est obligatoire

L'effet de ces instructions est le remplacement pur et simple du code qui est encapsulé par le calcul correspondant. Cela est détaillé §9.2 page 69.

15 Attributs

[(X)] signifie que l'argument est optionnel, et, si omis, vaudra 0 pour une dimension qui se comptent à partir de 0, *un delta* pour un temps,
etc. $A^{(N)}$ signifie: la Nième dimension de A.

Rend un type

```
T'BASE        le type de base du type T
```

Rendent des valeurs

```
T'LEFT        la valeur la plus à gauche du type T (la plus grande si downto)
T'RIGHT       la valeur la plus à droite du type T (la plus petite si downto)
T'HIGH        la valeur la plus grande du type T
T'LOW         la valeur la plus petite du type T
T'ASCENDING   boolean true si le range de T est defini avec to
A'LENGTH[(N)] nombre d'éléments de la Nème dimension de A
A'ASCENDING[(N)] Boolean, true si le range de la Nème dimension de A défini avec to
```

Rendent des fonctions

```
T'IMAGE(X)    représentation en string de X du type T.
T'VALUE(X)    valeur du type T lue depuis la string X.
T'POS(X)    position (integer) de X dans le type discret T (début à 0)
T'VAL(X)    valeur du type discret T à la position X (début 0).
T'SUCC(X)     successeur de X dans le type discret T
T'PRED(X)     prédécesseur de X dans le type discret T
T'LEFTOF(X)   valeur à gauche de X dans le type discret T
T'RIGHTOF(X)  valeur à droite de X dans le type discret T
A'LEFT[(N)]   valeur la plus à gauche de l'index de Nème dimension du [type/objet] tableau A
A'RIGHT[(N)]  valeur la plus à droite (idem supra)
A'HIGH[(N)]   valeur la plus haute de l'index de Nème dimension du [type/objet] tableau A
A'LOW[(N)]    valeur la plus basse (idem supra)
```

Rendent des étendues (*ranges*)

A'RANGE[(N)] l'étendue (range) $A^{[(N)]}$'LEFT **to** $A^{[(N)]}$'RIGHT ou $A^{[(N)]}$'LEFT **downto** $A^{[(N)]}$'RIGHT

```
A'REVERSE_RANGE[(N)]  idem en inversant to et downto
```

Rendent des signaux

```
S'DELAYED[(t)] valeur du signal S au temps now – t. t doit être statique.
S'STABLE[(t)]  boolean, true s'il n'y a pas eu d'événement sur le signal S depuis un temps t
S'QUIET[(t)]   boolean, true s'il n'y a pas eu d'événement sur le signal S depuis un temps t
S'TRANSACTION  bit qui change de valeur chaque fois que S est actif.
```

Rendent des fonctions

```
S'EVENT    boolean, true s'il y a un événement sur le signal S
S'ACTIVE   boolean, true s'il y a une activité sur le signal S
S'LAST_EVENT le temps depuis le dernier événement sur le signal S
S'LAST_ACTIVE  le temps depuis la dernière activité sur le signal S
S'LAST_VALUE valeur précédente de S
S'DRIVING    Boolean, false si le driver de S contient une transaction nulle.
```

S'DRIVING_VALUE valeur du driver de S dans le processus [equivalent].

Rendent des valeurs

E'SIMPLE_NAME string, le nom de l'objet E
E'INSTANCE_NAME string, le nom hiérarchique de l'objet E
E'PATH_NAME string, le nom du chemin hiérarchique qui va à E

[AMS] Rendent des quantités

S'RAMP([TR[,TF]])rampe basée sur le signal S dont on donne les temps de montée et de descente. ***Break*** *implicite si on les omet.*
S'SLEW([SR][,SF]])rampe basée sur le signal S dont on donne les pentes de montée et de descente. ***Break*** *implicite si on les omet.*
Q'DOT derivée de la quantité Q par rapport au temps
Q'INTEG intégrale sur la quantité Q de 0 au temps courant / temps
Q'DELAYED(T) meme quantité décalée de T (statique)
Q'ZOH(T[,TD])Echantilloneur idéal sur la quantité Q, période T; TD = délai initial (0 si omis)
Q'LTF(num,den)transformée de Laplace de la quantité Q.
Q'ZTF(num,den,T[,initial_delay]) Transformée en Z de la quantité Q
Q'SLEW([SR][,SF]]) rampe basée sur la quantité Q dont on donne les pentes de montée et de descente. . ***Break*** *implicite si on les omet.*
T'CONTRIBUTION somme des quantités ***through*** *du terminal T, sa contribution à l'équipotentielle.*
T'REFERENCE quantité ***across*** *entre le terminal T et la référence de la nature de T*

[AMS] Rendent des types

N'ACROSS type ***across*** *de la nature N*
N'THROUGH type ***through*** *de la nature N*

[AMS] Rend un terminal

N'REFERENCE Terminal de référence de la nature N

[AMS] Rend une valeur (string)

Q'TOLERANCE nom du groupe de tolérance de la quantité Q.
T'TOLERANCE nom du groupe de tolérance du terminal T.

[AMS] Rend un signal

*Q'ABOVE(E) T*RUE si Q > E

16 Paquetages Standards et Usuels

Les paquetages ci-après sont, soit des paquetages standardisés (STANDARD, TEXTIO, STD_LOGIC_1164), soit des paquetages mis dans le domaine public par un vendeur, Synopsys en l'occurrence, (STD_LOGIC_TEXTIO, STD_LOGIC_ARITH, STD_LOGIC_SIGNED, STD_LOGIC_UNSIGNED), soit enfin la dernière version des versions de travail lorsque la version finale est soumise à copyright (les paquetages mathématiques). Même lorsqu'ils ne sont pas normalisés par l'IEEE, l'usage est de les mettre dans la bibliothèque de nom symbolique IEEE, hormis les deux premiers évidemment. *[AMS]Enfin, pour AMS, les disciplines qui décrivent les domaines physiques sont dans une bibliothèque DISCIPLINES.*

16.1 Paquetages indispensables

16.1.1 STD.STANDARD

En italique les lignes spécifiques AMS.

```
package standard is
      type boolean is (false,true);
      type bit is ('0', '1');

      type character is (
            nul, soh, stx, etx, eot, enq, ack, bel, bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
           dle, dc1, dc2, dc3, dc4, nak, syn, etb, can, em,  sub, esc, fsp, gsp, rsp, usp,
           ' ', '!', '"', '#', '$', '%', '&', ''', '(', ')', '*', '+', ',', '-', '.', '/',
           '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?',
           '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
           'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_',
           '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
           'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~', del,
           c128, c129, ……c143,
           c144, c145, ……c159,
           ' ', '¡', '¢', '£', '¤', '¥', '¦', '§', '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
           '°', '±', '²', '³', '´', 'µ', '¶', '·', '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
           'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç', 'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
           'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×', 'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
           'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç', 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
           'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷', 'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ'
                        );

      type severity_level is (note, warning, error, failure);

      type integer is range -2147483648 to 2147483647;
      type real is range -1.0E308 to 1.0E308;
```

```
      type time is range -2147483647 to 2147483647
            units
                  fs;
                  ps = 1000 fs;
                  ns = 1000 ps;
                  us = 1000 ns;
                  ms = 1000 us;
                  sec = 1000 ms;
                  min = 60 sec;
                  hr = 60 min;
            end units;
        type DOMAIN_TYPE is (QUIESCENT_DOMAIN,
                             TIME_DOMAIN,
                             FREQUENCY_DOMAIN);
        signal DOMAIN: DOMAIN_TYPE:=QUIESCENT_DOMAIN;
      subtype delay_length is time range 0 fs to time'high;
      impure function now return delay_length;
       impure function now return real;
              -- deviendra pure dans la prochaine révision
       function frequency return real;
      subtype natural is integer range 0 to integer'high;
      subtype positive is integer range 1 to integer'high;
       type real_vector is array(natural range <>) of real;
      type string is array (positive range <>) of character;
      type bit_vector is array (natural range <>) of bit;
      type file_open_kind is (read_mode, write_mode, append_mode);
      type file_open_status is (open_ok,
                                status_error,
                                name_error,
                                mode_error);
      attribute foreign : string;
end standard;
```

16.1.2 STD.TEXTIO

```
package TEXTIO is
    type LINE is access string;
    type TEXT is file of string;
    type SIDE is (right, left);
    subtype WIDTH is natural;
    file input : TEXT open read_mode is "STD_INPUT";
    file output : TEXT open write_mode is "STD_OUTPUT";
    procedure READLINE(file f: TEXT; L: out LINE);
    procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out bit);
    procedure READ(L:inout LINE; VALUE: out bit_vector;
                   GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out bit_vector);
    procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out character);
    procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out integer);
    procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out real);
    procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out string);
    procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out time);
    procedure WRITELINE(file f : TEXT; L : inout LINE);
    procedure WRITE(L : inout LINE; VALUE : in bit;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in bit_vector;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in character;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in integer;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in real;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0;
            DIGITS: in NATURAL := 0);
    procedure WRITE(L : inout LINE; VALUE : in string;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in time;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0;
            UNIT: in TIME := ns);
end;
```

16.1.3 IEEE.STD_LOGIC_1164

```
PACKAGE std_logic_1164 IS
    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with
    -- the resolution function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;
    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
    ------------------------------------------------------------------
    -- common subtypes
    ------------------------------------------------------------------
    SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1';
                    -- ('X','0','1')
    SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z';
                    -- ('X','0','1','Z')
    SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1';
                    -- ('U','X','0','1')
    SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z';
                    -- ('U','X','0','1','Z')
    ------------------------------------------------------------------
    -- overloaded logical operators
    ------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
    ------------------------------------------------------------------
    -- vectorized overloaded logical operators
    ------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
```

```
    FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
    FUNCTION "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
    FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
    ---------------------------------------------------------------------
    -- conversion functions
    ---------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic; xmap : BIT := '0') RETURN BIT;
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0')
                            RETURN BIT_VECTOR;
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0')
                            RETURN BIT_VECTOR;
    FUNCTION To_StdULogic  ( b : BIT ) RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR) RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
                               RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR) RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector)
                               RETURN std_ulogic_vector;
    ---------------------------------------------------------------------
    -- strength strippers and type convertors
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( b : BIT               ) RETURN  X01;
    FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;
    FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;
    ---------------------------------------------------------------------
    -- edge detection
    ---------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    ---------------------------------------------------------------------
    -- object contains an unknown
    ---------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic) RETURN  BOOLEAN;
END std_logic_1164;
```

16.1.4 IEEE.STD_LOGIC_TEXTIO

Une copie de TEXTIO mais le type BIT y est remplacé par le type STD_LOGIC. Paquetage non standard.

```
use STD.textio.all;
library IEEE;
use IEEE.std_logic_1164.all;
package STD_LOGIC_TEXTIO is

      -- Read and Write procedures for STD_ULOGIC and STD_ULOGIC_VECTOR
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC;
                     GOOD: out BOOLEAN);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                     GOOD: out BOOLEAN);
      procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      -- Read and Write procedures for STD_LOGIC_VECTOR
      procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                     GOOD: out BOOLEAN);
      procedure WRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      --
      -- Read and Write procedures for Hex and Octal values.
      -- The values appear in the file as a series of characters
      -- between 0-F (Hex), or 0-7 (Octal) respectively.
      --
      -- Hex
      procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure HWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure HWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      -- Octal
      procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure OWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure OWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
end STD_LOGIC_TEXTIO;
```

16.2 L'arithmétique entière sur BIT_VECTOR et STD_LOGIC_VECTOR

Quelques paquetages plus ou moins standardisés ou consensuels (avec les sources disponibles) permettent de faire de l'arithmétique entière sur BIT_VECTOR ou (surtout) sur STD_LOGIC_VECTOR.
À part les primitives qui sont faciles à appréhender, la question essentielle est celle de la représentation des entiers signés. Deux approches incompatibles sont disponibles:

- Faire dans un paquetage –NUMERIC_BIT, STD_LOGIC_ARITH- deux types incompatibles SIGNED et UNSIGNED, définis exactement comme BIT_VECTOR ou STD_LOGIC_VECTOR: dans ce cas le langage dit qu'on a le droit de convertir l'un dans l'autre, les types sont dit apparentés (*closely related*). L'arithmétique est définie en surchargeant lourdement tous les opérateurs pour permettre toutes les combinaisons. Ainsi, pour additionner deux STD_LOGIC_VECTOR nommés STDV1 et STDV2, et mettre le résultat dans STDV3, on écrira:
  STD3 <= UNSIGNED(STDV1) + UNSIGNED(STDV2);
- Faire deux paquetages différents –STD_LOGIC_SIGNED, STD_LOGIC_UNSIGNED- définissant l'un des opérations signées, l'autre des opérations non signées sur le type STD_LOGIC_VECTOR. L'avantage est qu'il n'y a plus de conversion. L'inconvénient est qu'on ne peut voir par la clause **use** qu'un seul de ces paquetages directement. Si on veut mélanger les deux, l'autre sera accessible seulement par la notation pointée.

