# Cours de Programmation en Langage Synchrone SIGNAL

Bernard HOUSSAIS
IRISA. Équipe ESPRESSO

24 septembre 2004

# Table des matières

# Chapitre 1

# Introduction

**Signal** est un langage pour la programmation d'applications *Temps Réel*, suivant une approche *synchrone*.

## 1.1 La Programmation Temps Réel

Les applications **Temps Réel** se distinguent des autres applications par la présence d'interactions entre le programme et son environnement, qui impose sa propre échelle de temps. Un système Temps Réel reçoit ses informations à certains instants, et agit lui-même sur cet environnement à des instants précis. L'ordre d'apparition des données, leur entrelacement, comptent autant que les valeurs de ces données.

N'importe quelle machine avec plusieurs capteurs ou boutons de commande est un système Temps Réel. Elle se comporte comme un automate, qui réagit, ou change d'état, au gré des diverses actions qu'elle reçoit.

Certaines autres applications, dites également *Temps Réel*, prennent en compte le temps d'exécution de tâches, en imposant par exemple des contraintes temporelles sur leur terminaison. C'est pourquoi les systèmes considérés ici sont aussi appelés *systèmes réactifs*, ou *enfouis*, ou *embarqués*...

Ces applications se trouvent dans des contextes de plus en plus critiques (avions, centrales nucléaires, armes,...), nécessitant un degré de fiabilité maximum. Or, le test de programmes Temps Réel est particulièrement difficile, ainsi que leur débogage (reconstitution de la séquence d'événements ayant conduit à un crash).

Les systèmes Temps Réel ont longtemps été programmée soit au moyen de primitives de bas niveau, soit avec des langages de programmation du parallélisme : CSP, Ada,... Cette approche, dite *asynchrone*, est non-déterministe, et se prête mal au développement de preuves de propriétés.

L'approche *synchrone* vise à écrire des programmes déterministes. Spécifications et programmes devront faciliter la vérification automatique de certaines propriétés du système, telle que l'absence de blocage. *SIGNAL*([1],[2],[3])est un des langages synchrones disponibles, en même temps que LUSTRE[4] et ESTEREL[5].

## 1.2 Traitement complet d'un exemple : le VEILLEUR

Afin de donner une idée intuitive de ce qu'est la programmation dans ce langage SIGNAL, nous proposons de traiter et de commenter une application simple, mais assez complète : le *programme* VEILLEUR.

### 1.2.1 Le problème

Un processus DEMANDEUR envoie des commandes à un EXECUTANT. Une commande doit être exécutée dans un certain DELAI, sinon il y a anomalie. On appelle VEILLEUR la *boite* chargée de vérifier ce délai. Elle reçoit copie de chaque commande émise, ainsi qu'un signal de l'exécutant lorsque la commande précédente est terminée. Elle déclenche une alarme si un délai trop long (fixé en paramètre) s'écoule entre ces deux signaux.

Si une nouvelle commande arrive alors que la précédente n'est pas achevée, le comptage du temps recommence à zéro. Un signal de terminaison arrivant après le délai, ou ne correspondant à aucune commande, est à négliger.

### 1.2.2 Les signaux d'entrée et de sortie.

Le processus VEILLEUR reçoit à un instant quelconque l'avis du lancement d'une commande. On supposera que cette commande est codée par un entier. Au cours de sa vie, le VEILLEUR recevra donc sur une *entrée* appelée *COMMANDE* une *suite d'entiers*, chacun à un instant précis.

Il reçoit également sur une autre entrée des avis de terminaison. Il s'agit d'une simple information élémentaire, appelée en SIGNAL **événement**, et codée par un "booléen" toujours vrai (type *event*).

Pour mesurer le temps à partir du lancement d'une commande, VEILLEUR utilise un *chronomètre* extérieur, qui lui envoie une impulsion (un TOP, de type événement) à chaque unité de temps, par exemple chaque seconde. Remarquer que l'on n'utilise pas le mot *horloge*, qui a un autre sens en SIGNAL. Le DELAI sera exprimé en nombre de tops de ce chronomètre, ce qui évite d'avoir à se préoccuper de l'unité de temps utilisée.

En sortie, VEILLEUR produit une ALARME chaque fois que le DELAI est dépassé après lancement d'une COMMANDE. L'alarme peut être un *event*, ou mieux, l'HEURE à laquelle se produit l'alarme, heure comptée par numérotation des tops d'entrée.

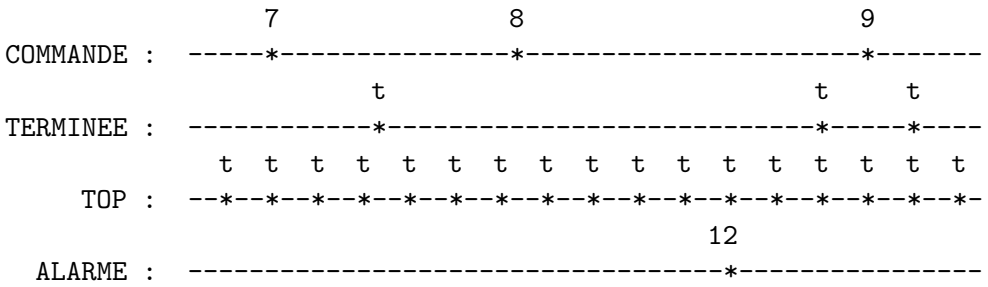
En résumé, les objets utilisés sont donc des **suites de valeurs** de divers types, chaque valeur de la suite étant présente à un **instant** donné. Un tel objet est appelé **signal**. La suite des instants où un signal prend une valeur est appelée **l'horloge** du signal.

Ce processus consomme 3 signaux d'entrée : COMMANDE, TERMINEE, et le TOP du chronomètre externe. Il produit un signal de sortie : ALARME.

### 1.2.3 Déroulement du processus VEILLEUR.

Pour figurer un scénario possible pour le processus VEILLEUR, on représente le temps sur un axe horizontal, et l'on note pour chaque signal les instants où ce signal est présent, en y associant sa valeur. Les signaux se produisant au même instant sont représentés sur la même verticale.

On suppose ici DELAI = 5 :




L'alarme s'est bien produite sur le *cinquième* TOP suivant la commande 8; c'était aussi le *douzième* TOP depuis le début.

Une autre représentation - plus précise - d'un scénario pourra être de ne représenter que ces instants où au moins un signal est présent, un signal absent étant noté par le symbole $\perp$ *(bottom)*.

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COMMANDE : | $\perp$ | 7 | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | 8 | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | 9 | $\perp$ | $\perp$ |
| TERMINEE : | $\perp$ | $\perp$ | $\perp$ | $\perp$ | *t* | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | *t* | $\perp$ | *t* | $\perp$ |
| TOP : | *t* | *t* | *t* | *t* | $\perp$ | *t* | *t* | *t* | $\perp$ | *t* | *t* | *t* | *t* | *t* | *t* | *t* | *t* | *t* | *t* |
| ALARME : | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ | 12 | $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |

Avec cette représentation, on voit - plus clairement que sur le diagramme précédent - que la commande **7** est arrivée exactement sur un TOP du chronomètre, alors que la première terminaison est entre les tops 4 et 5.

### 1.2.4 Hypothèses de synchronisme.

Un programme SIGNAL ne prend pas en compte l'intervalle de temps absolu entre deux valeurs d'un signal, mais seulement l'entrelacement - ou l'ordre relatif - ou la simultanéité des signaux.

Une hypothèse importante en SIGNAL est que l'environnement doit être capable de préciser si **deux valeurs arrivent au même instant**, ou à des instants différents. A un instant donné, on peut avoir un signal présent, ou plusieurs, ou aucun. On ne s'intéressera évidemment qu'aux instants où au moins un signal est présent.

Une autre hypothèse fondamentale est que **les calculs prennent un temps nul**. Ainsi, le signal ALARME est produit lors d'une certaine occurrence du signal TOP (la cinquième suivant le signal COMMANDE, dans l'exemple), et à condition que le signal TERMINEE ne soit pas arrivé avant, ou en même temps, que cette occurrence de TOP. La décision de produire ALARME **dépend** donc de TOP et de TERMINEE, et suit forcément dans le temps l'arrivée de ces signaux. Mais on suppose que la décision peut être prise en un temps nul (ou négligeable), et le signal ALARME sera donc produit **au même instant logique** que le TOP de fin de délai.

Le déroulement d'un programme Signal est donc une *succession d'instants* bien différenciés, séparés par une durée quelconque. Lors de chacun de ces instants, des valeurs disponibles sont utilisées, et peuvent servir à produire de nouvelles valeurs.

### 1.2.5 Le processus VEILLEUR en langage SIGNAL.

Les lignes numérotées sont celles du programme SIGNAL.

```
1 :  process VEILLEUR =
2 :  % déclenche une ALARME si une COMMANDE n'est pas TERMINEE dans un DELAI %
3 :    { integer DELAI}
4 :    ( ? integer COMMANDE;
5 :        event TERMINEE, TOP
6 :      ! integer ALARME )
```

Ces lignes constituent l'en-tête *(interface)* du processus. Elles contiennent l'information nécessaire aux autres modules susceptibles de l'utiliser.

Syntaxiquement, DELAI est un *paramètre*. Il est considéré comme une constante, et doit être connu à la compilation. Le symbole **?** introduit les signaux d'entrée, avec leur type. Le **!** précède la déclaration des signaux de sortie.

```
7 :   (|
```

Début du corps du processus, composé d'un ensemble d'*équations* définissant des valeurs ou des contraintes sur les signaux. Ces équations sont encadrées par les séparateurs : $(|\ldots|\ldots|\ldots|)$.

```
8 :    HEURE ^= TOP
9 :  | HEURE := (HEURE$ init 0) + 1
```

Les signaux peuvent être des signaux d'entrée ou de sortie (comme *TOP*), ou des signaux locaux (comme *HEURE*), dont la déclaration est reportée en fin de texte. HEURE est un compteur servant à numéroter les TOPs du chronomètre; il permettra de connaître l'heure associée à une ALARME.

La ligne 8 impose que ce signal HEURE soit présent aux mêmes instants que TOP. HEURE et TOP ont la même **horloge**, ce sont de signaux **synchrones**. $\hat{}=$ est l'opérateur d'égalité d'horloge. Les équations de cette forme sont utilisées lorsque le contexte n'est pas suffisant pour déterminer à lui seul l'horloge d'un signal.

Son horloge étant fixée, la **valeur** de HEURE est définie ligne 9. À l'exemple d'autres langages déclaratifs, SIGNAL n'utilise pas d'instructions modifiant des variables, comme *HEURE := HEURE + 1*. On procède en amenant à l'instant présent la valeur qu'avait *HEURE* à l'instant précédent : c'est *HEURE$*. Sa valeur à l'instant initial est fixée à 0 par le *init*.