16.2.1 IEEE.NUMERIC_BIT

```
package NUMERIC_BIT is
      type UNSIGNED is array (NATURAL range <>) of BIT;
      type SIGNED is array (NATURAL range <>) of BIT;
      function "abs" (ARG:SIGNED) return SIGNED;
      function "-" (ARG:SIGNED) return SIGNED;
      function "+" (L,R:UNSIGNED) return UNSIGNED;
      function "+" (L,R:SIGNED) return SIGNED;
      function "+" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
      function "+" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
      function "+" (L:INTEGER; R:SIGNED) return SIGNED;
      function "+" (L:SIGNED; R:INTEGER) return SIGNED;
      function "-" (L,R:UNSIGNED) return UNSIGNED;
      function "-" (L,R:SIGNED) return SIGNED;
      function "-" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
      function "-" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
      function "-" (L:SIGNED; R:INTEGER) return SIGNED;
      function "-" (L:INTEGER; R:SIGNED) return SIGNED;
      function "*" (L,R:UNSIGNED) return UNSIGNED;
      function "*" (L,R:SIGNED) return SIGNED;
      function "*" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
      function "*" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
      function "*" (L:SIGNED; R:INTEGER) return SIGNED;
      function "*" (L:INTEGER; R:SIGNED) return SIGNED;
      function "/" (L,R:UNSIGNED) return UNSIGNED;
      function "/" (L,R:SIGNED) return SIGNED;
      function "/" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
      function "/" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
      function "/" (L:SIGNED; R:INTEGER) return SIGNED;
```

```
function "/" (L:INTEGER; R:SIGNED) return SIGNED;
function "rem" (L,R:UNSIGNED) return UNSIGNED;
function "rem" (L,R:SIGNED) return SIGNED;
function "rem" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
function "rem" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "rem" (L:SIGNED; R:INTEGER) return SIGNED;
function "rem" (L:INTEGER; R:SIGNED) return SIGNED;
function "mod" (L,R:UNSIGNED) return UNSIGNED;
function "mod" (L,R:SIGNED) return SIGNED;
function "mod" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
function "mod" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "mod" (L:SIGNED; R:INTEGER) return SIGNED;
function "mod" (L:INTEGER; R:SIGNED) return SIGNED;
function ">" (L,R:UNSIGNED) return BOOLEAN;
function ">" (L,R:SIGNED) return BOOLEAN;
function ">" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function ">" (L:INTEGER; R:SIGNED) return BOOLEAN;
function ">" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function ">" (L:SIGNED; R:INTEGER) return BOOLEAN;
function "<" (L,R:UNSIGNED) return BOOLEAN;
function "<" (L,R:SIGNED) return BOOLEAN;
function "<" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "<" (L:INTEGER; R:SIGNED) return BOOLEAN;
function "<" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function "<" (L:SIGNED; R:INTEGER) return BOOLEAN;
function "<=" (L,R:UNSIGNED) return BOOLEAN;
function "<=" (L,R:SIGNED) return BOOLEAN;
function "<=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "<=" (L:INTEGER; R:SIGNED) return BOOLEAN;
function "<=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function "<=" (L:SIGNED; R:INTEGER) return BOOLEAN;
function ">=" (L,R:UNSIGNED) return BOOLEAN;
function ">=" (L,R:SIGNED) return BOOLEAN;
function ">=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function ">=" (L:INTEGER; R:SIGNED) return BOOLEAN;
function ">=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function ">=" (L:SIGNED; R:INTEGER) return BOOLEAN;
function "=" (L,R:UNSIGNED) return BOOLEAN;
function "=" (L,R:SIGNED) return BOOLEAN;
function "=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "=" (L:INTEGER; R:SIGNED) return BOOLEAN;
function "=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function "=" (L:SIGNED; R:INTEGER) return BOOLEAN;
function "/=" (L,R:UNSIGNED) return BOOLEAN;
function "/=" (L,R:SIGNED) return BOOLEAN;
function "/=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "/=" (L:INTEGER; R:SIGNED) return BOOLEAN;
function "/=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function "/=" (L:SIGNED; R:INTEGER) return BOOLEAN;
function SHIFT_LEFT (ARG:UNSIGNED; COUNT:NATURAL)
return UNSIGNED;
function SHIFT_RIGHT (ARG:UNSIGNED; COUNT:NATURAL)
return UNSIGNED;
function SHIFT_LEFT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function SHIFT_RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function ROTATE_LEFT (ARG:UNSIGNED; COUNT:NATURAL)
return UNSIGNED;
function ROTATE_RIGHT (ARG:UNSIGNED; COUNT:NATURAL)
return UNSIGNED;
function ROTATE_LEFT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function ROTATE_RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
```

```
      function "sll" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
      function "sll" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
      function "srl" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
      function "srl" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
      function "rol" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
      function "rol" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
      function "ror" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
      function "ror" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
      function RESIZE (ARG:SIGNED; NEW_SIZE:NATURAL) return SIGNED;
      function RESIZE (ARG:UNSIGNED;NEW_SIZE:NATURAL) return UNSIGNED;
      function TO_INTEGER (ARG:UNSIGNED) return NATURAL;
      function TO_INTEGER (ARG:SIGNED) return INTEGER;
      function TO_UNSIGNED (ARG,SIZE:NATURAL) return UNSIGNED;
      function TO_SIGNED (ARG:INTEGER; SIZE:NATURAL) return SIGNED;
      function "not" (L:UNSIGNED) return UNSIGNED;
      function "and" (L,R:UNSIGNED) return UNSIGNED;
      function "or" (L,R:UNSIGNED) return UNSIGNED;
      function "nand" (L,R:UNSIGNED) return UNSIGNED;
      function "nor" (L,R:UNSIGNED) return UNSIGNED;
      function "xor" (L,R:UNSIGNED) return UNSIGNED;
      function "xnor" (L,R:UNSIGNED) return UNSIGNED;
      function "not" (L:SIGNED) return SIGNED;
      function "and" (L,R:SIGNED) return SIGNED;
      function "or" (L,R:SIGNED) return SIGNED;
      function "nand" (L,R:SIGNED) return SIGNED;
      function "nor" (L,R:SIGNED) return SIGNED;
      function "xor" (L,R:SIGNED) return SIGNED;
      function "xnor" (L,R:SIGNED) return SIGNED;
      function RISING_EDGE (signal S:BIT) return BOOLEAN;
      function FALLING_EDGE(signal S:BIT) return BOOLEAN;
end package NUMERIC_BIT;
```

16.2.2 IEEE.STD_LOGIC_ARITH

```
library IEEE;
use IEEE.std_logic_1164.all;
package std_logic_arith is
    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;
    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: INTEGER) return SIGNED;
    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: INTEGER) return SIGNED;
    function "-"(L: INTEGER; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED) return SIGNED;
    function "-"(L: SIGNED) return SIGNED;
```

```
function "ABS"(L: SIGNED) return SIGNED;
function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED; R: SIGNED) return SIGNED;
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;
```

```
    function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function CONV_INTEGER(ARG: INTEGER) return INTEGER;
    function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
    function CONV_INTEGER(ARG: SIGNED) return INTEGER;
    function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;
    function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return UNSIGNED;
    function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return SIGNED;
    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
-- zero extend STD_LOGIC_VECTOR (ARG) to SIZE, SIZE < 0 is same as SIZE = 0
    -- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER)
                 return STD_LOGIC_VECTOR;
-- sign extend STD_LOGIC_VECTOR (ARG) to SIZE, SIZE < 0 is same as SIZE = 0
    -- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return
STD_LOGIC_VECTOR;
end Std_logic_arith ;
```

16.2.3 IEEE.STD_LOGIC_SIGNED

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_SIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC)
                return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC)
                return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "ABS"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR;
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR)
              return INTEGER;
end STD_LOGIC_SIGNED;
```

16.2.4 IEEE.STD_LOGIC_UNSIGNED

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
package STD_LOGIC_UNSIGNED is
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER)
                 return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC)
                 return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC)
                 return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR)
                 return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR)
                  return STD_LOGIC_VECTOR;
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;
end STD_LOGIC_UNSIGNED;
```

16.3 L'arithmétique réelle et complexe pour l'analogique

16.3.1 IEEE.MATH_REAL

```
package MATH_real is

constant MATH_E: real:= 2.71828_18284_59045_23536; -- Value of e
constant MATH_1_OVER_E: real:= 0.36787_94411_71442_32160; -- Value of 1/e
constant MATH_PI: real:= 3.14159_26535_89793_23846; -- Value of pi
constant MATH_2_PI: real := 6.28318_53071_79586_47693; -- Value of 2*pi
constant MATH_1_OVER_PI: real := 0.31830_98861_83790_67154;
-- Value of 1/pi
constant MATH_PI_OVER_2: real := 1.57079_63267_94896_61923;
-- Value of pi/2
constant MATH_PI_OVER_3: real := 1.04719_75511_96597_74615;
-- Value of pi/3
constant MATH_PI_OVER_4: real := 0.78539_81633_97448_30962;
-- Value of pi/4
constant MATH_3_PI_OVER_2: real := 4.71238_89803_84689_85769;
-- Value of 3*pi/2
constant MATH_LOG_OF_2: real := 0.69314_71805_59945_30942;
-- Natural log of 2
constant MATH_LOG_OF_10: real := 2.30258_50929_94045_68402;
-- Natural log of 10
constant MATH_LOG2_OF_E: real := 1.44269_50408_88963_4074;
-- Log base 2 of e
constant MATH_LOG10_OF_E: real 0.43429_44819_03251_82765;
-- Log base 10 of e
constant MATH_SQRT_2: real := := 1.41421_35623_73095_04880;
-- square root of 2
constant MATH_1_OVER_SQRT_2: real := 0.70710_67811_86547_52440;
-- 1/SQRT(2)
constant MATH_SQRT_PI: real := 1.77245_38509_05516_02730;
-- Square root of pi
constant MATH_DEG_TO_RAD: real:= 0.01745_32925_19943_29577;
   -- Conversion factor from degree to radian
constant MATH_RAD_TO_DEG: real:= 57.29577_95130_82320_87680;
   -- Conversion factor from radian to degree
function SIGN (X: in real) return real;
-- Returns 1.0 if X > 0.0; 0.0 if X = 0.0; -1.0 if X < 0.0
-- Special values: None
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(SIGN(X)) <= 1.0
-- Notes: None
function CEIL (X: in real) return real;
-- Returns smallest INTEGER value (as real) not less than X
-- Special values: None
-- Domain: X in real
-- Error conditions: None
-- Range: CEIL(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)
function FLOOR (X: in real) return real;
-- Returns largest integer value (as real) not greater than X
-- Special values: FLOOR(0.0) = 0.0
```

```
-- Domain: X in real
-- Error conditions: None
-- Range: FLOOR(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)
function ROUND (X: in real) return real;
-- Rounds X to the nearest integer value (as real). If X is
-- halfway between two integers, rounding is away from 0.0
-- Special values: ROUND(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ROUND(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)
function TRUNC (X: in real) return real;
-- Truncates X towards 0.0 and returns truncated value
-- Special values: TRUNC(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: TRUNC(X) is mathematically unbounded
-- Notes: Implementations have to support at least the
-- domain ABS(X) < real(INTEGER'HIGH)
function "MOD" (X, Y: in real) return real;
-- Returns floating point modulus of X/Y, with the same sign
-- as Y, and absolute value less than the absolute value of
-- Y, and for some INTEGER value N the result satisfies th
-- relation X = Y*N + MOD(X,Y)
-- Special values: None
-- Domain: X in real; Y in real and Y /= 0.0
-- Error conditions: Error if Y = 0.0
-- Range: ABS(MOD(X,Y)) < ABS(Y)
-- Notes: None
function REALMAX (X, Y : in real) return real;
-- Returns the algebraically larger of X and Y
-- Special values: REALMAX(X,Y) = X when X = Y
-- Domain: X in real; Y in real
-- Error conditions: None
-- Range: REALMAX(X,Y) is mathematically unbounded
-- Notes: None
function REALMIN (X, Y : in real) return real;
-- Returns the algebraically smaller of X and Y
-- Special values: REALMIN(X,Y) = X when X = Y
-- Domain: X in real; Y in real
-- Error conditions: None
-- Range: REALMIN(X,Y) is mathematically unbounded
-- Notes: None
procedure UNIFORM (
variable SEED1, SEED2: inout positive;
variable X: out real);
-- Returns, in X, a pseudo-random number with uniform
-- distribution in the open interval (0.0, 1.0)
-- Special values: None
-- Domain: 1 <= SEED1 <= 2147483562;
-- 1 <= SEED2 <= 2147483398
-- Error conditions:
-- Error if SEED1 or SEED2 outside of valid domain
-- Range: 0.0 < X < 1.0
-- Notes:
-- a) The semantics for this function are described by the
-- algorithm published by Pierre L'Ecuyer in
-- "Communications of the ACM," vol. 31, no. 6, June 1988,
```

```
-- pp. 742-774.
-- The algorithm is based on the combination of two
-- multiplicative linear congruential generators for 32-bit
-- platforms.
-- b) Before the first call to UNIFORM, the seed values
-- (SEED1, SEED2) have to be initialized to values in the
-- range [1, 2147483562] and [1, 2147483398] respectively.
-- The seed values are modified after each call to UNIFORM.
-- c) This random number generator is portable for 32-bit
-- computers, and it has a period of ~2.30584*(10**18) for
-- each set of seed values.
-- d) For information on spectral tests for the algorithm, refer
-- to the L'Ecuyer article.
function SQRT (X: in real) return real;
-- Returns square root of X
-- Special values: SQRT(0.0) = 0.0, SQRT(1.0) = 1.0
-- Domain: X >= 0.0
-- Error conditions: Error if X < 0.0
-- Range: SQRT(X) >= 0.0
-- Notes: The upper bound of the reachable range of SQRT
-- is approximately given by SQRT(X) <= SQRT(real'HIGH)
function CBRT (X: in real) return real;
-- Returns cube root of X
-- Special values:
-- CBRT(0.0) = 0.0, CBRT(1.0) = 1.0, CBRT(-1.0) = -1.0
-- Domain: X in real
-- Error conditions: None
-- Range: CBRT(X) is mathematically unbounded
-- Notes: The reachable range of CBRT is approximately
-- given by: ABS(CBRT(X)) <= CBRT(real'HIGH)
function "**" (X: in integer; Y: in real) return real;
-- Returns Y power of X ==> X**Y
-- Special values:
-- X**0.0 = 1.0; X /= 0
-- 0**Y = 0.0; Y > 0.0
-- X**1.0 = real(X); X >= 0
-- 1**Y = 1.0
-- Domain:
-- X > 0
-- X = 0 for Y > 0.0
-- X < 0 for Y = 0.0
-- Error conditions:
-- Error if X < 0 and Y /= 0
-- Error if X = 0 and Y <= 0.0
-- Range: X**Y >= 0.0
-- Notes: The upper bound of the reachable range for "**" is
-- approximately given by: X**Y <= real'HIGH
function "**" (X: in real; Y: in real) return real;
-- Returns Y power of X ==> X**Y
-- Special values:
-- X**0.0 = 1.0; X /= 0
-- 0**Y = 0.0; Y > 0.0
-- X**1.0 = real(X); X >= 0
-- 1.0**Y = 1.0
-- Domain:
-- X > 0.0
-- X = 0.0 for Y > 0.0
-- X < 0.0 for Y = 0.0
-- Error conditions:
-- Error if X < 0.0 and Y /= 0
-- Error if X = 0.0 and Y <= 0.0
```

```
-- Range: X**Y >= 0.0
-- Notes: The upper bound of the reachable range for "**" is
-- approximately given by: X**Y <= real'HIGH
function EXP (X: in real) return real;
-- Returns e**X; where e = MATH_E
-- Special values:
-- EXP(0.0) = 1.0
-- EXP(1.0) = MATH_E
-- EXP(-1.0) = MATH_1_OVER_E
-- EXP(X) = 0.0 for X <= -LOG(real'HIGH)
-- Domain: X in real such that EXP(X) <= real'HIGH
-- Error conditions: Error if X > LOG(real'HIGH)
-- Range: EXP(X) >= 0.0
-- Notes: The usable domain of EXP is approximately given
-- by: X <= LOG(real'HIGH)
function LOG (X: in real) return real;
-- Returns natural logarithm of X
-- Special values: LOG(1.0) = 0.0, LOG(MATH_E) = 1.0
-- Domain: X > 0.0
-- Error conditions: Error if X <= 0.0
-- Range: LOG(X) is mathematically unbounded
-- Notes: The reachable range of LOG is approximately
-- given by: LOG(0+) <= LOG(X) <= LOG(real'HIGH)
function LOG2 (X: in real) return real;
-- Returns logarithm base 2 of X
-- Special values: LOG2(1.0) = 0.0, LOG2(2.0) = 1.0
-- Domain: X > 0
-- Error conditions: Error if X <= 0.0
-- Range: LOG2(X) is mathematically unbounded
-- Notes: The reachable range of LOG2 is approximately
-- given by: LOG2(0+) <= LOG2(X) <= LOG2(real'HIGH)
function LOG10 (X: in real) return real;
-- Returns logarithm base 10 of X
-- Special values: LOG10(1.0) = 0.0, LOG10(10.0) = 1.0
-- Domain: X > 0.0
-- Error conditions: Error if X <= 0.0
-- Range: LOG10(X) is mathematically unbounded
-- Notes: The reachable range of LOG10 is approximately
-- given by: LOG10(0+) <= LOG10(X) <= LOG10(real'HIGH)
function LOG (X: in real; BASE: in real) return real;
-- Returns logarithm base BASE of X
-- Special values:
-- LOG(1.0, BASE) = 0.0, LOG(BASE, BASE) = 1.0
-- Domain: X > 0.0, BASE > 0.0, BASE /= 1.0
-- Error conditions:
-- Error if X <= 0.0
-- Error if BASE <= 0.0 or if BASE = 1.0
-- Range: LOG(X, BASE) is mathematically unbounded
-- Notes:
-- a) When BASE > 1.0, the reachable range of LOG is
-- approximately given by:
-- LOG(0+, BASE) <= LOG(X, BASE) <= LOG(real'HIGH,
-- BASE)
-- b) When 0.0 < BASE < 1.0, the reachable range of LOG is
-- approximately given by:
-- LOG(real'HIGH, BASE) <= LOG(X, BASE) <= LOG(0+,
-- BASE)
function SIN (X: in real) return real;
-- Returns sine of X; X in radians
-- Special values:
-- SIN(X) = 0.0 for X = k*MATH_PI
```

```
-- SIN(X) = 1.0 for X = (4*k+1)*MATH_PI_OVER_2
-- SIN(X) = -1.0 for X = (4*k+3)*MATH_PI_OVER_2
-- where k is an integer
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(SIN(X)) <= 1.0
-- Notes: For larger values of ABS(X), degraded accuracy is allowed
function COS ( X: in real) return real;
-- Returns cosine of X; X in radians
-- Special values:
-- COS(X) = 0.0 for X = (2*k+1)*MATH_PI_OVER_2
-- COS(X) = 1.0 for X = (2*k)*MATH_PI
-- COS(X) = -1.0 for X = (2*k+1)*MATH_PI
-- where k is an INTEGER
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(COS(X)) <= 1.0
-- Notes: For larger values of ABS(X), degraded accuracy is allowed
function TAN (X: in real) return real;
-- Returns tangent of X; X in radians
-- Special values:
-- TAN(X) = 0.0 for X = k*MATH_PI, where k is an integer
-- Domain: X in real and
-- X /= (2*k+1)*MATH_PI_OVER_2,
-- where k is an INTEGER
-- Error conditions:
-- Error if X = ((2*k+1) * MATH_PI_OVER_2)
-- where k is an integer
-- Range: TAN(X) is mathematically unbounded
-- Notes: For larger values of ABS(X), degraded accuracy is allowed
function ARCSIN (X: in real) return real;
-- Returns inverse sine of X
-- Special values:
-- ARCSIN(0.0) = 0.0
-- ARCSIN(1.0) = MATH_PI_OVER_2
-- ARCSIN(-1.0) = -MATH_PI_OVER_2
-- Domain: ABS(X) <= 1.0
-- Error conditions: Error if ABS(X) > 1.0
-- Range: ABS(ARCSIN(X) <= MATH_PI_OVER_2
-- Notes: None
function ARCCOS (X: in real) return real;
-- Returns inverse cosine of X
-- Special values:
-- ARCCOS(1.0) = 0.0
-- ARCCOS(0.0) = MATH_PI_OVER_2
-- ARCCOS(-1.0) = MATH_PI
-- Domain: ABS(X) <= 1.0
-- Error conditions: Error if ABS(X) > 1.0
-- Range: 0.0 <= ARCCOS(X) <= MATH_PI
-- Notes: None
function ARCTAN (Y : in real) return real;
-- Returns the value of the angle in radians of the point
-- (1.0, Y), which is in rectangular coordinates
-- Special values: ARCTAN(0.0) = 0.0
-- Domain: Y in real
-- Error conditions: None
-- Range: ABS(ARCTAN(Y)) <= MATH_PI_OVER_2
-- Notes: None
function ARCTAN (Y : in real; X: in real) return real;
-- Returns the principal value of the angle in radians of
-- the point (X, Y), which is in rectangular coordinates
```

```
-- Special values:
-- ARCTAN(0.0, X) = 0.0 if X > 0.0
-- ARCTAN(0.0, X) = MATH_PI if X < 0.0
-- ARCTAN(Y, 0.0) = MATH_PI_OVER_2 if Y > 0.0
-- ARCTAN(Y, 0.0) = -MATH_PI_OVER_2 if Y < 0.0
-- Domain:
-- Y in real, X in real, X /= 0.0 when Y = 0.0
-- Error conditions: Error if X = 0.0 and Y = 0.0
-- Range: -MATH_PI < ARCTAN(Y,X) <= MATH_PI
-- Notes: None
function SINH (X: in real) return real;
-- Returns hyperbolic sine of X
-- Special values: SINH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: SINH(X) is mathematically unbounded
-- Notes: The usable domain of SINH is approximately given
-- by: ABS(X) <= LOG(real'HIGH)
function COSH (X: in real) return real;
-- Returns hyperbolic cosine of X
-- Special values: COSH(0.0) = 1.0
-- Domain: X in real
-- Error conditions: None
-- Range: COSH(X) >= 1.0
-- Notes: The usable domain of COSH is approximately
-- given by: ABS(X) <= LOG(real'HIGH)
function TANH (X: in real) return real;
-- Returns hyperbolic tangent of X
-- Special values: TANH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ABS(TANH(X)) <= 1.0
-- Notes: None
function ARCSINH (X: in real) return real;
-- Returns inverse hyperbolic sine of X
-- Special values: ARCSINH(0.0) = 0.0
-- Domain: X in real
-- Error conditions: None
-- Range: ARCSINH(X) is mathematically unbounded
-- Notes: The reachable range of ARCSINH is approximately
-- given by: ABS(ARCSINH(X)) <= LOG(real'HIGH)
function ARCCOSH (X: in real) return real;
-- Returns inverse hyperbolic cosine of X
-- Special values: ARCCOSH(1.0) = 0.0
-- Domain: X >= 1.
-- Error conditions: Error if X < 1.0
-- Range: ARCCOSH(X) >= 0.0
-- Notes: The upper bound of the reachable range of
-- ARCCOSH is approximately given by:
-- ARCCOSH(X) <= LOG(real'HIGH)
function ARCTANH (X: in real) return real;
-- Returns inverse hyperbolic tangent of X
-- Special values: ARCTANH(0.0) = 0.0
-- Domain: ABS(X) < 1.0
-- Error conditions: Error if ABS(X) >= 1.0
-- Range: ARCTANH(X) is mathematically unbounded
-- Notes: The reachable range of ARCTANH is
-- approximately given by:
-- ABS(ARCTANH(X)) < LOG(real'HIGH)
end MATH_REAL;
```