HEURE à l'instant présent est : (HEURE à l'instant précédent)+1. Cette définition ne donne par elle-même aucune indication sur les instants de présence de HEURE, et la ligne 8 est donc bien nécessaire.

```
10 :    | CPT ^= TOP ^+ COMMANDE ^+ TERMINEE
11 :    | ZCPT := CPT $ init (-1)
12 :    | CPT := DELAI when ^COMMANDE
13 :              default -1 when TERMINEE
14 :              default  ZCPT - 1 when ZCPT >= 0
15 :              default -1
```

Le temps séparant une COMMANDE de sa terminaison est décompté par un compteur décroissant CPT. Il est fixé à DELAI à l'arrivée d'une COMMANDE, puis décroit sur chaque TOP, soit jusqu'à l'arrivée de TERMINEE, soit jusqu'à 0 si la terminaison n'arrive pas.

L'égalité d'horloges de la ligne 10 précise les instants où *CPT* est présent. C'est l'union (opérateur ^+) des horloges des trois signaux d'entrée, c'est-à-dire l'ensemble des instants où au moins l'un d'entre eux est présent.

*ZCPT* porte la valeur de CPT à l'instant précédent, nécessaire pour calculer sa valeur actuelle. On l'initialise à -1, qui va être la valeur "de repos" du compteur, lorsqu'il n'y a pas d'attente de terminaison. La ligne 12 lui donne la valeur du DELAI lorsqu'une COMMANDE arrive. L'opérateur ^ extrait du signal COMMANDE son horloge, de type *event*.

Si à l'instant considéré, on n'a pas de COMMANDE, on regarde s'il y a une occurrence du signal TERMINEE ; dans ce cas le compteur prend directement la valeur -1. Il est inutile d'écrire ^TERMINEE, car ce signal est déjà de type *événement*.

En l'absence de l'un ou l'autre signal, le compteur se décrémente de 1 tant qu'il est positif ou nul ; sinon, il conserve la valeur -1. *when* et *default* sont des **opérateurs polychrones**. Le *when* est plus prioritaire que le *default*.

```
16 :   | ALARME := HEURE when CPT = 0
```

Le signal ALARME prend la valeur de HEURE si le compteur décroit jusqu'à 0. C'est donc que le signal TERMINEE n'est pas arrivé à temps pour le positionner directement à -1.

On note que HEURE est toujours présent lorsque CPT = 0, car l'horloge de CPT **inclut** celle de TOP, qui est égale à celle de HEURE.

```
17 :   |)
```

Cette ligne termine l'ensemble des équations. Ces équations peuvent être écrites dans un ordre quelconque : SIGNAL analyse les définitions pour établir leur ordre de dépendance, ainsi que les instants où ces expressions doivent être calculées.

Déclaration des signaux locaux :

```
18 :  where
19 :    integer HEURE, ZCPT, CPT
20 : end % VEILLEUR %
```

### 1.2.6 Mise en œuvre et exécution.

Ce processus constituera un module intégré à une application plus large, qui lui fournira ses signaux d'entrée. Le programme peut être saisi à l'éditeur de texte, ou construit au moyen d'un éditeur graphique décrit par ailleurs. Il est également possible de le compiler et de le faire fonctionner séparément, après avoir fixé la valeur du paramètre DELAI.

Le compilateur SIGNAL analyse la syntaxe, vérifie la définition des horloges, établit l'ordre de calcul des signaux. La compilation comporte une phase particulièrement importante : le **calcul d'horloges**. Il vérifie la bonne définition et la cohérence de l'horloge de chaque signal; mais surtout il établit les relations d'inclusion de certaines horloges dans d'autres, ce qui permet de n'effectuer le calcul d'un signal que lorsqu'il est présent.

Le code objet est produit sous forme de programmes en C (ou en Fortran), dont la compilation fournit l'exécutable final. Pour un langage "jeune", le passage par un langage intermédiaire de haut(?) niveau permet d'avoir un résultat de compilation lisible, de bénéficier de ses bibliothèques, et de connecter facilement un programme Signal avec d'autres modules écrits en C.

Les signaux d'entrée doivent être préparés sous forme de fichiers :

- un fichier *RCOMMANDE.dat* contient la suite des valeurs entières de COMMANDE (7 8 9 ...);

- mais il faut aussi fixer les horloges relatives des 3 signaux d'entrée : c'est le rôle de trois autres fichiers nommés *RC_COMMANDE.dat, RC_TERMINEE.dat, RC_TOP.dat*, de même longueur, et qui contiennent autant de *booléens* qu'il y a d'instants dans le scénario représenté : dans *RC_X.dat*, un *1* indiquant la présence du signal X, un *0* son absence.

Voici le contenu de ces fichiers, avec les données correspondant à l'exemple du début :

```
  RCOMMANDE.dat :  7  8  9
RC_COMMANDE.dat :  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0
RC_TERMINEE.dat :  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0
     RC_TOP.dat :  1  1  1  1  0  1  1  1  0  1  1  1  1  1  1  1  1  1  1  1
```

Il y a autant de valeurs dans *RX.dat* que de *1* dans *RC_X.dat*.

Le signal résultat est produit dans un fichier *WALARME.dat*, sans horloge.

Rajoutons les signaux locaux au scénario vu ci-dessus :

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COMMANDE : | ⊥ | 7 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 8 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 9 | ⊥ | ⊥ |
| TERMINEE : | ⊥ | ⊥ | ⊥ | ⊥ | 1 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 1 | ⊥ | 1 | ⊥ |
| TOP : | 1 | 1 | 1 | 1 | ⊥ | 1 | 1 | 1 | ⊥ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| HEURE : | 1 | 2 | 3 | 4 | ⊥ | 5 | 6 | 7 | ⊥ | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| ZCPT : | −1 | −1 | 5 | 4 | 3 | −1 | −1 | −1 | −1 | 5 | 4 | 3 | 2 | 1 | 0 | −1 | −1 | 5 | −1 |
| CPT : | −1 | 5 | 4 | 3 | −1 | −1 | −1 | −1 | 5 | 4 | 3 | 2 | 1 | 0 | −1 | −1 | 5 | −1 | −1 |
| ALARME : | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | 12 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |

# Chapitre 2

# Les objets du langage : les signaux

## 2.1 Qu'est-ce qu'un signal ?

Les automaticiens (entre autres) considèrent deux sortes de signaux, suivant le modèle du temps utilisé :

- les signaux **analogiques**, fonctions d'un argument réel : le temps continu
- les signaux **échantillonnés**, faisant référence à un temps discret ; ils proviennent souvent de l'échantillonnage de signaux analogiques.

C'est ce dernier type de signaux qui est considéré en SIGNAL.

Il ne s'agit pas seulement d'une **suite de valeurs** appartenant à un certain domaine. La notion de signal contient également la **suite des instants** où ces valeurs apparaissent : cette suite est ce qu'on appelle **l'horloge du signal**.

La *suite des instants* d'occurrence des valeurs d'un signal fait référence implicitement à une certaine mesure du temps, plus ou moins universelle. Mais toute mesure du temps ne peut se faire qu'en *mettant en relation* la suite d'événements avec une autre suite choisie comme référence : les vibrations d'un quartz, le passage du soleil au point le plus haut de sa course, etc. Pour comparer les instants d'occurrence de deux signaux, il faut pouvoir décider si **deux occurrences sont simultanées**, ou si elles ont lieu à des instants différents.

La mise en œuvre d'un tel mécanisme de décision peut s'avérer délicate. Une hypothèse fondamentale pour la sémantique du langage SIGNAL est que, *pour tous les signaux considérés, on peut toujours décider si deux valeurs de signaux sont présentes simultanément ou non.* Corollairement, on pourra dire qu'un signal est absent - ne présente pas de valeur - alors qu'un autre est présent. Par contre, on ne suppose pas l'existence d'un signal prédéfini ayant valeur d'horloge universelle.

## 2.2 Ports, traces, flots

Comme dans tout langage de programmation, un signal particulier est désigné par un identificateur, auquel est associé le type des valeurs que peut prendre le signal. Ces identificateurs se distinguent cependant des variables classiques par plusieurs points :

- ces *variables* n'ont de valeurs qu'à des instants bien précis, et ne sont pas accessibles en dehors de ces instants (notion de *valeur absente*)

- Si une variable classique peut changer plusieurs fois de valeur au cours d'un programme (par exemple dans une itération), on ne s'intéresse jamais aux valeurs passées de cette variable. En SIGNAL, par contre, on fera souvent référence à la valeur précédente du signal $S$, ou à la N-ième occurrence passée de $S$, ou même à la suite des M dernières valeurs du signal (fenêtre glissante).
- Une *variable* SIGNAL évoque davantage une *suite de valeurs* plutôt qu'une valeur unique, même si à un instant donné, une variable n'a qu'une valeur, ou est absente.

C'est pourquoi on préfère souvent le terme de **port** à celui de variable, en parlant de valeur présente sur - ou portée par - un port.

Pour dénoter l'absence de valeur sur un port à un instant donné, on introduit le symbole *bottom*, noté $\perp$. Nous l'ajouterons systématiquement au domaine des valeurs des signaux.

Soit $A = \{a_1, a_2, ..., a_n\}$ un ensemble de signaux utilisés dans une application. A un instant donné, chacun de ces signaux porte une valeur, ou est absent. On appelle **événement** l'ensemble de ces valeurs de signaux à cet instant, en donnant la valeur $\perp$ aux signaux absents.

Une **trace** sur $A$ est une suite d'événements portant les valeurs des signaux de $A$ en des instants successifs. Une trace peut être visualisée en représentant sur une verticale les valeurs des signaux à un même instant.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $a_1$ : | 1 | 2 | $\perp$ | $\perp$ | 7 | $\perp$ | 5 | $\perp$ |
| $a_2$ : | $\perp$ | 10 | $\perp$ | $\perp$ | 123 | $\perp$ | 5 | $-1$ |
| $a_3$ : | 0 | 6 | $\perp$ | $\perp$ | 13 | $\perp$ | 2 | $\perp$ |
| $a_4$ : | $\perp$ | 11 | $\perp$ | $\perp$ | $\perp$ | $\perp$ | 1 | $\perp$ |

En se limitant aux quatre signaux représentés ci-dessus, on constate qu'il ne s'est rien passé aux instants 3, 4 et 6 : ce sont des *événements vides*. De telles traces n'auraient d'intérêt que s'il existait un autre signal permettant de *marquer* ces instants où il ne se passe rien. Sinon, ils peuvent être supprimés sans inconvénient. Le *dernier* événement non vide peut cependant être suivi d'un nombre quelconque d'événements vides.

Un **flot** est une trace ne contenant aucun événement vide, ou dont tous les événements sont vides à partir du premier événement vide qu'elle contient. Voici le flot correspondant à la trace ci-dessus :

| | | | | | |
|---|---|---|---|---|---|
| $a_1$ : | 1 | 2 | 7 | 5 | $\perp$ |
| $a_2$ : | $\perp$ | 10 | 123 | 5 | $-1$ |
| $a_3$ : | 0 | 6 | 13 | 2 | $\perp$ |
| $a_4$ : | $\perp$ | 11 | $\perp$ | 1 | $\perp$ |

### Comparaison d'horloges

Deux signaux sont dits **synchrones** s'ils sont présents (ou absents) exactement aux mêmes instants. C'est le cas dans l'exemple ci-dessus de $a_1$ et $a_3$. Le synchronisme est une relation d'équivalence entre signaux, et chaque classe d'équivalence définit une horloge : tous les signaux appartenant à la même classe ont la même horloge.