16.3.2 IEEE.MATH_COMPLEX

```
Library IEEE;
Package MATH_COMPLEX is
    type COMPLEX        is record RE, IM: real; end record;
    type COMPLEX_VECTOR is array (integer range <>) of COMPLEX;
    type COMPLEX_POLAR  is record MAG: real; ARG: real; end record;
    constant  CBASE_1: complex := COMPLEX'(1.0, 0.0);
    constant  CBASE_j: complex := COMPLEX'(0.0, 1.0);
    constant  CZERO: complex := COMPLEX'(0.0, 0.0);
    function CABS(Z: in complex ) return real;
       -- returns absolute value (magnitude) of Z
    function CARG(Z: in complex ) return real;
       -- returns argument (angle) in radians of a complex number
    function CMPLX(X: in real;  Y: in real:= 0.0 ) return complex;
       -- returns complex number X + iY
    function "-" (Z: in complex ) return complex;
       -- unary minus
    function "-" (Z: in complex_polar ) return complex_polar;
       -- unary minus
    function CONJ (Z: in complex) return complex;
       -- returns complex conjugate
    function CONJ (Z: in complex_polar) return complex_polar;
       -- returns complex conjugate
    function CSQRT(Z: in complex ) return complex_vector;
       -- returns square root of Z; 2 values
    function CEXP(Z: in complex ) return complex;
       -- returns e**Z
    function COMPLEX_TO_POLAR(Z: in complex ) return complex_polar;
       -- converts complex to complex_polar
    function POLAR_TO_COMPLEX(Z: in complex_polar ) return complex;
       -- converts complex_polar to complex
    -- arithmetic operators
    function "+" ( L: in complex;  R: in complex ) return complex;
    function "+" ( L: in complex_polar; R: in complex_polar)
                  return complex;
    function "+" ( L: in complex_polar; R: in complex ) return complex;
    function "+" ( L: in complex;  R: in complex_polar) return complex;
    function "+" ( L: in real;     R: in complex ) return complex;
    function "+" ( L: in complex;  R: in real )    return complex;
    function "+" ( L: in real;  R: in complex_polar) return complex;
    function "+" ( L: in complex_polar;  R: in real) return complex;

    function "-" ( L: in complex;  R: in complex ) return complex;
    function "-" ( L: in complex_polar; R: in complex_polar)
                    return complex;
    function "-" ( L: in complex_polar; R: in complex ) return complex;
    function "-" ( L: in complex;  R: in complex_polar) return complex;
    function "-" ( L: in real;     R: in complex ) return complex;
    function "-" ( L: in complex;  R: in real )    return complex;
    function "-" ( L: in real;  R: in complex_polar) return complex;
    function "-" ( L: in complex_polar;  R: in real) return complex;

    function "*" ( L: in complex;  R: in complex ) return complex;
    function "*" ( L: in complex_polar; R: in complex_polar)
                  return complex;
    function "*" ( L: in complex_polar; R: in complex ) return complex;
    function "*" ( L: in complex;  R: in complex_polar) return complex;
```

```
    function "*" ( L: in real;     R: in complex ) return complex;
    function "*" ( L: in complex;  R: in real )    return complex;
    function "*" ( L: in real;  R: in complex_polar) return complex;
    function "*" ( L: in complex_polar;  R: in real) return complex;
    function "/" ( L: in complex;  R: in complex ) return complex;
    function "/" ( L: in complex_polar; R: in complex_polar)
                  return complex;
    function "/" ( L: in complex_polar; R: in complex ) return complex;
    function "/" ( L: in complex;  R: in complex_polar) return complex;
    function "/" ( L: in real;     R: in complex ) return complex;
    function "/" ( L: in complex;  R: in real )    return complex;
    function "/" ( L: in real;  R: in complex_polar) return complex;
    function "/" ( L: in complex_polar;  R: in real) return complex;
end  MATH_COMPLEX;
```

16.4 Les constantes pour l'analogique

16.4.1 [AMS]IEEE.FUNDAMENTAL_CONSTANTS

```
library IEEE; use IEEE.MATH_REAL.all;
package FUNDAMENTAL_CONSTANTS is
-- some declarations
  attribute SYMBOL : STRING;
  attribute UNIT   : STRING;
-- PHYSICAL CONSTANTS
   -- electron charge <COULOMB>
   -- NIST value: 1.602 176 462e-19 coulomb
   -- uncertainty: 0.000 000 063e-19
   constant PHYS_Q : REAL :=  1.602_176_462e-19;
   -- permittivity of vacuum <FARADS/METER>
   -- NIST value: 8.854 187 817e-12 farads/meter
   -- uncertainty: exact
   constant PHYS_EPS0 : REAL := 8.854_187_817e-12;
   -- permeability of vacuum <HENRIES/METER>
   -- NIST value: 4e-7*pi henries/meter
   -- uncertainty: exact
   constant PHYS_MU0 : REAL := 4.0e-7 * MATH_PI;
   -- Boltzmann's constant <JOULES/KELVIN>
   -- NIST value: 1.380 6503e-23 joules/kelvin
   -- uncertainty: 0.000 0024e-23:
   constant PHYS_K : REAL :=  1.380_6503e-23;
   -- Acceleration due to gravity <METERS/SECOND_SQUARED>
   -- NIST value: 9.806 65 meters/square second
   -- uncertainty: exact
   constant PHYS_GRAVITY : REAL := 9.806_65;
   -- Conversion between Kelvin and degree Celsius
   -- NIST value: 273.15
   -- uncertainty: exact
   constant PHYS_CTOK : REAL := 273.15;
   -- object declarations
   -- common scaling factors
   constant  YOCTO : REAL := 1.0e-24;
   constant  ZEPTO : REAL := 1.0e-21;
   constant  ATTO  : REAL := 1.0e-18;
   constant  FEMTO : REAL := 1.0e-15;
   constant  PICO  : REAL := 1.0e-12;
   constant  NANO  : REAL := 1.0e-9;
   constant  MICRO : REAL := 1.0e-6;
   constant  MILLI : REAL := 1.0e-3;
   constant  CENTI : REAL := 1.0e-2;
   constant  DECI  : REAL := 1.0e-1;
   constant  DEKA  : REAL := 1.0e+1;
   constant  HECTO : REAL := 1.0e+2;
   constant  KILO  : REAL := 1.0e+3;
   constant  MEGA  : REAL := 1.0e+6;
   constant  GIGA  : REAL := 1.0e+9;
   constant  TERA  : REAL := 1.0e+12;
   constant  PETA  : REAL := 1.0e+15;
   constant  EXA   : REAL := 1.0e+18;
   constant  ZETTA : REAL := 1.0e+21;
   constant  YOTTA : REAL := 1.0e+24;
   alias DECA is DEKA;
end package FUNDAMENTAL_CONSTANTS;
```

16.4.2 [AMS]IEEE.MATERIAL_CONSTANTS

```
library IEEE;
    use IEEE.MATH_REAL.all;

package MATERIAL_CONSTANTS is

   -- Relative permittivity of silicon
   -- source?
   -- uncertainty: ?
   -- constant EPS_SI : REAL := 11.7;
   constant PHYS_EPS_SI : REAL;

   -- Relative permittivity of silicon dioxide
   -- source?
   -- uncertainty: ?
   -- constant EPS_SIO2 : REAL := 3.9;
   constant PHYS_EPS_SIO2 : REAL;

   -- Young's Modulus for silicon <PASCALS>
   -- source?
   -- uncertainty: ?
   -- constant E_SI : REAL := 190.0e+9;
   constant PHYS_E_SI : REAL;

   -- Young's Modulus for silicon dioxide <PASCALS>
   -- source?
   -- uncertainty: ?
   -- constant E_SIO2 : REAL := 73.0e+9;
   constant PHYS_E_SIO2 : REAL;

   -- Poisson's Ratio for silicon <100orientation>
   -- source?
   -- uncertainty: ?
   -- constant NU_SI : REAL := 0.28;
   constant PHYS_NU_SI : REAL;

   -- Density of Polysilicon
   -- source?
   -- uncertainty: ?
   -- constant RHO_POLY : REAL := 2330.0;
   constant PHYS_RHO_POLY : REAL;

   -- Environmental constants
   constant AMBIENT_TEMPERATURE : REAL;
   constant AMBIENT_PRESSURE    : REAL;
   constant AMBIENT_LUMINANCE   : REAL;

end package MATERIAL_CONSTANTS;
```

16.5 La bibliothèque Disciplines

16.5.1 [AMS] DISCIPLINES.ENERGY_SYSTEMS

```
library IEEE;
   use IEEE.FUNDAMENTAL_CONSTANTS.all;

package ENERGY_SYSTEMS is

   -- type declarations

   -- subtype declarations
   subtype ENERGY      is REAL tolerance "DEFAULT_ENERGY";
   subtype POWER       is REAL tolerance "DEFAULT_POWER";
   subtype PERIODICITY is REAL tolerance "DEFAULT_PERIODICITY";

   -- attribute declarations
   attribute UNIT of ENERGY      : subtype is "Joule";
   attribute UNIT of POWER       : subtype is "Watt";
   attribute UNIT of PERIODICITY : subtype is "";
   attribute SYMBOL of ENERGY      : subtype is "J";
   attribute SYMBOL of POWER       : subtype is "W";
   attribute SYMBOL of PERIODICITY : subtype is "";

   -- nature declarations

   -- vector subtype declarations
   subtype ENERGY_VECTOR      is REAL_VECTOR tolerance "DEFAULT_ENERGY";
   subtype POWER_VECTOR       is REAL_VECTOR tolerance "DEFAULT_POWER";
   subtype PERIODICITY_VECTOR is REAL_VECTOR tolerance
"DEFAULT_PERIODICITY";

   -- attributes of vector subtypes
   attribute UNIT of ENERGY_VECTOR      : subtype is "Joule";
   attribute UNIT of POWER_VECTOR       : subtype is "Watt";
   attribute UNIT of PERIODICITY_VECTOR : subtype is "";
   attribute SYMBOL of ENERGY_VECTOR      : subtype is "J";
   attribute SYMBOL of POWER_VECTOR       : subtype is "W";
   attribute SYMBOL of PERIODICITY_VECTOR : subtype is "";

   -- subnature declarations

   -- object declarations

   -- operation declarations (e.g., subprograms)

   -- alias declarations
end package ENERGY_SYSTEMS;
```

16.5.2 [AMS] DISCIPLINES.ELECTRICAL_SYSTEMS

```
library IEEE;
    use IEEE.FUNDAMENTAL_CONSTANTS.all;
package ELECTRICAL_SYSTEMS is
   subtype VOLTAGE     is REAL tolerance "DEFAULT_VOLTAGE";
   subtype CURRENT     is REAL tolerance "DEFAULT_CURRENT";
   subtype CHARGE      is REAL tolerance "DEFAULT_CHARGE";
   subtype RESISTANCE  is REAL tolerance "DEFAULT_RESISTANCE";
   subtype CAPACITANCE is REAL tolerance "DEFAULT_CAPACITANCE";
   subtype MMF         is REAL tolerance "DEFAULT_MMF";
   subtype FLUX        is REAL tolerance "DEFAULT_FLUX";
   subtype INDUCTANCE  is REAL tolerance "DEFAULT_INDUCTANCE";
   -- attribute declarations
   -- Use of UNIT to designate units
   attribute UNIT of VOLTAGE       : subtype is "Volt";
   attribute UNIT of CURRENT       : subtype is "Ampere";
   attribute UNIT of CHARGE        : subtype is "Coulomb";
   attribute UNIT of RESISTANCE    : subtype is "Ohm";
   attribute UNIT of CAPACITANCE   : subtype is "Farad";

   attribute UNIT of MMF           : subtype is "Ampere-turns";
   attribute UNIT of FLUX          : subtype is "Weber";
   attribute UNIT of INDUCTANCE    : subtype is "Henry";
   -- Use of SYMBOL to designate abbreviation of units
   attribute SYMBOL of VOLTAGE     : subtype is "V";
   attribute SYMBOL of CURRENT     : subtype is "A";
   attribute SYMBOL of CHARGE      : subtype is "C";
   attribute SYMBOL of RESISTANCE  : subtype is "Ohm";
   attribute SYMBOL of CAPACITANCE : subtype is "F";
   attribute SYMBOL of MMF         : subtype is "A-t";
   attribute SYMBOL of FLUX        : subtype is "W";
   attribute SYMBOL of INDUCTANCE  : subtype is "H";
   -- nature declarations
   nature ELECTRICAL is VOLTAGE across
                        CURRENT  through
                        ELECTRICAL_REF reference;
   nature ELECTRICAL_VECTOR is array (NATURAL range <>) of ELECTRICAL;
   -- vector subtypes for array types
   nature MAGNETIC is MMF  across
                      FLUX  through
                      MAGNETIC_REF reference;
   nature MAGNETIC_VECTOR is array (NATURAL range <>) of MAGNETIC;
   -- vector subtype declarations
   subtype VOLTAGE_VECTOR    is ELECTRICAL_VECTOR'across;
   subtype CURRENT_VECTOR    is ELECTRICAL_VECTOR'through;
   subtype CHARGE_VECTOR     is REAL_VECTOR tolerance "DEFAULT_CHARGE";
   subtype RESISTANCE_VECTOR is REAL_VECTOR tolerance "DEFAULT_RESISTANCE";
   subtype MMF_VECTOR        is MAGNETIC_VECTOR'across;
   subtype FLUX_VECTOR       is MAGNETIC_VECTOR'through;
   subtype INDUCTANCE_VECTOR is REAL_VECTOR tolerance "DEFAULT_INDUCTANCE";
   -- attributes of vector subtypes
   -- Use of UNIT to designate units
   attribute UNIT of VOLTAGE_VECTOR       : subtype is "Volt";
   attribute UNIT of CURRENT_VECTOR       : subtype is "Ampere";
   attribute UNIT of CHARGE_VECTOR        : subtype is "Coulomb";
   attribute UNIT of RESISTANCE_VECTOR    : subtype is "Ohm";
   attribute UNIT of CAPACITANCE_VECTOR   : subtype is "Farad";
   attribute UNIT of MMF_VECTOR           : subtype is "Ampere-turns";
```

```
   attribute UNIT of FLUX_VECTOR          : subtype is "Weber";
   attribute UNIT of INDUCTANCE_VECTOR    : subtype is "Henry";
   -- Use of SYMBOL to designate abbreviation of units
   attribute SYMBOL of VOLTAGE_VECTOR     : subtype is "V";
   attribute SYMBOL of CURRENT_VECTOR     : subtype is "A";
   attribute SYMBOL of CHARGE_VECTOR      : subtype is "C";
   attribute SYMBOL of RESISTANCE_VECTOR  : subtype is "Ohm";
   attribute SYMBOL of CAPACITANCE_VECTOR : subtype is "F";
   attribute SYMBOL of MMF_VECTOR         : subtype is "A-t";
   attribute SYMBOL of FLUX_VECTOR        : subtype is "W";
   attribute SYMBOL of INDUCTANCE_VECTOR  : subtype is "H";
   alias GROUND is ELECTRICAL_REF;
end package ELECTRICAL_SYSTEMS;
```