Il existe aussi une relation d'ordre sur les horloges : l'horloge de $a_4$ est *moins fréquente* (ou *inférieure*) à celle de $a_3$, car tous les instants de $a_4$ sont des instants de $a_3$, alors que $a_3$ peut être présent sans que $a_4$ soit là.

Cette relation d'ordre n'est que partielle : ainsi les horloges de $a_1$ et $a_2$ ne sont pas comparables.

## 2.3 Types des valeurs des signaux

Les signaux peuvent avoir des valeurs booléennes (*vrai* ou *faux*), ou numériques : entiers, réels en simple ou double précision, complexes. Par extension, on considère que les signaux *purs*, dont le seul intérêt réside dans leur horloge, portent néanmoins une valeur unique - la valeur *vrai* - de type *event*. Une valeur de signal peut être également un regroupement (*tableau*) de valeurs simples, avec une horloge unique pour tout le tableau. Les tableaux sont présentés dans un chapitre ultérieur.

### 2.3.1 Type *event*

Ne pas confondre avec un *événement* défini ci-dessus comme l'ensemble des valeurs de divers signaux à un instant donné. On parle également de *top*, ou *d'impulsion*. Un objet de ce type n'a qu'une valeur possible, le *vrai* des booléens. Remarquer qu'on n'a jamais besoin de représenter cette constante, du moins comme valeur de type *event*.

Un signal de type *event* peut être produit par les opérateurs unaires *horloge (^)* et *when*.

#### Opérateur horloge

Si S est un signal de type quelconque, l'opération unaire **^S** définit un signal de type *event* qui est l'horloge que S. Par définition, $S$ et $\hat{}S$ sont synchrones.

Dans l'exemple du VEILLEUR présenté dans l'introduction, on aurait pû ne transmettre au processus que l'horloge de COMMANDE, par *^COMMANDE*, car sa valeur exacte (et même son type) sont sans intérêt. COMMANDE n'est d'ailleurs utilisé dans l'exemple que à travers son horloge.

```
 COMMANDE : ---- 7 -------- 8 --------------- 9 ------

^COMMANDE : ---- t -------- t --------------- t ------
```

#### Opérateur when

*when condition* est un opérateur temporel, qui extrait les valeurs *vrai* de l'expression booléenne (n'existe que lorsque la condition est *true*). Il est présenté plus loin.

### 2.3.2 Type *boolean*

Une valeur de ce type est obtenue par les notations de constantes *true* et *false*, ou par un signal de type *event*, ou par l'un des six opérateurs de comparaison : $=$ $/=$ $<$ $<=$ $>$ $>=$, ou par les opérateurs *NOT, AND, OR*. Les priorités sont celles que l'on rencontre dans la plupart des langages (sauf Pascal). Il est préférable de parenthéser l'opérande d'un NOT s'il contient des opérateurs. L'expression :

```
NOT (L1/=L2) OR L3 AND X=0
```

(où les $Li$ sont booléens) est correcte, et implicitement parenthésée ainsi :

```
(NOT (L1/=L2)) OR (L3 AND (X=0))
```

### 2.3.3 Type *integer*

Opérateurs habituels : $+$, $-$ unaires, $**$, $*$, $/$, *modulo*, $+$, $-$ binaires.

- utilisation de fonctions : $F(argument, ...)$ ; ces fonctions doivent être définies dans la section des déclarations locales (voir exemple plus loin).
- les fonctions prédéfinies (telles que la racine carrée *sqrt*, ou la valeur absolue *abs*) sont celles du langage objet intermédiaire (C) ; seul leur en-tête doit être déclaré localement (exemple en fin de chapitre)
- le seul opérateur de division est le /. Si I et J sont entiers, I / J donne le quotient entier ; sinon, le résultat est réel
- les règles de conversion de type sont celles du langage intermédiaire.

### 2.3.4 Types *real, dreal,...*

Notations habituelles. Toute expression à résultat entier est acceptable en opérande lorsque l'autre opérande est réel. Les fonctions prédéfinies suivent les mêmes règles que pour les entiers.

## 2.4 Déclarations de signaux

Tous les signaux intervenant dans un processus doivent être déclarés, précédés de leur type. Il peut s'agir de signaux d'entrée, définis dans l'interface derrière le symbole **?**, ou de signaux de sortie, derrière le **!**, ou encore de signaux locaux, déclarés en fin de processus après le symbole **where**. Les signaux de sortie et les signaux locaux peuvent recevoir une valeur initiale constante lors de leur déclaration.

L'exemple suivant illustre divers points de syntaxe :

```
process DECLARATIONS =
  ( ? real a;
      event UN, HH2; integer B
    ! boolean ok init false, TROUVE )

  (| XX := sqrt(fabs(-A))
   |  ...
   | OK := inter <= 1 when TROUVE
   |)
where
  real XX, YY init -1.5;   % seul YY est initialise %
  integer inter;
     % en-têtes de fonctions %
  function sqrt = (? dreal A !dreal B);  % en C, paramètre ‘‘double’’ %
  function fabs = (? dreal A !dreal B)
end
```

# Chapitre 3

# Définitions des signaux. Opérateurs

Dans un processus, des signaux d'entrée sont transformés, au moyen d'un certain nombre d'opérations, pour obtenir des signaux de sortie. Ces transformations nécessitent le plus souvent l'élaboration de signaux intermédiaires, définis localement.

Le présent chapitre étudie les opérations de base permettant de définir un signal. Chacun des opérateurs devra fixer à la fois **l'horloge** et la **valeur** du signal produit.

## 3.1 Définition d'un signal

### 3.1.1 Syntaxe

Un signal est défini dans les cas simples par une *équation* de la forme :

```
NOM du signal  :=  EXPRESSION de definition
```

Exemple :

```
OK := INTER <= 1 when TROUVE
```

L'expression de définition combine divers signaux et constantes au moyen d'opérateurs. Ces opérateurs sont :

- soit des opérateurs liés directement aux types des signaux, et agissant sur leurs valeurs, tels qu'ils ont été vus au chapitre précédent. Tous leurs opérandes doivent être présents aux mêmes instants (opérateurs **monochrones**). Dans l'exemple, <= compare deux valeurs entières pour donner une valeur booléenne. L'accès à une valeur passée d'un signal (retard) est également un opérateur monochrone.
- soit des opérateurs plus généraux, dits **polychrones**, faisant intervenir l'horloge des signaux ; le *when* de l'exemple indique que OK ne prendra de valeur que lorsque le signal TROUVE sera présent en même temps que INTER, et que ce signal TROUVE aura la valeur *vrai*.

Le type de la valeur rendue par l'expression doit être **le même** que celui du signal défini. La seule exception est que si l'expression est un *event*, le signal défini peut être un booléen (*boolean*). Pour les autres conversions de type, utiliser les fonctions prédéfinies du langage intermédiaire.

### 3.1.2 Détermination de l'horloge d'un signal

L'horloge du signal défini doit être déterminée avant sa valeur, car ce n'est que lorsque le signal est présent qu'il vaut quelque chose !

*Le signal défini et l'expression de définition sont synchrones.*

Le plus souvent, c'est l'expression à droite du $:=$ qui suffit à définir entièrement cette horloge. C'est le cas dans l'exemple ci-dessus : OK est présent lorsque l'on a à la fois INTER présent, et TROUVE présent et vrai ; OK est absent sinon. Cela suppose néanmoins que les horloges de INTER et TROUVE soient parfaitement fixées.

Dans d'autres cas, l'horloge élaborée par l'expression se révèle insuffisamment précisée. Par exemple, $S := 1$ ne définit S que comme une suite de valeurs 1, pouvant être présentes n'importe quand.

L'horloge du signal calculé doit alors être fixée par ailleurs :

- comme effet de bord d'une autre équation ; par exemple, si un signal X est défini par $X := A + S$, l'opérateur $+$ contraint ses opérandes à avoir la même horloge (opérateur monochrone). Si l'horloge de A est fixée, celle de S le devient également, comme celle de X. Si c'est l'horloge de X qui est fixée par une autre équation, A et S auront (ou devront avoir) cette même horloge.
- les **contraintes sur horloges** : l'équation $S \hat{} = A$, mêlée aux autres définitions de signaux, contraint les horloges de S et de A à être égales. Plus généralement, un opérande de $\hat{} =$ peut être une expression sur signaux.
- Dans le processus :

```
process UNS =
  ( ?     % pas d'entrees %
    ! integer S )
(| S := 1 |)
```

  l'horloge de S peut être fixée par un processus extérieur qui utilise *uns* (sous-processus appelé par un processus principal).

  Si on le fait fonctionner isolément, l'horloge de S sera une *horloge de base* supposée plus rapide que toutes les autres horloges. Concrètement, une suite infinie de 1 sera produite à la vitesse de la machine exécutant ce processus !

- Par contre, si un processus produit deux signaux indépendants :

```
process DEUX =
  ( ?
    ! integer S1, S2)
(| S1 := 1
 | S2 := 2 |)
```

  on devra : soit rajouter une contrainte sur les horloges

```
(| S1 := 1 | S2 := 2 | S1 ^= S2 |)
```

  l'horloge commune étant alors l'horloge de base ;
  - soit fixer depuis un processus appelant les horloges de S1 et S2, si elles sont différentes. En cas d'exécution isolée, deux *fichiers d'horloge* RC_S1 et RC_S2 devront alors être fournis.

Il peut arriver que l'horloge d'un signal soit déterminée - plus ou moins complètement - de plusieurs manières. Ces définitions doivent bien entendu être cohérentes. Cette cohérence doit même pouvoir être vérifiée à la compilation.

Par contre, dans un processus donné, un signal ne peut avoir qu'une seule équation de définition.

### 3.1.3 Élaboration des définitions de signaux

Le corps d'un processus contient un ensemble de définitions de signaux et de contraintes sur horloges, écrites dans un ordre quelconque, et séparées par des barres verticales.

Le *calcul d'horloges*, effectué par le compilateur, détermine la suite des instants où au moins un signal peut posséder ou prendre une valeur.

Ainsi, dans l'exemple :

```
OK := INTER <= 1 when TROUVE
```

OK ne peut être présent qu'aux instants où INTER et TROUVE sont présents simultanément. À partir des définitions de INTER et TROUVE, le compilateur détermine leur horloge, en prouvant par exemple qu'ils sont synchrones, ou que les instants de l'un sont inclus dans ceux de l'autre. Il en déduit que l'horloge de OK est *inférieure à* (incluse dans) ces horloges de INTER et TROUVE. Ce calcul d'horloges est présenté dans un chapitre ultérieur sur la compilation de SIGNAL.

L'horloge de OK ne peut cependant être déterminée entièrement à la compilation, car elle dépend de la *valeur* de TROUVE. A l'exécution, les instants sont considérés dans l'ordre croissant du temps. Les signaux pouvant être définis à cet instant sont évalués, en appliquant leur expression de définition. Il en résulte soit une valeur, soit le constat que le signal est absent.