16.5.3 [AMS] DISCIPLINES.MECHANICAL_SYSTEMS

```
library IEEE;
    use IEEE.FUNDAMENTAL_CONSTANTS.all;
package MECHANICAL_SYSTEMS is
   subtype DISPLACEMENT         is REAL tolerance "DEFAULT_DISPLACEMENT";
   subtype FORCE                is REAL tolerance "DEFAULT_FORCE";
   subtype VELOCITY             is REAL tolerance "DEFAULT_VELOCITY";
   subtype ACCELERATION         is REAL tolerance "DEFAULT_ACCELERATION";
   subtype MASS                 is REAL tolerance "DEFAULT_MASS";
   subtype STIFFNESS            is REAL tolerance "DEFAULT_STIFFNESS";
   subtype DAMPING              is REAL tolerance "DEFAULT_DAMPING";
   subtype MOMENTUM             is REAL tolerance "DEFAULT_MOMENTUM";
   subtype COMPLIANCE           is REAL tolerance "DEFAULT_COMPLIANCE";
   subtype ANGLE                is REAL tolerance "DEFAULT_ANGLE";
   subtype TORQUE               is REAL tolerance "DEFAULT_TORQUE";
   subtype ANGULAR_VELOCITY   is REAL tolerance "DEFAULT_ANGULAR_VELOCITY";
   subtype ANGULAR_ACCELERATION is REAL tolerance
                               "DEFAULT_ANGULAR_ACCELERATION";
   subtype MOMENT_INERTIA       is REAL tolerance "DEFAULT_MOMENT_INERTIA";
   subtype ANGULAR_MOMENTUM   is REAL tolerance "DEFAULT_ANGULAR_MOMENTUM";
   subtype ANGULAR_STIFFNESS is REAL tolerance "DEFAULT_ANGULAR_STIFFNESS";
   subtype ANGULAR_DAMPING    is REAL tolerance "DEFAULT_ANGULAR_DAMPING";
   -- attribute declarations
   -- Use of UNIT to designate units
   attribute UNIT of DISPLACEMENT         : subtype is "meter";
   attribute UNIT of FORCE                : subtype is "Newton";
   attribute UNIT of VELOCITY             : subtype is "meter/second";
   attribute UNIT of ACCELERATION         : subtype is "meter/second^2";
   attribute UNIT of MASS                 : subtype is "kilogram";
   attribute UNIT of STIFFNESS            : subtype is "Newton/meter";
   attribute UNIT of DAMPING          : subtype is "Newton-second/meter";
   attribute UNIT of MOMENTUM         : subtype is "kilogram-meter/second";
   attribute UNIT of COMPLIANCE           : subtype is ""; ???
   attribute UNIT of ANGLE                : subtype is "radian";
   attribute UNIT of TORQUE               : subtype is "Newton-meter";
   attribute UNIT of ANGULAR_VELOCITY     : subtype is "radian/second";
   attribute UNIT of ANGULAR_ACCELERATION : subtype is "radian/second^2";
   attribute UNIT of MOMENT_INERTIA       : subtype is "kilogram-meter^2";
   attribute UNIT of ANGULAR_MOMENTUM: subtype is "kilogram radian/second";
   attribute UNIT of ANGULAR_STIFFNESS    : subtype is "Newton radian"; ???
   attribute UNIT of ANGULAR_DAMPING  : subtype is "Newton second/radian";
   -- Use of UNIT to designate abbreviations of units
   attribute SYMBOL of DISPLACEMENT         : subtype is "m";
   attribute SYMBOL of FORCE                : subtype is "N";
   attribute SYMBOL of VELOCITY             : subtype is "m/s";
   attribute SYMBOL of ACCELERATION         : subtype is "m/s^2";
   attribute SYMBOL of MASS                 : subtype is "kg";
   attribute SYMBOL of STIFFNESS            : subtype is "N/m";
   attribute SYMBOL of DAMPING              : subtype is "N-s/m";
   attribute SYMBOL of MOMENTUM             : subtype is "kg-m/s";
   attribute SYMBOL of COMPLIANCE           : subtype is ""; ???
   attribute SYMBOL of ANGLE                : subtype is "rad";
   attribute SYMBOL of TORQUE               : subtype is "N-m";
   attribute SYMBOL of ANGULAR_VELOCITY     : subtype is "rad/s";
   attribute SYMBOL of ANGULAR_ACCELERATION : subtype is "rad/s^2";
   attribute SYMBOL of MOMENT_INERTIA       : subtype is "kg-m^2";
   attribute SYMBOL of ANGULAR_MOMENTUM     : subtype is "kg-rad/s"; ???
   attribute SYMBOL of ANGULAR_STIFFNESS    : subtype is "N-rad"; ???
   attribute SYMBOL of ANGULAR_DAMPING      : subtype is "N-s/rad"; ???
```

```
   -- nature declarations
   nature TRANSLATIONAL is
      DISPLACEMENT      across
      FORCE             through
      TRANSLATIONAL_REF reference;
   nature TRANSLATIONAL_VECTOR is
      array (NATURAL range <>) of TRANSLATIONAL;
   nature TRANSLATIONAL_VELOCITY is
      VELOCITY                  across
      FORCE                     through
      TRANSLATIONAL_VELOCITY_REF  reference;
   nature TRANSLATIONAL_VELOCITY_VECTOR is
      array (NATURAL range <>) of TRANSLATIONAL_VELOCITY;
   nature ROTATIONAL is
      ANGLE          across
      TORQUE         through
      ROTATIONAL_REF reference;
   nature ROTATIONAL_VECTOR is
      array (NATURAL range <>) of ROTATIONAL;
   nature ROTATIONAL_VELOCITY is
      ANGULAR_VELOCITY         across
      TORQUE                   through
      ROTATIONAL_VELOCITY_REF  reference;
   nature ROTATIONAL_VELOCITY_VECTOR is
      array (NATURAL range <>) of ROTATIONAL_VELOCITY;
   -- vector subtype declarations
   subtype DISPLACEMENT_VECTOR     is TRANSLATIONAL_VECTOR'across;
   subtype FORCE_VECTOR            is TRANSLATIONAL_VECTOR'through;
   subtype VELOCITY_VECTOR         is TRANSLATIONAL_VELOCITY_VECTOR'across;
   subtype FORCE_VELOCITY_VECTOR  is TRANSLATIONAL_VELOCITY_VECTOR'through;
   subtype ANGLE_VECTOR            is ROTATIONAL_VECTOR'across;
   subtype TORQUE_VECTOR           is ROTATIONAL_VECTOR'through;
   subtype ANGULAR_VELOCITY_VECTOR is ROTATIONAL_VELOCITY_VECTOR'across;
   subtype TORQUE_VELOCITY_VECTOR  is ROTATIONAL_VELOCITY_VECTOR'through;
   subtype ACCELERATION_VECTOR is REAL_VECTOR tolerance
"DEFAULT_ACCELERATION";
   subtype MASS_VECTOR           is REAL_VECTOR tolerance "DEFAULT_MASS";
   subtype STIFFNESS_VECTOR   is REAL_VECTOR tolerance "DEFAULT_STIFFNESS";
   subtype DAMPING_VECTOR   is REAL_VECTOR tolerance "DEFAULT_DAMPING";
   subtype MOMENTUM_VECTOR    is REAL_VECTOR tolerance "DEFAULT_MOMENTUM";
   subtype COMPLIANCE_VECTOR is REAL_VECTOR tolerance "DEFAULT_COMPLIANCE";
   subtype ANGULAR_ACCELERATION_VECTOR is REAL_VECTOR tolerance
"DEFAULT_ANGULAR_ACCEL";
   subtype MOMENT_INERTIA_VECTOR is REAL_VECTOR tolerance
"DEFAULT_MOMENT_INERTIA";
   subtype ANGULAR_MOMENTUM_VECTOR is REAL_VECTOR tolerance
"DEFAULT_ANGULAR_MOMENTUM";
   subtype ANGULAR_STIFFNESS_VECTOR is REAL_VECTOR tolerance
"DEFAULT_ANGULAR_STIFFNESS";
   subtype ANGULAR_DAMPING_VECTOR is REAL_VECTOR tolerance
"DEFAULT_ANGULAR_DAMPING";
   -- attributes of vector subtypes
   -- Use of UNIT to designate units
   attribute UNIT of DISPLACEMENT_VECTOR   : subtype is "meter";
   attribute UNIT of FORCE_VECTOR          : subtype is "Newton";
   attribute UNIT of VELOCITY_VECTOR       : subtype is "meter/second";
   attribute UNIT of FORCE_VELOCITY_VECTOR : subtype is "Newton";
   attribute UNIT of ACCELERATION_VECTOR   : subtype is "meter/second^2";
   attribute UNIT of MASS_VECTOR           : subtype is "kilogram";
   attribute UNIT of STIFFNESS_VECTOR      : subtype is "Newton/meter";
   attribute UNIT of DAMPING_VECTOR    : subtype is "Newton second/meter";
```

```
   attribute UNIT of MOMENTUM_VECTOR  : subtype is "kilogram meter/second";
   attribute UNIT of COMPLIANCE_VECTOR  : subtype is "";
   attribute UNIT of ANGLE_VECTOR     : subtype is "radian";
   attribute UNIT of TORQUE_VECTOR    : subtype is "Newton meter";
   attribute UNIT of ANGULAR_VELOCITY_VECTOR: subtype is "radian/second";
   attribute UNIT of TORQUE_VELOCITY_VECTOR: subtype is "Newton meter";
   attribute UNIT of ANGULAR_ACCELERATION_VECTOR : subtype is
                "radian/second^2";
   attribute UNIT of MOMENT_INERTIA_VECTOR : subtype is "kilogram meter^2";
   attribute UNIT of ANGULAR_MOMENTUM_VECTOR : subtype is
               "kilogram radian/second"; ???
   attribute UNIT of ANGULAR_STIFFNESS_VECTOR  : subtype is
               "Newton radian";
   attribute UNIT of ANGULAR_DAMPING_VECTOR: subtype is "Newton
second/radian";
   -- Use of SYMBOL to designate abbreviation of units
   attribute SYMBOL of DISPLACEMENT_VECTOR         : subtype is "m";
   attribute SYMBOL of FORCE_VECTOR                : subtype is "N";
   attribute SYMBOL of VELOCITY_VECTOR             : subtype is "m/s";
   attribute SYMBOL of FORCE_VELOCITY_VECTOR       : subtype is "N";
   attribute SYMBOL of ACCELERATION_VECTOR         : subtype is "m/s^2";
   attribute SYMBOL of MASS_VECTOR                 : subtype is "kg";
   attribute SYMBOL of STIFFNESS_VECTOR            : subtype is "N/m";
   attribute SYMBOL of DAMPING_VECTOR              : subtype is "N-s/m";
   attribute SYMBOL of MOMENTUM_VECTOR             : subtype is "kg-m/s";
   attribute SYMBOL of COMPLIANCE_VECTOR           : subtype is ""; ???
   attribute SYMBOL of ANGLE_VECTOR                : subtype is "rad";
   attribute SYMBOL of TORQUE_VECTOR               : subtype is "N-m";
   attribute SYMBOL of ANGULAR_VELOCITY_VECTOR     : subtype is "rad/s";
   attribute SYMBOL of TORQUE_VELOCITY_VECTOR      : subtype is "N-m";
   attribute SYMBOL of ANGULAR_ACCELERATION_VECTOR : subtype is "rad/s^2";
   attribute SYMBOL of MOMENT_INERTIA_VECTOR       : subtype is "kg-m^2";
   attribute SYMBOL of ANGULAR_MOMENTUM_VECTOR     : subtype is "kg-rad/s";
   attribute SYMBOL of ANGULAR_STIFFNESS_VECTOR    : subtype is "N-rad";
   attribute SYMBOL of ANGULAR_DAMPING_VECTOR      : subtype is "N-s/rad";
   -- alias declarations
   alias ANCHOR is TRANSLATIONAL_REF;
   alias TRANSLATIONAL_V is TRANSLATIONAL_VELOCITY;
   alias ROTATIONAL_V is ROTATIONAL_VELOCITY;
end package MECHANICAL_SYSTEMS;
```

16.5.4 [AMS] DISCIPLINES.THERMAL_SYSTEMS

```
library IEEE;use IEEE.FUNDAMENTAL_CONSTANTS.all;
package THERMAL_SYSTEMS is
   -- type declarations
   -- subtype declarations (no thermal resistance or capacitance?)
   subtype TEMPERATURE         is REAL tolerance "DEFAULT_TEMPERATURE";
   subtype HEAT_FLOW           is REAL tolerance "DEFAULT_HEAT_FLOW";
   subtype THERMAL_CAPACITANCE is REAL tolerance
"DEFAULT_THERMAL_CAPACITANCE";
   subtype THERMAL_RESISTANCE  is REAL tolerance
"DEFAULT_THERMAL_RESISTANCE";
   -- attribute declarations
   attribute UNIT of TEMPERATURE         : subtype is "Kelvin";
   attribute UNIT of HEAT_FLOW           : subtype is "Joule/second";
   attribute UNIT of THERMAL_CAPACITANCE : subtype is "Joule-Kelvin";
   attribute UNIT of THERMAL_RESISTANCE  : subtype is "Kelvin-
second/Joule";
   attribute SYMBOL of TEMPERATURE         : subtype is "K";
   attribute SYMBOL of HEAT_FLOW           : subtype is "J/s";
   attribute SYMBOL of THERMAL_CAPACITANCE : subtype is "J-K";
   attribute SYMBOL of THERMAL_RESISTANCE  : subtype is "K-s/J";

   -- nature declarations
   nature THERMAL is
      TEMPERATURE across
      HEAT_FLOW   through
      THERMAL_REF reference;
   nature THERMAL_VECTOR is array (NATURAL range <>) of THERMAL;

   -- vector subtype declarations
   subtype TEMPERATURE_VECTOR         is THERMAL_VECTOR'across;
   subtype HEAT_FLOW_VECTOR           is THERMAL_VECTOR'through;
   subtype THERMAL_CAPACITANCE_VECTOR is REAL_VECTOR       tolerance
"DEFAULT_THERMAL_CAPACITANCE";
   subtype THERMAL_RESISTANCE_VECTOR  is REAL_VECTOR       tolerance
"DEFAULT_THERMAL_RESISTANCE";

   -- attributes of vector subtypes
   attribute UNIT of TEMPERATURE_VECTOR       : subtype is "Kelvin";
   attribute UNIT of HEAT_FLOW_VECTOR        : subtype is "Joule/second";
   attribute UNIT of THERMAL_CAPACITANCE_VECTOR : subtype is "Joule-
Kelvin";
   attribute UNIT of THERMAL_RESISTANCE_VECTOR  : subtype is
                  "Kelvin-second/Joule";
   attribute SYMBOL of TEMPERATURE_VECTOR         : subtype is "K";
   attribute SYMBOL of HEAT_FLOW_VECTOR           : subtype is "J/s";
   attribute SYMBOL of THERMAL_CAPACITANCE_VECTOR : subtype is "J-K";
   attribute SYMBOL of THERMAL_RESISTANCE_VECTOR  : subtype is "K-s/J";
   -- subnature declarations
   -- object declarations (no quantities)
   -- operation declarations (e.g., subprograms)
   -- alias declarations
end package THERMAL_SYSTEMS;
```

16.5.5 [AMS] DISCIPLINES.FLUIDIC_SYSTEMS

```
library IEEE;
    use IEEE.FUNDAMENTAL_CONSTANTS.all;

package FLUIDIC_SYSTEMS is
   -- type declarations

   -- subtype declarations
   subtype PRESSURE     is REAL tolerance "DEFAULT_PRESSURE";
   subtype VFLOW_RATE   is REAL tolerance "DEFAULT_VFLOW_RATE";
   subtype VOLUME       is REAL tolerance "DEFAULT_VOLUME";
   subtype DENSITY      is REAL tolerance "DEFAULT_DENSITY";
   subtype VISCOSITY    is REAL tolerance "DEFAULT_VISCOSITY";
   subtype FRESISTANCE  is REAL tolerance "DEFAULT_FRESISTANCE";
   subtype FCAPACITANCE is REAL tolerance "DEFAULT_FCAPACITANCE";
   subtype INERTANCE    is REAL tolerance "DEFAULT_INERTANCE";

   -- attribute declarations
   attribute UNIT of PRESSURE     : subtype is "Pascal";
   attribute UNIT of VFLOW_RATE   : subtype is "meter^3/second";
   attribute UNIT of DENSITY      : subtype is "kilogram/meter^3";
   attribute UNIT of VISCOSITY    : subtype is "Newton-second/meter^2";
   attribute UNIT of VOLUME       : subtype is "meter^3";
   attribute UNIT of FRESISTANCE  : subtype is "Newton-second/meter^5";
   attribute UNIT of FCAPACITANCE : subtype is "meter^5/Newton";
   attribute UNIT of INERTANCE    : subtype is "Newton-second^2/meter^5";
   -- SYMBOL for abbreviation of UNIT
   attribute SYMBOL of PRESSURE     : subtype is "P";
   attribute SYMBOL of VFLOW_RATE   : subtype is "m^3/s";
   attribute SYMBOL of DENSITY      : subtype is "kg/m^3";
   attribute SYMBOL of VISCOSITY    : subtype is "N-s/m^2";
   attribute SYMBOL of VOLUME       : subtype is "m^3";
   attribute SYMBOL of FRESISTANCE  : subtype is "N-s/m^5";
   attribute SYMBOL of FCAPACITANCE : subtype is "m^5/N";
   attribute SYMBOL of INERTANCE    : subtype is "N-s^2/m^5";

   -- nature declarations
   nature FLUIDIC is
      PRESSURE    across
      VFLOW_RATE  through
      FLUIDIC_REF reference;
   nature FLUIDIC_VECTOR is array (NATURAL range <>) of FLUIDIC;

   -- vector subtype declarations
   subtype PRESSURE_VECTOR     is FLUIDIC_VECTOR'across;
   subtype VFLOW_RATE_VECTOR   is FLUIDIC_VECTOR'through;
   subtype VOLUME_VECTOR       is REAL_VECTOR  tolerance "DEFAULT_VOLUME";
   subtype DENSITY_VECTOR      is REAL_VECTOR  tolerance "DEFAULT_DENSITY";
   subtype VISCOSITY_VECTOR   is REAL_VECTOR tolerance "DEFAULT_VISCOSITY";
   subtype FRESISTANCE_VECTOR is REAL_VECTOR tolerance
                   "DEFAULT_FRESISTANCE";
   subtype FCAPACITANCE_VECTOR is REAL_VECTOR  tolerance
                   "DEFAULT_FCAPACITANCE";
   subtype INERTANCE_VECTOR    is REAL_VECTOR  tolerance
                   "DEFAULT_INERTANCE";
   -- attributes of vector subtypes
   attribute UNIT of PRESSURE_VECTOR     : subtype is "Pascal";
   attribute UNIT of VFLOW_RATE_VECTOR   : subtype is "meter^3/second";
```

```
   attribute UNIT of DENSITY_VECTOR      : subtype is "kilogram/meter^3";
   attribute UNIT of VISCOSITY_VECTOR    : subtype is
                        "Newton-second/meter^2";
   attribute UNIT of VOLUME_VECTOR       : subtype is "meter^3";
   attribute UNIT of FRESISTANCE_VECTOR  : subtype is
                        "Newton-second/meter^5";
   attribute UNIT of FCAPACITANCE_VECTOR : subtype is "meter^5/Newton";
   attribute UNIT of INERTANCE_VECTOR    : subtype is
                        "Newton-second^2/meter^5";
   -- SYMBOL for abbreviation of UNIT
   attribute SYMBOL of PRESSURE_VECTOR     : subtype is "P";
   attribute SYMBOL of VFLOW_RATE_VECTOR   : subtype is "m^3/s";
   attribute SYMBOL of DENSITY_VECTOR      : subtype is "kg/m^3";
   attribute SYMBOL of VISCOSITY_VECTOR    : subtype is "N-s/m^2";
   attribute SYMBOL of VOLUME_VECTOR       : subtype is "m^3";
   attribute SYMBOL of FRESISTANCE_VECTOR  : subtype is "N-s/m^5";
   attribute SYMBOL of FCAPACITANCE_VECTOR : subtype is "m^5/N";
   attribute SYMBOL of INERTANCE_VECTOR    : subtype is "N-s^2/m^5";
end package FLUIDIC_SYSTEMS;
```

16.5.6 [AMS] DISCIPLINES.RADIANT_SYSTEMS

```
library IEEE;
    use IEEE.FUNDAMENTAL_CONSTANTS.all;

package RADIANT_SYSTEMS is
   -- type declarations

   -- subtype declarations
   subtype ILLUMINANCE is REAL tolerance "DEFAULT_ILLUMINANCE";
   subtype OPTIC_FLUX  is REAL tolerance "DEFAULT_OPTIC_FLUX";

   -- attribute declarations
   attribute UNIT of ILLUMINANCE : subtype is "candela";
   attribute UNIT of OPTIC_FLUX  : subtype is "lumen";
   -- SYMBOL is abbreviation of UNIT
   attribute SYMBOL of ILLUMINANCE : subtype is "can";
   attribute SYMBOL of OPTIC_FLUX  : subtype is "lum";

   -- nature declarations
   nature RADIANT is
      ILLUMINANCE across
      OPTIC_FLUX  through
      RADIANT_REF reference;
   nature RADIANT_VECTOR is array (NATURAL range <>) of RADIANT;

   -- vector subtype declarations
   subtype ILLUMINANCE_VECTOR is RADIANT_VECTOR'across;
   subtype OPTIC_FLUX_VECTOR  is RADIANT_VECTOR'through;

   -- attribute declarations
   attribute UNIT of ILLUMINANCE_VECTOR : subtype is "candela";
   attribute UNIT of OPTIC_FLUX_VECTOR  : subtype is "lumen";
   -- SYMBOL is abbreviation of UNIT
   attribute SYMBOL of ILLUMINANCE_VECTOR : subtype is "can";
   attribute SYMBOL of OPTIC_FLUX_VECTOR  : subtype is "lum";

   -- subnature declarations

   -- object declarations

   -- operation declarations (e.g., subprograms)

   -- alias declarations
end package RADIANT_SYSTEMS;
```

17 Syntaxe BNF

Voir §1.2 page 9 pour la définition. Cette syntaxe contient à la fois les extensions AMS et la mise à jour 2001. Il est probable que les outils AMS existants n'implémentent pas les dernières mises à jour du standard « mère », par exemple les types protégés. Mais ils ont vocation à le faire à court terme.

- abstract_literal ::=
  decimal_literal |
  based_literal

- access_type_definition ::= **access** subtype_indication

- across_aspect ::= identifier_list [tolerance_aspect] [**:=** expression]

- actual_designator ::= expression | signal_name | variable_name |
  file_name | terminal_name | quantity_name | **open**

- actual_parameter_part ::= parameter_association_list

- actual_part ::= actual_designator | function_name **(** actual_designator**)** |
  type_mark **(** actual_designator **)**

- adding_operator ::= **+** | **-** | **&**

- aggregate ::= **(** element_association { **,** element_association } **)**

- alias_declaration ::= **alias** alias_designator [**:** subtypealias_indication
  ] **is** name [signature] **;**

- alias_designator ::= identifier | character_literal | operator_symbol

- alias_indication ::= subtype_indication | subnature_indication

- allocator ::= **new** subtype_indication | **new** qualified_expression

- architecture_body ::= **architecture** identifier **of** entity_name **is**
  architecture_declarative_part **begin** architecture_statement_part **end**
  [**architecture**] [identifier] **;**

- architecture_declarative_part ::= { block_declarative_item }

- architecture_statement ::= simultaneous_statement |
  concurrent_statement

- architecture_statement_part ::= { architecture_statement }

- array_nature_definition ::= unconstrained_nature_definition |
  constrained_nature_definition

- array_type_definition ::= unconstrained_array_definition | constrained_array_definition
- assertion ::= **assert** condition [**report** expression] [**severity** expression]
- assertion_statement ::= [label :] assertion ;
- association_element ::= [formal_part =>] actual_part
- association_list ::= association_element { , association_element }
- attribute_declaration ::= **attribute** identifier : type_mark ;
- attribute_designator ::= attribute_simple_name
- attribute_name ::= prefix [signature] ' attribute_designator [**(** expression { , expression } **)**]
- attribute_specification ::= **attribute** attribute_designator **of** entity_specification **is** expression ;
- base ::= integer
- base_specifier ::= B | O | X
- base_unit_declaration ::= identifier ;
- based_integer ::= extended_digit { [underline] extended_digit }
- based_literal ::= base **#** based_integer [. based_integer] **#** [exponent]
- basic_character ::= basic_graphic_character | format_effector
- basic_graphic_character ::= upper_case_letter | digit | special_character | space_character
- basic_identifier ::= letter { [underline] letter_or_digit }
- binding_indication ::= [**use** entity_aspect] [generic_map_aspect] [port_map_aspect]
- bit_string_literal ::= base_specifier " [bit_value] "
- bit_value ::= extended_digit { [underline] extended_digit }
- block_configuration ::= **for** block_specification { use_clause } { configuration_item } **end for** ;