Ce calcul de la valeur d'un signal S à un instant donné peut faire intervenir :

- des constantes, toujours définies
- des valeurs de signaux d'entrée présents à cet instant
- des valeurs d'autres signaux, présents à cet instant et déja calculés
- des valeurs de S à des instants précédents (voir opérateur **retard** plus loin)
- des valeurs retardées d'autres signaux présents à cet instant (mais pas forcément déja calculés).

Il existe donc une relation de dépendance entre signaux, qui doivent être calculés dans un certain ordre.

Ces relations de dépendance sont évaluées à la compilation. Elles ne doivent pas comporter de cycle, c'est-à-dire qu'un signal ne peut dépendre - directement ou non - de sa propre valeur à l'instant présent. C'est ce qui empêche d'écrire de définitions telles que : $S := S + 1$ ! Par contre, les valeurs d'entrée et les valeurs retardées (de S ou d'un autre signal) sont disponibles dès le début du calcul.

## 3.2 Opérateurs monochrones

Les signaux intervenant comme opérandes de ces opérateurs doivent être tous **synchrones** (présents ou absents aux mêmes instants). Si leurs horloges sont déja fixées par

ailleurs, l'égalité des horloges est vérifiée. Si certaines horloges étaient libres, elles deviennent contraintes à être égales entre elles.

### 3.2.1 Opérateurs propres aux types des signaux

Il s'agit des opérateurs arithmétiques, de comparaison, etc, ainsi que des fonctions, vus au chapitre précédent. On ne peut additionner ou comparer deux valeurs que si elles sont présentes au même instant. Le résultat apparait également à ce même instant, car on suppose le temps de calcul négligeable.

### 3.2.2 Opérateur *horloge*

On rappelle cet opérateur, vu au chapitre précédent ; le résultat de $\hat{}X$, de type *event*, est présent aux mêmes instants que X (horloge de X).

### 3.2.3 Valeurs précédentes d'un signal. Opérateur retard

#### Définition

A $ est la valeur précédente de A. A $ N est la valeur qu'avait le signal N instants auparavant, en ne comptant que les instants où A est présent.

N est une expression entière constante positive. Elle peut contenir des identificateurs de *paramètres* (voir *Appels de processus*), connus à la compilation.

Exemple :

```
PIXEL $ (NBLIGNES * NBCOLONNES - 1)
```

L'horloge du signal retardé est la même que celle du signal initial.

#### Initialisation

Les N premières valeurs de A $ N sont indéfinies. Elles peuvent être initialisées "sur place" par:

```
A $ init 0  ou Y := 5*( X $ 2 init [10,20])+1
```

ou bien lors de la déclaration d'une variable portant la valeur retardée :

```
   | ZA := A $ 1
   | ZB := A $
   | ZC := A $ 3
where
   integer ZA, ZB init 0, ZC init [10,20,30]
```

ce qui produit :

```
   A  : ---  1 ---  2 -------  3 --- 4 --- 5 ------- 6 ---

  ZA  : ---  ? ---  1 -------  2 --- 3 --- 4 ------- 5 ---

  ZB  : ---  0 ---  1 -------  2 --- 3 --- 4 ------- 5 ---

  ZC  : --- 10 --- 20 ------- 30 --- 1 --- 2 ------- 3 ---
```

Lorsque A est absent (ce qui suppose que d'autres signaux non représentés soient présents), les valeurs retardées de A sont également absentes.

**Utilisation**

Pour produire un signal égal à 1, 2, 3, 4,... on peut écrire :

```
process compteur =
 ( ?
   ! integer C )
 (| C := (C$ init 0) + 1 |)
```

ou passer par un autre signal portant la valeur retardée de C :

```
process compteur =
 ( ?
   ! integer C )
 (| C := ZC + 1
  | ZC := C$ |)
where
   integer ZC init 0
end
```

L'horloge de C est libre. Pour numéroter les occurrences d'un signal d'entrée A, il suffit que chaque valeur de C soit associée à une occurrence de A ; autrement dit, on donnera à C l'horloge de A :

```
process compteur =
 ( ? real A
   ! integer C )
 (| A ^= C
  | C := (C$ init 0) + 1 |)
```

**Exercices :**

1. Un même bouton, concrétisé par un signal BOUTON de type *event*, permet de démarrer ou d'arrêter un appareil. Produire un booléen ENMARCHE indiquant l'état de l'appareil après chaque action. L'appareil est initialement à l'arrêt.
2. Produire pour chaque occurrence d'un signal d'entrée réel A le cumul (la somme) de toutes les valeurs de A entrées jusque là.
3. Produire la moyenne de toutes les valeurs entrées.

4. Produire un signal booléen PREMIER de même horloge que A, égal à *vrai* sur la première occurrence de A, et à *faux* sur toutes les suivantes ; il est possible de ne pas utiliser de compteur !

#### Calculs sur une fenêtre d'un signal

On appelle *fenêtre* une suite de N valeurs consécutives d'un signal ; en général, il s'agit de la valeur courante et des N-1 valeurs précédentes. Beaucoup de calculs font intervenir ces valeurs : filtrages, convolutions, lissages, etc. Si N est petit et connu, on peut programmer directement un tel calcul :

```
| AA := (4 * A + 2 * A $1 + A $2) / 7
```

Sinon, les N valeurs doivent être rangées dans un tableau, présenté par ailleurs.

#### Exercice :

Produire pour un signal **A** la moyenne des N dernières valeurs, où N est un paramètre constant entier. Indication : on n'a pas besoin de considérer les N valeurs, mais seulement A et A $ N, avec la valeur précédente de la somme.

L'initialisation de N valeurs à 0 s'écrit : *init* [ { *to N* } : 0 ]. Les moyennes produites sur les premières valeurs de A sont sans intérêt.

## 3.3 Opérateurs polychrones

Ces opérateurs combinent des expressions de définition de signaux pouvant avoir des horloges différentes, le résultat ayant également sa propre horloge.

On ne présente ici que les opérateurs de base : le **when** *unaire* qui extrait les valeurs *vrai* d'un signal booléen, le **when** *binaire*, ou échantillonnage, qui extrait certaines valeurs d'une expression sur signaux, et le **default**, qui mélange des signaux d'horloges quelconques.

### 3.3.1 Opérateur *when* unaire

#### Définition

```
when <expression booléenne sur signaux>
```

Exemple : $NUL := when\ CPT = 0$

Le résultat est de type *event*, n'ayant d'intérêt que par son horloge. Il est présent - et égal à *vrai* - lorsque le signal élaboré par l'expression booléenne est présent et *vrai*. Sinon (résultat de l'expression absent, ou présent mais avec la valeur *faux*), le *when* ne produit pas de valeur.

Dans l'exemple, NUL n'est présent que sur les instants de CPT pour lesquels la valeur de CPT est nulle. Son horloge est *inférieure à* (incluse dans) celle de CPT.

```
CPT  : --- 1 --- 0 ------ 2 --- 0 --- 0 --- 3 ----

NUL  : --------- t ------------ t --- t ----------
```

**Utilisation**

Le *when* est utilisable dans tout contexte acceptant un événement. Il peut servir à fixer l'horloge d'un signal aux instants où une condition est vérifiée :

```
| S ^= when condition
```

Pour compter le nombre de valeurs *false* d'un signal booléen B :

```
| CPT ^= when not B
| CPT :=(CPT $ init 0) + 1
```

**Exercices**

1. Soit A un signal d'entrée, de type entier, positif ; émettre un signal MONTE, de type *event*, chaque fois qu'une valeur de A est supérieure à la valeur précédente.
2. Émettre un événement SOMMET dès que l'on constate que les valeurs de A sont passées par un maximum local (... ce qui n'est possible que sur la première valeur décroissante !). Supposer d'abord que 2 valeurs consécutives de A ne peuvent être égales.
   Initialisation : la première valeur de A n'est considérée comme un maximum que si elle est suivie d'une valeur inférieure.
3. Prendre ensuite en compte les paliers, à ne détecter que s'ils sont suivis d'un changement de sens effectif.

### 3.3.2 Échantillonnage. Opérateur *when* binaire

Cet opérateur permet d'extraire (*d'échantillonner*) les occurrences d'un signal en fonction d'une certaine condition booléenne.

**Définition**

```
<expression signal> when <condition>
```

Exemples :

```
ALARME := HEURE when CPT = 0

OK := INTER <= 1 when TROUVE
```

L'opérateur *when* a une priorité plus faible que les opérateurs arithmétiques ou de comparaison.

Le signal résultat n'est présent qu'aux instants où l'expression et la condition sont présents simultanément, et si à cet instant la condition a la valeur *vrai*. L'horloge de *A when B* est l'intersection des horloges de *A* et de *when B*.

La valeur du résultat est alors celle de l'expression. OK est donc ici de type booléen.

```
 INTER  : --- 0 ------- 0 --- 0 --- 3 --- 1 --- 1 --- 1 ---

TROUVE  : -------- t -- f --- t --- t --------- f --- t ---

    OK  : ------------------- t --- f --------------- t ---
```

**Utilisation**

L'expression *A when condition(A)* permet d'extraire certaines valeurs d'un signal A : celles qui vérifient la condition.

```
S := NOTE when NOTE >= 0 and NOTE <= 20
```

S ne contiendra que les valeurs de NOTE comprises entre 0 et 20.

L'expression devant le *when* peut être une constante : l'horloge résultante est alors celle de *when B*. En particulier, *true when B* est équivalent à *when B*.

```
S := 0 when A < 0
```

Le *signal* constant 0 est toujours présent. S n'est présent (et nul) que sur les valeurs de A négatives.

Il peut être tentant d'utiliser le *when* pour synchroniser un signal S avec un autre signal B d'horloge fixée :

```
S := A when ^B
```

qui se veut la mise en forme de la phrase : S est égal à A lorsque B est présent. On doit cependant remarquer que l'horloge de S ne sera pas celle de B, mais **l'intersection** de l'horloge de A et de celle de B ! Il est en particulier équivalent d'écrire :

```
S := 1 when ^B
ou
| S := 1
| S ^= B
```

Par contre, l'écriture suivante :

```
C := (C $ init 0) + 1 when ^B    % ERREUR !! %
```

est **erronée**, si l'on n'a pas dit par ailleurs que les horloges de C et de B étaient les mêmes. Pour le compilateur, la partie droite a pour horloge l'intersection des horloges de C et de B; il considère que cette intersection n'a aucune raison d'être égale à l'horloge de C,... même s'il n'y a pas d'autre possibilité pour que l'écriture soit correcte! Le message d'erreur est un *clocks constraints*.

On ne peut compter correctement les occurrences du signal B qu'en écrivant :

```
| C := (C $ init 0) + 1
| C ^= B
```

**Exercices**

1. Simplifier l'expression : *true when ^A*
   Pourrait-on détecter les instants où A est absent en écrivant : *when not (^A)*?
2. Soit A un signal entier positif ; sortir les valeurs de A correspondant à des maxima locaux ; voir dans un exercice précédent comment détecter ces sommets.
3. Sortir une valeur toutes les N d'un signal A, N étant une constante. Commencer par sortir la première valeur de A. Utiliser le modulo.
4. Un signal booléen B comporte alternativement des suites de valeurs *true* et *false*, suites de longueur $\geq 1$. Produire un signal PREMB extrait de B, et ne comportant que la **première** valeur de chacune de ces suites - donc alternativement *true* et *false*. La première valeur de PREMB doit bien être la première valeur de B, que celle-ci soit *true* ou *false* !

5. S1 et S2 sont deux signaux indépendants, de type quelconque. Sortir un top (*event*) aux instants où S1 et S2 sont présents simultanément. Attention, on ne peut utiliser un *and* que si ses opérandes sont synchrones !

### 3.3.3 Mélange. Opérateur *default*

Cet opérateur permet de mélanger des signaux d'horloges quelconques.

**Définition**

```
< expression signal >  default  <expression signal >
```

Exemple :

```
S := A default B
```

mélange les deux signaux A et B, avec priorité pour A.

Plus précisément, lorsque A est présent, S est égal à A. Si A est absent, mais B présent, S est égal à B. Sinon, S est absent. L'horloge de S est donc l'union des horloges de A et de B.

```
         A  : ---- 1 ----------  2 ---- 3 -------------- 4 ---  5 ---

         B  : ---------- 10 --- 20 --------- 30 -- 40 -------- 50 ---

A default B : ---- 1 --- 10 ---  2 ---- 3 -- 30 -- 40 -- 4 ---  5 ---
```

Les types des deux expressions doivent être compatibles.
La priorité du *default* est inférieure à celle du *when*.

**Utilisation**

– Si l'expression de droite est une constante (celle de gauche ne peut jamais l'être !), l'horloge de l'opération est indéfinie, et doit être fixée par le contexte. Ainsi, dans :

```
S := B default false
```

où S et B sont des booléens, l'horloge de S doit être fixée par ailleurs, et doit inclure l'horloge de B dans ses instants possibles. Par exemple :

```
S ^= ^X default B
```

(on passe par l'horloge de X pour que le type de l'opérande gauche du *default* soit compatible avec le type *boolean* de B); ou mieux, avec l'opérateur *union d'horloges* :

```
S ^= X ^+ B
```

– L'expression :

```
S := A when condition default B
```

est fréquemment utilisée : elle simule la conditionnelle des langages classiques : *si condition alors S := A sinon S := B*. Une différence importante est que S prend la valeur de B non seulement lorsque la condition est fausse, mais aussi lorsque la condition est absente, ou lorsque A est absent, que la condition soit vraie ou fausse. Ne pas oublier qu'il s'agit du mélange du signal *(A when condition)* avec le signal *B*.

– Dans une expression telle que

```
| SIGNE := 1 when X >= 0 default -1
```

l'horloge de SIGNE n'a aucune raison d'être celle de X; elle doit être fixée par ailleurs, par exemple par :

```
SIGNE ^= X
```

Si l'on ajoutait *when* $X < 0$ derrière -1, le compilateur retiendrait que l'horloge de *SIGNE* est inférieure à (ou extraite de) celle de $X$, puisque les deux branches du *default* ont cette propriété. La contrainte d'égalité ci-dessus provoquerait alors une erreur : pour prouver l'égalité des horloges, il faudrait pouvoir démontrer que l'union des conditions recouvre tous les cas possibles, ce qui n'est pas possible dans le cas général.

Par contre, les expressions *f(A) when cond(X) default g(A)* ou *f(X) when cond(A) default g(A)* ont bien l'horloge de A.

– On peut mélanger plus de deux signaux par *A default B default C ...* Des tests en cascade peuvent ainsi être écrits :

```
| S := val1 when cond1
    default val2 when cond2
    default val3 when cond3
```

Par contre, les multi-définitions sont interdites :

```
| S := val1 when cond1
| S := val2 when cond2   % Erreur : définition multiple !! %
```

Voir aussi l'exemple du VEILLEUR, dans l'Introduction.

**Exercices**

1. Afin d'étudier l'expression *A when B default C*, dessiner les 3 ensembles des instants de A, B, C, à intersections non vides, l'ensemble B étant lui-même partagé en *vrai/faux* (cercle intérieur à B, d'intersection non vide avec A et C). Colorier chacune des 11 zones selon que l'expression vaut A, ou C, ou est absente.
2. Dans quel cas l'expression suivante est-elle correcte ?

   ```
   B := true when condition default false
   ```

   Lorsqu'elle est correcte, peut-elle toujours être simplifiée en :

   ```
   B := condition
   ```
3. Soient S1 et S2 deux signaux indépendants; produire à chaque arrivée de l'un d'eux : un 1 s'il s'agit de S1 seul, un 2 s'il s'agit de S2 seul, un 3 s'ils sont ensemble.
4. Produire un signal MAX, de même horloge que A entier positif, égal à la plus grande valeur de A rencontrée jusqu'à présent.
5. Tenir à jour un signal entier PRESENTS, initialement nul, incrémenté de 1 par l'arrivée d'un *event* ENTREE, et diminué de 1 par un autre signal SORTIE. Une SORTIE peut avoir lieu au même instant qu'une ENTREE.
6. Programmer la soustraction de signaux *event* : A - B, le résultat étant présent sur les instants de A où il n'y a pas aussi de B.
7. Soient S1 et S2 deux signaux indépendants. Émettre un signal *event UNSEUL* lorsque un seulement des deux signaux est présent.

8. Soient A et B quelconques en entrée, de même type. Produire un signal de sortie constitué alternativement d'une valeur prise dans A et d'une valeur prise dans B. S'il y a une suite de A consécutifs sans B, seul le premier A de la suite est sorti ; même chose pour B. Si deux occurrences de A et B sont simultanées, l'une des deux est donc toujours sortie. La première valeur sortie est celle du premier signal arrivé, ou l'une quelconque si les deux premières occurrences sont simultanées.

# Chapitre 4

# Extensions

## 4.1 Répétition sur une autre horloge (opérateur cell)

On ne peut effectuer des calculs sur les valeurs de signaux X et Y que s'ils ont la même horloge. Si ce n'est pas le cas, on doit par exemple *propager* les valeurs de X aux instants où Y est présent.

```
 | XY ^= X ^+ Y
 | XY := X default XY$
```

XY porte la dernière valeur de X, mais est présent à la fois sur X et sur Y. On pourrait le restreindre aux instants de Y par : **XY when ^Y**.

Ce problème étant fréquent, on a introduit en SIGNAL un opérateur spécial, le **cell** :

```
  C := A cell B
```

B est un signal booléen. Son équivalent est :

```
  | C ^= A ^+ (when B)
  | C := A default C$
```

**horloge du résultat :** *A ^+ when B* : instants de A, plus instants où B est vrai.

**valeur du résultat :** *valeur de A* lorsque A est présent, ou le *dernier A* si A est absent et B vrai, ou une *valeur initiale* de C lorsque B est vrai avant la première valeur de A.

```
        A  :  --------- 1 ------------------ 2 ------- 3 ------------

        B  :  --- t --------- t -- f -- t ------- t -- f -- f -- t --

C (init 0) :  --- 0 --- 1 --- 1 ------- 1 -- 2 -- 2 -- 3 ------- 3 --
```

La priorité du **cell** est égale à celle du **when**.

La valeur initiale éventuelle est à préciser dans la déclaration de C ; elle n'est prise en compte que lorsque la partie droite est limitée à l'opération **cell**.

### Utilisation

Dans le problème initial, on peut donc propager les valeurs de X par **X cell ^Y**, et les restreindre aux instants de Y par **(X cell ^Y) when ^Y**.

La valeur à propager peut être en particulier un paramètre de l'application donné une seule fois sous forme d'un signal au début du calcul. Ne pas confondre avec les constantes, données entre accolades dans l'en-tête du processus, comme le DELAI dans l'exemple du VEILLEUR. Ces constantes, connues dès la compilation, sont disponibles à tout instant ; leur valeur effective est fixée à l'appel du processus.

```
process monome =
  { integer N }  % parametre constant %
  ( ? real COEF, % une valeur donnee en debut de calcul %
           X
    ! real Y )   % Y = COEF * X ** N pour tout X %

 (| Y := ((COEF cell ^X) when ^X ) * X ** N |)
```

**Exercices :**

1. Soit une image à NL lignes et NC colonnes, balayée ligne par ligne par un point, au rythme d'arrivée d'un signal booléen PIXEL. Tenir à jour et sortir pour chaque PIXEL les coordonnées LIGNE et COLONNE du point courant. Le balayage reprend en haut à gauche à la fin de chaque image. Utiliser le *cell* pour conserver le numéro d'une ligne pendant qu'on la parcourt.
2. *Resynchronisation* : soit un signal d'entrée A, et une horloge H, plus rapide que A. Resynchroniser A, c'est le *déplacer* (sans le répéter) sur le top de H le plus proche.
3. Le fonctionnement d'un passage à niveau est un exemple intéressant de système temps réel :
   - Considérons d'abord une seule voie, à sens unique. Un capteur *AVANT* émet un top en temps utile avant l'arrivée d'un train au passage à niveau ; un autre capteur *APRES* signale que le train est passé. Le moteur de la barrière est commandé par un signal booléen *FERMER*, à *vrai* pour fermer, à *faux* pour rouvrir. Élaborer ce signal *FERMER* en fonction de *AVANT* et de *APRES*.
   - Les capteurs sont en fait actionnés par chaque roue du train, et envoient donc de nombreuses impulsions consécutives. Par contre, pour ne pas perturber la commande de la barrière, le signal *FERMER* ne doit être produit qu'une fois. Modifier son calcul en conséquence. On suppose qu'un train passe en entier sur un capteur avant d'atteindre l'autre.
   - Considérer maintenant deux voies, ayant chacune leurs capteurs. Attention, les signaux reçus (éventuellement contradictoires) peuvent être simultanés ! Exemple :

     ```
     Fermer Voie A : --T--------F-----T-----F--T--F-----
     Fermer Voie B : -----T--F-----T--F--T--F--T-----F--
     Résultat      : --T--------F--T--------F--T-----F--
     ```

     Une solution avec compteur(s) peut être évitée.

## 4.2 Modularité

Un processus peut être utilisé comme "sous-programme" par d'autres processus. Le processus appelant fixe les paramètres constants éventuels, fournit les signaux d'entrée,

récupère les signaux de sortie. La syntaxe se rapproche de celle d'un appel de fonction dans les langages classiques :

```
| RES := monome {2} (A2, X)
```

Le processus *monome*, déclaré dans la section précédente, a un paramètre constant, deux signaux d'entrée, et un signal de sortie. Le paramètre constant effectif peut être une expression dépendant de paramètres constants locaux (mais pas de signaux). Les signaux effectifs d'entrée transmis au processus appelé peuvent être des expressions quelconques sur des signaux de l'appelant. Il n'y a pas de "variables globales".