- block_declarative_item ::= subprogram_declaration | subprogram_body | type_declaration | subtype_declaration | constant_declaration | signal_declaration | shared_variable_declaration | file_declaration | alias_declaration | component_declaration | attribute_declaration | attribute_specification | configuration_specification | disconnection_specification | step_limit_specification | use_clause | group_template_declaration | group_declaration | nature_declaration | subnature_declaration | quantity_declaration | terminal_declaration

- block_declarative_part ::= { block_declarative_item }

- block_header ::= [generic_clause [generic_map_aspect ;]] [port_clause [port_map_aspect ;]]

- block_specification ::= architecture_statement_label | block_statement_label | generate_statement_label [**(** index_specification **)**]

- block_statement ::= block_label : **block** [**(** guard_expression **)**] [**is**] block_header block_declarative_part **begin** block_statement_part **end block** [block_label] ;

- block_statement_part ::= { concurrent_statement architecture_statement }

- branch_quantity_declaration ::= **quantity** [across_aspect] [through_aspect] terminal_aspect ;

- break_element ::= [break_selector_clause] quantity_name **=>** expression

- break_list ::= break_element { , break_element }

- break_selector_clause ::= **for** quantity_name **use**

- break_statement ::= [label :] **break** [break_list] [**when** condition] ;

- case_statement ::= [case_label :] **case** expression **is** case_statement_alternative { case_statement_alternative } **end case** [case_label] ;

- case_statement_alternative ::= **when** choices **=>** sequence_of_statements

- character_literal ::= ' graphic_character '

- choice ::= simple_expression | discrete_range | element_simple_name | **others**

- choices ::= choice { **|** choice } [21]

- component_configuration ::= **for** component_specification [binding_indication ;] [block_configuration] **end for** ;

[21] Attention, la barre verticale fait partie de la syntaxe et n'est pas ici un méta-symbole de la BNF

- component_declaration ::= **component** identifier [**is**] [local_generic_clause] [local_port_clause] **end component** [component_simple_name] ;

- component_instantiation_statement ::= instantiation_label : instantiated_unit [generic_map_aspect] [port_map_aspect] ;

- component_specification ::= instantiation_list : component_name

- composite_nature_definition ::= array_nature_definition | record_nature_definition

- composite_type_definition ::= array_type_definition | record_type_definition

- concurrent_assertion_statement ::= [label :] [**postponed**] assertion ;

- concurrent_break_statement ::= [label :] **break** [break_list] [sensitivity_clause] [**when** condition] ;

- concurrent_procedure_call_statement ::= [label :] [**postponed**] procedure_call ;

- concurrent_signal_assignment_statement ::= [label :] [**postponed**] conditional_signal_assignment | [label :] [**postponed**] selected_signal_assignment

- concurrent_statement ::= block_statement | process_statement | concurrent_procedure_call_statement | concurrent_assertion_statement | concurrent_signal_assignment_statement | component_instantiation_statement | generate_statement | concurrent_break_statement

- condition ::= boolean_expression

- condition_clause ::= **until** condition

- conditional_signal_assignment ::= target **<=** options conditional_waveforms ;

- conditional_waveforms ::= { waveform **when** condition **else** } waveform [**when** condition]

- configuration_declaration ::= **configuration** identifier **of** entity_name **is** configuration_declarative_part block_configuration **end** [**configuration**] [identifier] ;

- configuration_declarative_item ::= use_clause | attribute_specification | group_declaration

- configuration_declarative_part ::= { configuration_declarative_item }

- configuration_item ::= block_configuration | component_configuration

- configuration_specification ::= **for** component_specification binding_indication ;

- constant_declaration ::= **constant** identifier_list : subtype_indication [:= expression] ;
- constrained_array_definition ::= **array** index_constraint **of** element_subtype_indication
- constrained_nature_definition ::= **array** index_constraint **of** subnature_indication
- constraint ::= range_constraint | index_constraint
- context_clause ::= { context_item }
- context_item ::= library_clause | use_clause
- decimal_literal ::= integer [. integer] [exponent]
- declaration ::= type_declaration | subtype_declaration | object_declaration | interface_declaration | alias_declaration | attribute_declaration | component_declaration | group_template_declaration | group_declaration | entity_declaration | configuration_declaration | subprogram_declaration | package_declaration | nature_declaration | subnature_declaration | quantity_declaration | terminal_declaration
- delay_mechanism ::= **transport** | [**reject** time_expression] **inertial**
- design_file ::= design_unit { design_unit }
- design_unit ::= context_clause library_unit
- designator ::= identifier | operator_symbol
- direction ::= **to** | **downto**
- disconnection_specification ::= **disconnect** guarded_signal_specification **after** time_expression ;
- discrete_range ::= discrete_subtype_indication | **range**
- element_association ::= [choices **=>**] expression
- element_declaration ::= identifier_list : element_subtype_definition ;
- element_subnature_definition ::= subnature_indication
- element_subtype_definition ::= subtype_indication
- entity_aspect ::= **entity** entity_name [**(** architecture_identifier**)**] | **configuration** configuration_name | **open**
- entity_class ::= **entity** | **architecture** | **configuration** | **procedure** | **function** | **package** | **type** | **subtype** | **constant** | **signal** | **variable** | **component** | **label** | **literal** | **units** | **group** | **file** | **nature** | **subnature** | **quantity** | **terminal**
- entity_class_entry ::= entity_class [**<>**]

- entity_class_entry_list ::= entity_class_entry { , entity_class_entry }
- entity_declaration ::= **entity** identifier **is** entity_header entity_declarative_part [**begin** entity_statement_part] **end** [**entity**] [entity_simple_name] ;
- entity_declarative_item ::= subprogram_declaration | subprogram_body | type_declaration | subtype_declaration | constant_declaration | signal_declaration | shared_variable_declaration | file_declaration | alias_declaration | attribute_declaration | attribute_specification | disconnection_specification | step_limit_specification | use_clause | group_template_declaration | group_declaration | nature_declaration | subnature_declaration | quantity_declaration | terminal_declaration
- entity_declarative_part ::= { entity_declarative_item }
- entity_designator ::= entity_tag [signature]
- entity_header ::= [formal_generic_clause] [formal_port_clause]
- entity_name_list ::= entity_designator { , entity_designator } | **others** | **all**
- entity_specification ::= entity_name_list : entity_class
- entity_statement ::= concurrent_assertion_statement | passive_concurrent_procedure_call_statement | passive_process_statement
- entity_statement_part ::= { entity_statement }
- entity_tag ::= simple_name | character_literal | operator_symbol
- enumeration_literal ::= identifier | character_literal
- enumeration_type_definition ::= **(** enumeration_literal { , enumeration_literal } **)**
- exit_statement ::= [label **:**] **exit** [loop_label] [**when** condition] ;
- exponent ::= **E** [**+**] integer | **E -** integer
- expression ::= relation { **and** relation } | relation { **or** relation } | relation { **xor** relation } | relation [**nand** relation] | relation [**nor** relation] | relation { **xnor** relation }
- extended_digit ::= digit | letter
- extended_identifier ::= **** graphic_character { graphic_character } ****
- factor ::= primary [****** primary] | **abs** primary | **not** primary
- file_declaration ::= **file** identifier_list **:** subtype_indication [file_open_information] ;
- file_logical_name ::= string_expression

- file_open_information ::= [**open** file_open_kind_expression] **is** file_logical_name
- file_type_definition ::= **file of** type_mark
- floating_type_definition ::= range_constraint
- formal_designator ::= generic_name | port_name | parameter_name
- formal_parameter_list ::= parameter_interface_list
- formal_part ::= formal_designator | function_name **(** formal_designator **)** | type_mark **(** formal_designator **)**
- free_quantity_declaration ::= **quantity** identifier_list : subtype_indication [:= expression] ;
- full_type_declaration ::= **type** identifier **is** type_definition ;
- function_call ::= function_name [**(** actual_parameter_part **)**]
- generate_statement ::= generate_label : generation_scheme **generate** [{ block_declarative_item } **begin**] { concurrent_statement } { architecture_statement } **end generate** [generate_label] ;
- generation_scheme ::= **for** generate_parameter_specification | **if** condition
- generic_clause ::= **generic (** generic_list **)** ;
- generic_list ::= generic_interface_list
- generic_map_aspect ::= **generic map (** generic_association_list **)**
- graphic_character ::= basic_graphic_character | lower_case_letter | other_special_character
- group_constituent ::= name | character_literal
- group_constituent_list ::= group_constituent { , group_constituent }
- group_declaration ::= **group** identifier : group_template_name **(** group_constituent_list **)** ;
- group_template_declaration ::= **group** identifier **is (** entity_class_entry_list **)** ;
- guarded_signal_specification ::= guarded_signal_list : type_mark
- identifier ::= basic_identifier | extended_identifier
- identifier_list ::= identifier { , identifier }
- if_statement ::= [if_label :] **if** condition **then** sequence_of_statements { **elsif** condition **then** sequence_of_statements } [**else** sequence_of_statements] **end if** [if_label] ;

- incomplete_type_declaration ::= **type** identifier ;
- index_constraint ::= **(** discrete_range { , discrete_range } **)**
- index_specification ::= discrete_range | static_expression
- index_subtype_definition ::= type_mark **range <>**
- indexed_name ::= prefix **(** expression { , expression } **)**
- instantiated_unit ::= [**component**] component_name | **entity** entity_name [**(** architecture_identifier **)**] | **configuration** configuration_name
- instantiation_list ::= instantiation_label { , instantiation_label } | **others** | **all**
- integer ::= digit { [underline] digit }
- integer_type_definition ::= range_constraint
- interface_constant_declaration ::= [**constant**] identifier_list **:** [**in**] subtype_indication [**:=** static_expression]
- interface_declaration ::= interface_constant_declaration | interface_signal_declaration | interface_variable_declaration | interface_file_declaration | interface_terminal_declaration | interface_quantity_declaration
- interface_element ::= interface_declaration
- interface_file_declaration ::= **file** identifier_list **:** subtype_indication
- interface_list ::= interface_element { ; interface_element }
- interface_quantity_declaration ::= **quantity** identifier_list **:** [**in** | **out**] subtype_indication [**:=** static_expression]
- interface_signal_declaration ::= [**signal**] identifier_list **:** [mode] subtype_indication [**bus**] [**:=** static_expression]
- interface_terminal_declaration ::= **terminal** identifier_list **:** subnature_indication
- interface_variable_declaration ::= [**variable**] identifier_list **:** [mode] subtype_indication [**:=** static_expression]
- iteration_scheme ::= **while** condition | **for** loop_parameter_specification
- label ::= identifier
- letter ::= upper_case_letter | lower_case_letter
- letter_or_digit ::= letter | digit
- library_clause ::= **library** logical_name_list ;

- library_unit ::= primary_unit | secondary_unit
- literal ::= numeric_literal | enumeration_literal | string_literal | bit_string_literal | **null**
- logical_name ::= identifier
- logical_name_list ::= logical_name { , logical_name }
- logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** | **xnor**
- loop_statement ::= [loop_label :] [iteration_scheme] **loop** sequence_of_statements **end loop** [loop_label] ;
- miscellaneous_operator ::= ****** | **abs** | **not**
- mode ::= **in** | **out**| **inout** | **buffer** | **linkage**
- multiplying_operator ::= ***** | **/** | **mod** | **rem**
- name ::= simple_name | operator_symbol | selected_name | indexed_name | slice_name | attribute_name
- nature_declaration ::= **nature** identifier **is** nature_definition ;
- nature_definition ::= scalar_nature_definition | composite_nature_definition
- nature_element_declaration ::= identifier_list : element_subnature_definition
- nature_mark ::= nature_name | subnature_name
- next_statement ::= [label :] **next** [loop_label] [**when** condition] ;
- null_statement ::= [label :] **null** ;
- numeric_literal ::= abstract_literal | physical_literal
- object_declaration ::= constant_declaration | signal_declaration | variable_declaration | file_declaration | terminal_declaration | quantity_declaration
- operator_symbol ::= string_literal
- options ::= [**guarded**] [delay_mechanism]
- package_body ::= **package body** package_simple_name **is** package_body_declarative_part **end** [**package body**] [package_simple_name] ;
- package_body_declarative_item ::= subprogram_declaration | subprogram_body | type_declaration | subtype_declaration | constant_declaration | shared_variable_declaration | file_declaration | alias_declaration | use_clause | group_template_declaration | group_declaration