Les signaux de sortie du processus appelé sont désignés chez l'appelant par des noms de signaux placés à gauche du := (entre parenthèses s'il y en a plusieurs).

```
| ...
| (Y1, B) := proc (when INTER > 0 )
| ...
```

Le signaux de l'appelant et de l'appelé doivent évidemment correspondre en nombre, en ordre et en type.

S'il n'a qu'un signal de sortie, un processus peut être utilisé comme une fonction dans une expression :

```
Y := A0 + monome {1} (A1, X) + K * monome {2} (A2, X)
```

Exemple complet :

```
1: process pendule = % affiche chaque seconde : jour, heure, minute, seconde,
2:              à partir d'une source de tops battant la seconde  %
3:   ( ? event TOPS
4:     ! integer JOUR, HEURE, MINUTE, SECONDE
5:   )
```

Le processus principal *pendule* va utiliser un "sous-processus" *compteur modulo N*, avec $N = 60$ pour les secondes et les minutes, et $N = 24$ pour les heures. On distingue les tops sur lesquels le compteur doit être incrémenté (chaque seconde, chaque minute, chaque heure), des instants où l'on demande de connaitre sa valeur (chaque seconde pour tous).

En plus de la valeur du compteur, le processus signale les instants de retour à zéro. C'est cette impulsion qui permet d'incrémenter le compteur suivant.

```
7:   (| (SECONDE, NOUV_MINUTE) := CPT_MOD {60} (TOPS, TOPS)
8:    | (MINUTE,  NOUV_HEURE ) := CPT_MOD {60} (TOPS, NOUV_MINUTE)
9:    | (HEURE,   NOUV_JOUR  ) := CPT_MOD {24} (TOPS, NOUV_HEURE)
10:   | ZJOUR := JOUR $
11:   | JOUR := ZJOUR + 1 when NOUV_JOUR default ZJOUR
12:   | JOUR ^= TOPS
13:   |)
14: where
15:   event NOUV_MINUTE, NOUV_HEURE, NOUV_JOUR;
16:   integer ZJOUR init 1
```

Le sous-processus est déclaré avec les objets locaux. Un mécanisme de compilation séparée sera installé dans une version ultérieure.

```
18:   process CPT_MOD =  % compteur modulo N %
19:     { integer N }
```

```
20:   ( ? event TOP_SORTIE,   % horloge de sortie de la valeur du compteur %
21:             TOP_INCR      % incrementation du compteur %
22:     ! integer CPT;        % le compteur %
23:       event RAZ           % lorsque le compteur revient a 0 %
24:   )
25:  (| CPT ^= TOP_SORTIE ^+ TOP_INCR
```

Le $\hat{}+$ *TOP_INCR* est ici facultatif : le contexte d'appel montre que les instants de TOP_INCR sont inclus dans ceux de TOP_SORTIE. Il serait obligatoire si l'on compilait le processus isolément.

```
26:   | ZCPT := CPT $
27:   | CPT := (ZCPT+1) modulo N when TOP_INCR
28:                            default ZCPT
29:   | RAZ := when CPT = 0 when TOP_INCR
30:   |)
31:  where
32:    integer ZCPT init 0
33:  end % CPT_MOD %
34: end
```

# Chapitre 5

# Applications

On traite ici un certain nombre d'exemples, en n'utilisant que les types simples du langage SIGNAL, et ses opérateurs de base.

## 5.1 Intervalle entre deux événements.

Soient DEBUT et FIN deux événements, FIN étant postérieur à DEBUT.

### 5.1.1 Durée séparant DEBUT et FIN

Cette DUREE est le nombre de tops d'une horloge H émis entre DEBUT et FIN. Soit N un signal regroupant toutes les horloges, et dont la valeur, mise à zéro sur DEBUT, compte les tops de H.

```
| N ^= DEBUT ^+ FIN ^+ H
| N  := 0 when DEBUT default N$ + 1
| DUREE := N when FIN
```

Si l'on s'intéresse à la précision de la mesure, on remarque que DUREE est 1 + le nombre de tops de H entre DEBUT et FIN, bornes exclues. Si ces bornes sont sur un top de H, la mesure est exacte. Sinon, l'erreur est toujours par excès ; elle peut atteindre +2 lorsque DEBUT est juste avant un H, et FIN juste après.

### 5.1.2 Présence entre DEBUT et FIN

On peut vouloir simplement tester si un événement donné se situe ou non dans l'intervalle entre DEBUT et FIN. Soit H l'événement, et DANS le résultat booléen associé à H.

```
| S ^= DEBUT ^+ FIN ^+ H
| S  := DEBUT default not FIN default (S$ init false)
        % true when DEBUT default false when FIN default S$ %
| DANS := S when H
```

Qu'en est-il pour les H se situant exactement sur DEBUT ou sur FIN ?

Avec la définition simple ci-dessus, un H sur DEBUT est *dans* l'intervalle, un H sur FIN est *en dehors* : intervalle $[DEBUT..FIN[$.

**Exercice :** Programmer les trois autres cas, correspondant à un intervalle ouvert à gauche et/ou fermé à droite.

**Remarque :** Une extension récente, *SIGNAL GTi*[10], ajoute les intervalles parmi les types de base du langage, avec des opérateurs associés.

## 5.2 Automates.

Les *automates d'états finis* sont fréquemment utilisés pour modéliser des systèmes temps-réel. Un automate comporte un certain nombre d'états $S_i$ , dont un état initial $S_0$ . Les occurrences d'événements $e_i$ provoquent des changements d'états, ou un bouclage sur le même état. Ces transitions peuvent être accompagnées de l'exécution d'actions appropriées.

En SIGNAL, une variable S désigne l'état d'arrivée d'une transition, sa valeur retardée ZS l'état de départ. Les actions sont des changements de valeurs de variables, déclenchés soit lors du passage d'un état à un autre, soit sur occurrence d'événements. Attention à la validité des horloges!

```
  { ? ... A, B  % signaux d'entrée, pouvant provoquer des changements d'état %
    ! integer X % sorties de l'automate %
  }
             % Changement d'etats %
  | S ^= A ^+ B ^+ ...
  | S := (2 when ^A default 3 when ^B) when ZS = 1
      default (3 when ^B) when ZS = 2
      default ...
      default ZS  % bouclage sur état courant %
  | ZS := S $
       % Actions sur transitions %
  | X := 0 when ZS = 1 and S = 2
         default ZX+1 when ^B
         default ZX
  | ...
where integer S, ZS init 1
```

**Exercices.**

1. Programmer l'éclairage d'un couloir, pouvant être allumé ou éteint manuellement par le même *Bouton*. De plus, un détecteur de *Présence* émet chaque seconde un booléen, à *vrai* lorsqu'il y a quelqu'un. L'automate doit éteindre si c'est allumé, et qu'il n'y a personne pendant 10 secondes.
2. La modélisation d'un *chronomètre* constitue un exemple classique d'automate.
   Considérer d'abord seulement 2 boutons : **MA** (Marche/Arrêt) pour lancer ou arrêter le décompte du temps, et **TR** (Temps intermédiaire/Remise à zéro) qui bloque l'affichage sans arrêter le compteur, jusqu'à une nouvelle action sur TR qui affiche à nouveau le compteur courant. Même lorsque l'affichage est bloqué, le compteur peut être arrêté ou relancé indépendamment par MA. Si le compteur est arrêté et que l'affichage est libre, TR remet le décompte à zéro.

Construire un signal *État Courant* en fonction des actions sur *MA* ou *TR*.

3. Ajouter une entrée *H* de type *event* représentant l'unité de temps - supposée assez fine. Le compteur et l'affichage sont des entiers exprimant un nombre d'unités H. Construire ces deux signaux.

## 5.3 Synchronisation des signaux d'entrée. Sur-échantillonnage

Lorsqu'un processus a plusieurs entrées, celles-ci peuvent être indépendantes les unes des autres, et n'avoir entre elles aucune relation temporelle. C'était le cas du processus *VEILLEUR*, où chaque entrée avait sa propre horloge indépendante.

Au contraire, certains processus peuvent spécifier des contraintes temporelles entre leurs entrées ; par exemple, préciser qu'elles sont synchrones :

```
( ? integer A, B
   ...
   | A ^= B
```

Dans ce cas, seules les **valeurs** de A et B seront à fournir; les deux suites auront en principe la même longueur (sinon, l'exécution s'arrêtera à la fin de la plus courte). Elles seront "consommées" en parallèle, à la même vitesse.

```
( ? integer X, Y
   ...
   | X ^= when Y = 0
```

Ici aussi, les horloges des entrées sont contraintes par le processus, et n'ont donc pas à être fournies. L'horloge de Y est maîtresse : une valeur de Y est lue à chaque instant du programme. Par contre, une nouvelle valeur n'est lue sur le *port* X que lorsque la valeur de Y est nulle.

Une entrée peut même être synchronisée sur certaines valeurs d'un signal **local**:

```
( ? integer X
   ...
   | S := ... S$ ...
   | X ^= when cond(S)
```

L'horloge de X est donc *extraite* de celle de S ($X \hat{} < S$). L'horloge la plus rapide d'un programme n'est pas nécessairement celle d'une entrée !

Plus généralement, si l'on prend comme base un signal A, on peut "créer des instants" intermédiaires entre deux occurrences de A, en synchronisant A sur une condition portant sur un signal local plus rapide. Par opposition à l'échantillonnage d'un signal, qui consiste à en extraire certaines valeurs (opérateur *when*), on parle ici de *sur-échantillonnage*.

Par exemple, pour définir un signal S deux fois plus rapide que A, avec un instant de S sur A et un autre entre chaque occurrence de A :

```
| BASCULE := NOT (BASCULE $ init false)
| S ^= BASCULE
| A ^= when BASCULE
```

Deux signaux locaux définis indépendamment l'un de l'autre n'ont aucune raison d'être synchrones si on ne le précise pas. Donc, si l'on programme deux sur-échantillonnages

distincts de A, le système considère qu'il s'agit de deux contraintes imposées à l'horloge de A, et il devra pouvoir prouver qu'elles sont compatibles.

**Exercices**

1. Soit N un signal d'entrée entier, compris entre 0 et 99. On souhaite produire un signal CHIFFRE, entier entre 0 et 9, qui pour chaque N, vaut d'abord le chiffre des dizaines, puis celui des unités :
   $N = 35, 7, 10, ..$ => $CHIFFRE = 3, 5, 0, 7, 1, 0, ..$
   Attention à l'horloge des unités, surtout si on les calcule par $N - 10 * CHIFFRE$ !
2. Soit N un signal entier, positif ou nul. Produire "à la suite" de chaque occurrence de N un nombre N de signaux de type *event*. Le premier *event* produit n'est pas forcément "sur" N.
3. Écrire un *mélangeur* à deux entrées X1 et X2, une seule sortie X égale à X1 ou X2, et qui retarde l'une des entrées lorsqu'elles arrivent simultanément, de façon à n'en perdre aucune.

## 5.4 Synchronisation sur l'horloge des sorties

On a souvent remarqué que l'horloge d'un signal peut être fixée séparément, indépendamment de son équation de définition. L'horloge d'un signal de sortie peut même être fixée à l'extérieur du processus.