- package_body_declarative_part ::= { package_body_declarative_item }

- package_declaration ::= **package** identifier **is** package_declarative_part
  end [**package**] [identifier] ;

- package_declarative_item ::= subprogram_declaration | type_declaration |
  subtype_declaration | constant_declaration | signal_declaration |
  shared_variable_declaration | file_declaration | alias_declaration |
  component_declaration | attribute_declaration |
  attribute_specification | disconnection_specification | use_clause |
  group_template_declaration | group_declaration | nature_declaration
  | subnature_declaration | terminal_declaration

- package_declarative_part ::= { package_declarative_item }

- parameter_specification ::= identifier **in** discrete_range

- physical_literal ::= [abstract_literal] unit_name

- physical_type_definition ::= range_constraint **units**
  base_unit_declaration { secondary_unit_declaration } **end units** [
  physical_type_simple_name]

- port_clause ::= **port (** port_list **)** ;

- port_list ::= port_interface_list

- port_map_aspect ::= **port map (** port_association_list **)**

- prefix ::= name | function_call

- primary ::= name | literal | aggregate | function_call |
  qualified_expression | type_conversion | allocator | **(** expression **)**

- primary_unit ::= entity_declaration | configuration_declaration |
  package_declaration

- procedural_declarative_item ::= subprogram_declaration |
  subprogram_body | type_declaration | subtype_declaration |
  constant_declaration | variable_declaration | alias_declaration |
  attribute_declaration | attribute_specification | use_clause |
  group_template_declaration | group_declaration

- procedural_declarative_part ::= { procedural_declarative_item }

- procedural_statement_part ::= { sequential_statement }

- procedure_call ::= procedure_name [**(** actual_parameter_part **)**]

- procedure_call_statement ::= [label **:**] procedure_call ;

- process_declarative_item ::= subprogram_declaration | subprogram_body |
  type_declaration | subtype_declaration | constant_declaration |
  variable_declaration | file_declaration | alias_declaration |
  attribute_declaration | attribute_specification | use_clause |
  group_template_declaration | group_declaration

- process_declarative_part ::= { process_declarative_item }
- process_statement ::= [process_label :] [**postponed**] **process** [**(** sensitivity_list **)**] [**is**] process_declarative_part **begin** process_statement_part **end** [**postponed**] **process** [process_label] ;
- process_statement_part ::= { sequential_statement }
- protected_type_body ::= **protected body** protected_type_body_declarative_part **end protected body** [protected_type_simple_name]
- protected_type_body_declarative_item ::= subprogram_declaration | subprogram_body | type_declaration | subtype_declaration | constant_declaration | variable_declaration | file_declaration | alias_declaration | attribute_declaration | attribute_specification | use_clause | group_template_declaration | group_declaration
- protected_type_body_declarative_part ::= { protected_type_body_declarative_item }
- protected_type_declaration ::= **protected** protected_type_declarative_part **end protected** [protected_type_simple_name]
- protected_type_declarative_item ::= subprogram_specification | attribute_specification | use_clause protected_type_declarative_part ::= { protected_type_declarative_item }
- protected_type_definition ::= protected_type_declaration | protected_type_body
- qualified_expression ::= type_mark ' **(** expression **)** | type_mark ' aggregate
- quantity_declaration ::= free_quantity_declaration | branch_quantity_declaration | source_quantity_declaration
- quantity_list ::= quantity_name { **,** quantity_name } | **others** | **all**
- quantity_specification ::= quantity_list : type_mark
- range ::= range_attribute_name | simple_expression direction simple_expression
- range_constraint ::= **range** range
- record_nature_definition ::= **record** nature_element_declaration { nature_element_declaration } **end record** [record_nature_simple_name]
- record_type_definition ::= **record** element_declaration { element_declaration } **end record** [record_type_simple_name]
- relation ::= shift_expression [relational_operator shift_expression]
- relational_operator ::= **=** | **/=** | **<** | **<=** | **>** | **>=**

- report_statement ::= [label :] **report** expression [**severity** expression] ;
- return_statement ::= [label :] **return** [expression] ;
- scalar_nature_definition ::= type_mark **across** type_mark **through** identifier **reference**
- scalar_type_definition ::= enumeration_type_definition | integer_type_definition | floating_type_definition | physical_type_definition
- secondary_unit ::= architecture_body | package_body
- secondary_unit_declaration ::= identifier **=** physical_literal ;
- selected_name ::= prefix . suffix
- selected_signal_assignment ::= **with** expression **select** target **<=** options selected_waveforms ;
- selected_waveforms ::= { waveform **when** choices **,** } waveform **when** choices
- sensitivity_clause ::= **on** sensitivity_list
- sensitivity_list ::= signal_name { **,** signal_name }
- sequence_of_statements ::= { sequential_statement }
- sequential_statement ::= wait_statement | assertion_statement | report_statement | signal_assignment_statement | variable_assignment_statement | procedure_call_statement | if_statement | case_statement | loop_statement | next_statement | exit_statement | return_statement | null_statement | break_statement
- shift_expression ::= simple_expression [shift_operator simple_expression]
- shift_operator ::= **sll** | **srl** | **sla** | **sra** | **rol** | **ror**
- sign ::= **+** | **-**
- signal_assignment_statement ::= [label :] target **<=** [delay_mechanism] waveform ;
- signal_declaration ::= **signal** identifier_list **:** subtype_indication [signal_kind] [**:=** expression] ;
- signal_kind ::= **register** | **bus**
- signal_list ::= signal_name { **,** signal_name } | **others** | **all**
- signature ::= **[** [type_mark { **,** type_mark }] [**return** type_mark] **]** [22]

[22] Attention, les crochets englobants font partie de la syntaxe et ne sont pas ici des méta-symboles de la BNF.

- simple_expression ::= [sign] term { adding_operator term }
- simple_name ::= identifier
- simple_simultaneous_statement ::= [label :] simple_expression **==** simple_expression [tolerance_aspect] ;
- simultaneous_alternative ::= **when** choices **=>** simultaneous_statement_part
- simultaneous_case_statement ::= [case_label :] **case** expression **use** simultaneous_alternative { simultaneous_alternative } **end case** [case_label] ;
- simultaneous_if_statement ::= [if_label :] **if** condition **use** simultaneous_statement_part { **elsif** condition **use** simultaneous_statement_part } [**else** simultaneous_statement_part] **end use** [if_label] ;
- simultaneous_null_statement ::= [label :] **null** ;
- simultaneous_procedural_statement ::= [procedural_label :] **procedural** [**is**] procedural_declarative_part **begin** procedural_statement_part **end procedural** [procedural_label] ;
- simultaneous_statement ::= simple_simultaneous_statement | simultaneous_if_statement | simultaneous_case_statement | simultaneous_procedural_statement | simultaneous_null_statement
- simultaneous_statement_part ::= { simultaneous_statement }
- slice_name ::= prefix **(** discrete_range **)**
- source_aspect ::= **spectrum** magnitude_simple_expression , phase_simple_expression | **noise** power_simple_expression
- source_quantity_declaration ::= **quantity** identifier_list : subtype_indication source_aspect ;
- step_limit_specification ::= **limit** quantity_specification **with** real_expression ;
- string_literal ::= " { graphic_character } "
- subnature_declaration ::= **subnature** identifier **is** subnature_indication ;
- subnature_indication ::= nature_mark [index_constraint] [**tolerance** string_expression **across** string_expression **through**]
- subprogram_body ::= subprogram_specification **is** subprogram_declarative_part **begin** subprogram_statement_part **end** [subprogram_kind] [designator] ;

- subprogram_declaration ::= subprogram_specification ;
- subprogram_declarative_item ::= subprogram_declaration | subprogram_body | type_declaration | subtype_declaration | constant_declaration | variable_declaration | file_declaration | alias_declaration | attribute_declaration | attribute_specification | use_clause | group_template_declaration | group_declaration
- subprogram_declarative_part ::= { subprogram_declarative_item }
- subprogram_kind ::= **procedure** | **function**
- subprogram_specification ::= **procedure** designator [**(** formal_parameter_list **)**] | [**pure** | **impure**] **function** designator [**(** formal_parameter_list **)**] **return** type_mark
- subprogram_statement_part ::= { sequential_statement }
- subtype_declaration ::= **subtype** identifier **is** subtype_indication ;
- subtype_indication ::=[resolution_function_name] type_mark [constraint] [tolerance_aspect]
- suffix ::= simple_name | character_literal | operator_symbol | **all**
- target ::= name | aggregate
- term ::= factor { multiplying_operator factor }
- terminal_aspect ::= plus_terminal_name [**to** minus_terminal_name]
- terminal_declaration ::= **terminal** identifier_list : subnature_indication ;
- through_aspect ::= identifier_list [tolerance_aspect] [**:=** expression] **through**
- timeout_clause ::= **for** time_expression time_or_real_expression
- tolerance_aspect ::= **tolerance** string_expression
- type_conversion ::= type_mark **(** expression **)**
- type_declaration ::= full_type_declaration | incomplete_type_declaration
- type_definition ::= scalar_type_definition | composite_type_definition | access_type_definition | file_type_definition | protected_type_definition
- type_mark ::= type_name | subtype_name
- unconstrained_array_definition ::= **array (** index_subtype_definition { , index_subtype_definition } **) of** element_subtype_indication
- unconstrained_nature_definition ::= **array (** index_subtype_definition { , index_subtype_definition } **) of** subnature_indication
- use_clause ::= **use** selected_name { , selected_name } ;

- variable_assignment_statement ::= [label :] target **:=** expression ;

- variable_declaration ::= [**shared**] **variable** identifier_list :
  subtype_indication [**:=** expression] ;

- wait_statement ::= [label :] **wait** [sensitivity_clause] [
  condition_clause] [timeout_clause] ;

- waveform ::= waveform_element { , waveform_element } | **unaffected**

- waveform_element ::= value_expression [**after** time_expression] | **null**
  [**after** time_expression]

18 Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

19 Bibliographie

- [AIR] VHDL: Langage, Modélisation, Synthèse
  Roland Airiau, Jean-Michel Bergé, Vincent Olive, Jacques Rouillard,
  Presses Polytechniques Universitaires Romandes
  ISBN 2-88074-361-3

- [ASH] The VHDL Cookbook (anglais), une bonne et brève introduction à VHDL'87
  Peter Ashenden
  http://tams-www.informatik.uni-hamburg.de/vhdl/ ……
  …… doc/cookbook/VHDL-Cookbook.pdf

- [BER] VHDL Designer's Reference (anglais)
  Jean-Michel Bergé, Alain Fonkoua, Serge Maginot, Jacques Rouillard
  Kluwer Academic Publishers
  ISBN 0-7923-1756-4

- [CHR] Analog & Mixed-Signal extensions to VHDL (anglais)
  Ernst Christen, Kenneth Bakalar, in CIEM 5 Current Issues in Electronic Modeling
  "Analog & Mixed-Signal Hardware Description Languages",
  edited by *Alain Vachoux, Jean-Michel Bergé, Oz Levia, Jacques Rouillard*
  Kluwer Academic Publishers
  ISBN 0-7923-9875-0

- [HAM] Les archives VHDL de l'université de Hambourg
  http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/

- [HER] VHDL-AMS Applications et enjeux industriels
  Yannick Hervé, Dunod
  ISBN 9-782100-0058884

- [IEEE] Publications IEEE : Standards (anglais)
  - [IE1] 1076-2002 IEEE Standard **VHDL** Language Reference Manual
    Publication IEEE
    E-ISBN: 0-7381-3248-9
    ISBN: 0-7381-3247-0

  - [IE2] 1076.1-2007 IEEE Standard **VHDL Analog** and Mixed-Signal Extensions
    Publication IEEE
    E-ISBN: 0-7381-5628-0
    ISBN: 0-7381-5627-2

- [ROU1] Écrire & Comprendre VHDL
  Jacques Rouillard, Lulu.com
  ISBN 978-1-4092-3689-4
  Version pdf accessible : http://www.rouillard.org/ecrire-vhdl-et-ams.pdf

- [ROU2] Aide Mémoire VHDL, une version pdf de celui qui est à la fin de [ROU1].
  Jacques Rouillard
  http://www.rouillard.org/memovhdl.pdf

- [TUT] L'excellent tutorial sur VHDL-AMS sur le site de référence [HAM] (anglais):
  Ernst Christen, Kenneth Bakalar, Allen M. Dewey, Eduard Moser
  http://tams-www.informatik.uni-hamburg.de/vhdl/doc/P1076.1/tutdac99.pdf

- [VAC] VHDL(-AMS) Essentiel
  Alain Vachoux
  http://lsmwww.epfl.ch/Education/former/ ….
  ….. 2002-2003/modelmix03/Documents/VHDLAMS_instant.pdf

20 Table des figures