Dans l'exemple suivant, un "capteur" prélève la valeur la plus récente de A, et la rend dans S :

```
process CAPTEUR =
  ( ? integer A
    ! integer S )
(| S := (A cell ^S) when ^S |)
```

Il n'y a aucune relation entre l'horloge de A et celle de S. D'autre part, cette horloge de S n'est pas fixée par son équation de définition... mais le compilateur vérifie que cette définition est bien compatible avec l'horloge de S. Les horloges relatives de A et de S doivent donc être fixées depuis l'extérieur du processus (fichiers RC_A.dat et RC_S.dat en cas d'exécution isolée).

# Chapitre 6

# La compilation de SIGNAL

L'intérêt d'un langage évolué pour la programmation "temps réel" est de permettre une vérification *dès la compilation* de la complétude et de la cohérence des définitions d'horloges intervenant dans le programme.

En plus des problèmes classiques d'analyse syntaxique, compilation, production de code, la mise en œuvre de SIGNAL extrait les horloges des signaux, vérifie leur cohérence, établit leurs relations (inclusions en particulier), détermine l'ordre des calculs à partir des relations de dépendance [7].

Les fichiers intermédiaires produits, ainsi que le code objet en langage C, facilitent la compréhension du mécanisme de la compilation, ainsi que l'identification des erreurs.

## 6.1 Analyse du programme

Les erreurs de *syntaxe* sont citées à l'écran.

Les erreurs *contextuelles* (non-déclarations, erreurs de type,...) apparaissent sous forme d'annotations dans un fichier *NOMPROG_LIS.SIG*

Les constructions qui n'appartiennent pas au langage de base - telles que le **cell** - sont ensuite remplacées par leur définition (réduction au langage noyau).

Les appels de processus locaux sont développés et remplacés par le corps de ces processus, avec substitution des paramètres, des signaux d'entrée et de sortie, et renommage des variables locales. En effet, l'analyse et le fonctionnement d'un processus peuvent être assez différents d'une occurrence à l'autre. En particulier, un processus peut n'être correct que si certaines propriétés de synchronisme sont vérifiées sur ses signaux d'entrée. Compilé isolément, un tel processus serait rejeté, alors que dans ses diverses utilisations, les signaux effectifs peuvent vérifier ces propriétés.

## 6.2 Équations d'horloges. Cas simple endochrone

Il s'agit ensuite de déterminer l'ensemble des horloges intervenant dans le programme. Les signaux ayant la même horloge doivent être regroupés, puisque les calculs qui les concernent ont lieu aux mêmes instants. De plus, ces classes d'horloges équivalentes doivent être hiérarchisées, sous forme d'un arbre d'inclusions : si une horloge h(S1) est extraite

d'une autre horloge h(S2), par exemple par un $S1 := when\ cond(S2)$, la condition ne sera évaluée et testée que lorsque S2 est présent.

Il est surtout important de déterminer si la hiérarchie des horloges a une racine unique, c'est-à-dire s'il existe une horloge plus rapide que toutes les autres, et les incluant toutes. Si elle existe, c'est cette horloge qui contrôlera la boucle principale du programme, toutes les actions ayant lieu sur les tops de cette horloge - ou sur certains de ses tops.

On dit alors que le programme est **endochrone**.

**Exemple :** considérons le processus suivant, qui compte les valeurs nulles d'un signal.

```
process ENDO =  % processus endochrone %
  ( ? integer A
    ! integer CPT )
(| CPT ^= when A = 0
 | CPT := ZCPT + 1
 | ZCPT := CPT$ init 0
 |)
where
  integer ZCPT;
end
```

CPT et ZCPT ont la même horloge, extraite de celle de A. La hiérarchie des horloges peut être représentée ainsi:

```
                ^A
                 |
                 |
        [A=0] = ^CPT = ^ZCPT
```

où [ *cond* ] est l'horloge des instants où la condition *cond* est vraie. Lorsque plusieurs signaux ont la même horloge, on ne montre le plus souvent qu'un seul représentant de la classe.

La hiérarchie a donc une racine unique : l'horloge de A, qui est *maîtresse*. L'horloge de la condition en est extraite, et les calculs sur CPT et ZCPT sont regroupés à l'horloge de cette condition.

## 6.3 Le graphe du programme

La hiérarchie des horloges constitue l'ossature d'un "graphe" plus complet, que nous détaillerons plus loin. Le système peut visualiser ce graphe sous forme d'un programme en "pseudo-SIGNAL", de nom *NOMPROG_TRA.SIG*. Ce programme permet une lecture "en clair" (?) du résultat de la compilation, et peut aider à la recherche d'erreurs de synchronisation. On donne ci-dessous un extrait commenté du ENDO_TRA :

```
process ENDO_TRA=
     ( ? integer A
       ! integer CPT
     )
```

```
   (| CLK_A := ^A   % l'horloge de A %
    | CLK_A ^= A
    | ACT_CLK_A{}
      % un sous-processus avec les Actions à l'horloge de A %
    |)
   where
      event CLK_A;

      process ACT_CLK_A =
         (| CLK_CPT := when (A=0)   % l'horloge de la condition %
          | CLK_CPT ^= CPT     % CPT représente les signaux à cette horloge %
          | ACT_CLK_CPT{}
            % les Actions à l'horloge de la condition %
          |)
      where
         event CLK_CPT;
         process ACT_CLK_CPT =   % sous-processus de ACT_CLK_A, %
                         % pour ce qui est à l'horloge de la condition %
         (| CLK_CPT ^= ZCPT     % ajout des autres horloges équivalentes %
          | (| CPT := ZCPT+1  % Les actions à l'horloge du processus %
             | ZCPT := CPT$1 init 0
             |)
          |)
         where
            integer ZCPT
         end %ACT_CLK_CPT%
      end %ACT_CLK_A%
   end %ENDO_TRA%
```

## 6.4 Le code produit

Le résultat de la génération de code séquentiel est un programme en C, dont nous donnons un extrait pour l'exemple ci-dessus :

```
/*  ==> input signals     */
static int A;
/*  ==> output signals    */
static int CPT;
/*  ==> local signals     */
static int ZCPT;
static logical C_CPT;
            /* les horloges sont traduites par des booléens (logical) */

EXTERN logical ENDO_initialize()
{ ZCPT = 0;
  return TRUE;}
```

```
EXTERN logical ENDO_iterate()
{
    if (!r_ENDO_A(&A)) return FALSE;
       /* Lecture de A, rend faux si fin de fichier */
       /* r_ENDO_A est défini dans un module Entrées/Sorties,
         pouvant être adapté à l'environnement : fichiers, capteurs,.. */
    C_CPT = A == 0;
    if (C_CPT)
        {CPT = ZCPT + 1;
         w_ENDO_CPT(CPT);
       /* sortie de CPT (par exemple sur un fichier WCPT.dat) */
        }
    ENDO_STEP_finalize();
    /* préparation des valeurs pour l'instant suivant */
    return TRUE;}

EXTERN logical ENDO_STEP_finalize()
{   if (C_CPT)
        {ZCPT = CPT;}
    return TRUE;}
```

La procédure *ENDO_initialize()* est appelée une fois en début d'exécution; puis l'exécutif boucle sur *ENDO_iterate()* tant qu'elle rend *vrai*, c'est-à-dire jusqu'à la fin des entrées.

## 6.5 Processus exochrone

Dans le processus ci-dessous, l'horloge de la sortie regroupe certains instants de l'entrée A - ceux pour lesquels A vaut 1 - ainsi que tous les instants de B. Le processus n'établit aucune relation entre les horloges de A et de B.

```
process EXO = % processus exochrone %
( ? integer A, B
  ! integer S )
(| S := 0 when A = 1 default B + 2 |)
```

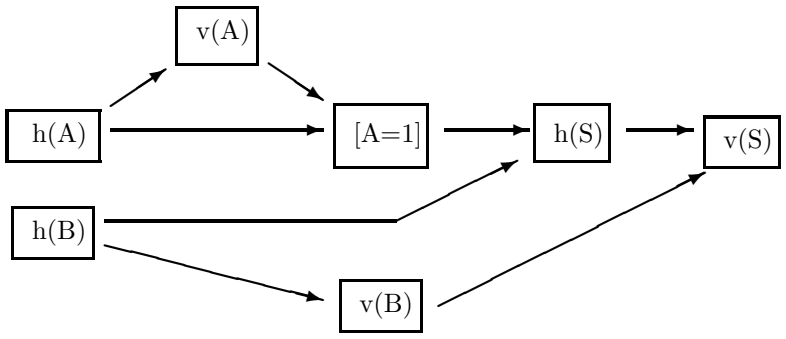
Seule, l'horloge de la condition $A = 1$ est extraite de l'horloge de A. D'où une "hiérarchie" sans racine commune.

```
  ^A       ^B       ^S
   |
   |
 [A=1]
```

L'horloge de S, qui dépend à la fois de l'horloge de la condition et de celle de B, est placée au niveau de l'horloge la plus "élevée", c'est-à-dire ^B.

À cette hiérarchie doit s'ajouter le *graphe des dépendances*, élaboré par le compilateur : il concerne à la fois les horloges et les valeurs des signaux, sachant que la valeur d'un signal

est toujours dépendante de son horloge, puisqu'un signal n'a de valeur qu'aux instants où il existe !




Les horloges relatives de A et B restent donc libres, et devront être fixées de l'extérieur : le processus est dit *exochrone*.

Un *tri topologique* est effectué sur le graphe, pour en extraire un ordre de calcul des différents signaux. Joint à la hiérarchie des horloges, il permet la construction du *graphe aux dépendances conditionnées*, traduit par le texte du *EXO_TRA.SIG*:

```
process EXO_TRA=
     ( ? integer A, B;
       ! integer S)

   (| (| CLK_A := ^A
       | CLK_A ^= A
       | ACT_CLK_A{}
       |)
    | (| CLK_B := ^B % horloge de B, au même niveau que ^A %
       | CLK_B ^= B
       |)
    | (| CLK := CLK_B ^- CLK_6 |)  % horloge du second terme du default %
    | (| CLK_S := CLK_6 ^+ CLK_B   % ^S = union de la condition et de ^B %
       | CLK_S ^= S
       | S := (0 when CLK_6) default ((B+2) when CLK)
       |)
    |)
   where
      event CLK_S, CLK, CLK_B, CLK_6, CLK_A;
      process ACT_CLK_A=    % le sous-processus correspondant à la condition %
      (| CLK_6 := when (A=1) |)
      end %ACT_CLK_A%        % pas d'action particulière à cette horloge %
   end %EXO_TRA%
```

Bien que l'on utilise les séparateurs d'instructions de SIGNAL, qui n'imposent pas d'ordre de calcul, les instructions apparaissent en fait dans l'ordre issu du tri; cet ordre sera exploité par le générateur de code objet séquentiel.

Remarquer dans la définition de $S$ que, pour des raisons de normalisation, chaque terme du *default* est présent avec son horloge exacte. Ces horloges sont toutes exclusives. Cela permettrait en cas d'exécution parallèle de répartir les calculs des diverses branches, en les faisant précéder de leur condition exacte d'exécution. Cette possibilité n'a pas d'intérêt en code séquentiel, où un simple *else* est produit dans le programme C suivant :

```
EXTERN logical EXO_iterate()
{
    if (!r_EXO_C_A(&C_A)) return FALSE;
             /* D'abord lecture de l'horloge de A ... */
    if (!r_EXO_C_B(&C_B)) return FALSE;  /* ... puis de celle de B   */
    if (C_A)
        {   /*  Si A est présent, ... */
        if (!r_EXO_A(&A)) return FALSE;  /* ... sa valeur est lue, ... */
        C_31 = A == 1;                /* ... et la condition calculée */
        }
    C_34 = (C_A ? C_31 : FALSE);
     /* C_34: booléen toujours présent, horloge du premier terme du default */
     /* C_31: horloge de la condition, n'est définie que si A est présent */
    C_S = C_B || C_34;          /* Horloge de S */
    if (C_B)                   /* Si B présent, ... */
        {if (!r_EXO_B(&B)) return FALSE;}
                              /*.. il est lu, qu'il soit utile ou non */
    if (C_S)                 /* Si S existe, ...    */
        {if (C_34) S = 0; else S = B + 2;  /* ... il est calculé... */
        w_EXO_S(S);           /* ... et sorti */
        }
    EXO_STEP_finalize();
    return TRUE;}
```

## 6.6 Les diagnostics du compilateur

On ne s'intéresse qu'aux messages d'erreur spécifiques au langage SIGNAL :

### 6.6.1 Input XX not declared in the process interface

Produit (dans le *NOMPROG_LIS.SIG*) si un signal XX non déclaré apparaît en partie droite d'expression. On l'obtient aussi sur $XX := XX + 1$ , même si XX est déclaré en local, ou comme sortie : l'analyse des dépendances considère que le XX déclaré est celui de la partie gauche ; celui de droite est alors... une entrée manquante !

### 6.6.2 Clocks Constraints

Signale l'échec du calcul d'horloges à démontrer la cohérence du système. Plus précisément, cette cohérence ne pourrait être démontrée que si les horloges de certains signaux possédaient des propriétés (contraintes)... qu'elles n'ont pas dans le contexte actuel.

```
process TEST =
```

```
 ( ? boolean A, B
   ! integer S )
(| X := 1 when A
 | Y := 2 when B
 | S := X + Y
 |)
```

Le "+" contraint X et Y à être synchrones; or, ils ne pourraient l'être que si les booléens A et B avaient leurs instants *vrai* exactement en même temps. L'erreur est signalée dans le fichier *TEST_TRA.SIG*, dont on donne un extrait :

```
......
 (| CLK_S ^= CLK_11 |)%**war: Clocks constraints%
......
   (| CLK_S := when A
......
   (| CLK_11 := when B
```

Le message indique les horloges dont on n'arrive pas à montrer l'égalité.

**Exercice :** Le programme suivant lit des suites consécutives d'entiers positifs, chaque suite étant terminée par un 0. On ne s'intéresse qu'à la **longueur** des suites. On veut avoir la longueur de la suite la plus longue, chaque fois qu'un nouveau maximum est trouvé.

```
process MAXLONG =
       ( ? integer A     % Suites d'entiers, terminees par 0 %
         ! integer MAX ) % Maxima successifs des longueurs des suites %
(| FIN := when A = 0
 | LG ^= A
 | LG := 0 when FIN default ZLG+1
 | ZLG := LG$ init 0
 | TAILLE := ZLG when FIN
 | MAX := TAILLE when TAILLE > ZMAX
 | ZMAX := MAX $ init 0
 |)
where
 event FIN;
 integer LG, ZLG, TAILLE, ZMAX;
end
```

Le programme ci-dessus produit une *Clocks constraints*. Trouver pourquoi !

On donne des extraits du MAXLONG_TRA.SIG :

```
  (| CLK_MAX ^= CLK_18 |) %**war: Clocks constraints
  (| CLK_MAX := when (A=0)
   | CLK_MAX ^= MAX
  (| CLK_MAX ^= TAILLE ^= ZMAX
  (| CLK_18 := when (TAILLE>ZMAX)
```

### 6.6.3 Dependency cycle in the graph

Ce message apparaît après la phase de hiérarchisation des horloges, lorsque le graphe de dépendance des signaux est parcouru pour établir leur équation de définition. Lorsque ce graphe comporte un cycle, l'erreur est signalée, et le détail des relations entre signaux participant au cycle est produit dans un fichier *NOMPROG_CYC.SIG*. Par exemple,

```
   (| S := X + 1
    | X := A + Y
    | Y := 2 * X
    |)
donne
   (| X --> Y  | Y --> X  |)
```

Les cycles concernent fréquemment les horloges, dont le calcul est plus difficile à maitriser.

### 6.6.4 Autres erreurs d'horloge

– La plus courante est le classique :

```
    | CPT := (CPT$ init 0) + 1 when ^A
```

qui tente de numéroter les occurrences de A.
On obtient un message *Clocks constraints*, avec dans le ..TRA :

```
     CLK_CPT := CLK_A ^* CLK_CPT
```

Il s'agit en fait d'une définition *récursive* de l'horloge *CLK_CPT*, que l'on trouve à gauche et à droite du := L'équation d'horloge qui en résulte ne serait vérifiée que si l'horloge de CPT était *incluse* dans celle de A. Le compilateur ne sait pas déduire cela, et ce n'est sans doute pas ce que souhaitait le programmeur !
Cette erreur n'est pas actuellement notée dans le ...TRA.

– Dans le problème suivant, la sortie *ETAT* peut prendre les valeurs 1 ou 2. Les signaux d'entrée $TOP_i$ tentent de faire ETAT à la valeur $i$ s'il ne l'a pas déjà :

```
process ...
  ( ? event TOP1, TOP2
    ! integer ETAT )
 (| CHANGE1 := TOP1 when ZETAT /= 1
  | CHANGE2 := TOP2 when ZETAT /= 2
  | ETAT := 1 when CHANGE1 default 2 when CHANGE2
  | ZETAT := ETAT $1
  |)
```

C'est l'horloge de ETAT qui n'est pas claire, puisque sa définition ne le fait exister que lorsqu'il change, alors qu'on en teste la valeur précédente sur chaque TOP. On peut corriger en le faisant exister sur chaque TOP :

```
  | ETAT_BIS ^= TOP1 ^+ TOP2
  | ETAT_BIS := 1 when CHANGE1 default 2 when CHANGE2 default ZETAT_BIS
  | ETAT := ETAT_BIS when (CHANGE1 default CHANGE2)
```

ce qui supprime l'erreur. La 3ème ligne permet de limiter les sorties aux valeurs modifiées.

– On retrouve également une erreur dans un cas comme :

```
| S := (A default 0) when B
```

où la parenthèse, contenant une constante, a donc son horloge insuffisamment définie, et se voit imposer l'horloge de B : cela peut provoquer incompatibilité ou cycle avec l'horloge de A.

**Exercice :** pour corriger l'erreur "Clocks constraints" du processus *MAXLONG* de la section précédente, on introduit un signal MEMAX, de même horloge que TAILLE. MEMAX est égal à ZMAX lorsque MAX prend une nouvelle valeur, puis prolonge le nouveau MAX sur toutes les fins de suites. On tient à ce que MAX ne soit présent que lorsqu'il prend une nouvelle valeur.

```
| TAILLE := ZLG when FIN
| MEMAX := ZMAX default (MAX cell FIN)
| MAX := TAILLE when TAILLE > MEMAX
| ZMAX := MAX $ init 0
```

Le compilateur détecte plusieurs circularités, concernant MEMAX et les horloges et valeurs de MAX et ZMAX ; une valeur ne peut être élaborée qu'après avoir déterminé qu'elle était présente.

Analyser cette anomalie. Proposer une version enfin correcte de MAXLONG.

Établir la hiérarche des horloges de ce processus.

# Chapitre 7

# Calcul de propriétés

L'hypothèse de synchronisme rend les programmes SIGNAL déterministes - ce qui ne serait pas le cas de programmes dans un langage asynchrone. Par exemple, lorsque deux signaux arrivent simultanément, le comportement d'un programme SIGNAL est parfaitement défini : si l'on a écrit *A default B*, c'est la valeur de A qui est prise. En Ada, si plusieurs rendez-vous sont possibles à un instant, l'un d'eux est choisi aléatoirement, et le programme n'est donc pas déterministe.

Il est alors envisageable en SIGNAL d'effectuer des calculs sur le comportement d'un programme, de prévoir son évolution. On peut en particulier prouver certaines propriétés, en démontrant qu'elles seront vérifiées quels que soient les signaux d'entrée.

La complexité des calculs croissant très vite avec la taille du programme, un logiciel spécialisé a pû être écrit pour conduire ces preuves : SIGALI [8][9]. On démontre par exemple qu'un *contrôleur d'exclusion mutuelle*, programmé en SIGNAL, ne laisse jamais entrer deux processus en section critique.

On donne ci-dessous un aperçu de la manière dont ces calculs peuvent être conduits.

## 7.1 Codage des horloges et valeurs de signaux

On choisit de coder chaque signal **A** par une variable **a**, de type *entier*. Ces variables sont définies à tous les instants "intéressants" du programme, c'est-à dire lorsque au moins un signal est présent. A chacun de ces instants, le signal A est soit absent, soit présent avec une certaine valeur. La variable $a$ va coder l'absence par **0**, et la présence par **1** ou **-1**.

On ne code actuellement la *valeur* des signaux que pour les *booléens* (et bien sûr *event*). Les signaux booléens conditionnent à travers l'utilisation du *when* la présence des autres signaux. On décide alors de coder un signal booléen présent par **+1** lorsqu'il est **vrai**, par **-1** lorsqu'il est **faux** - et toujours **0** s'il est **absent**.

Ainsi, une opération comme $B := not\ A$ pourra être codée par un simple changement de signe : $b = -a$ . Les autres opérations donneront des calculs plus complexes, mais le domaine du résultat devra rester limité à ces 3 valeurs -1, 0, +1. Aussi considère-t-on que les calculs s'effectuent dans le corps des **entiers modulo 3**, noté **Z/3Z**.

Cet ensemble possède des propriétés intéressantes :
$a + a = -a,\ a + a + a = 0,\ a^{2n} = a^2,\ a^{2n+1} = a,$

$a.(1-a^2)=0$, $(f(a^2))^n = f(a^2)$, en particulier : $(1-a^2)^2 = (1-a^2)$, etc.

## 7.2 Codage des opérations sur les signaux

On a vu que l'opérateur *not* pouvait être codé par un simple changement de signe.

Pour *a and b*, on trouve l'expression $ab(ab-a-b-1)$ : elle est bien égale à 0 (absence) si $a$ et $b$ sont nuls (absents) ; si $a=b=1$ (vrais), l'expression vaut -2, soit 1 dans Z/3Z ; elle vaut -1 si l'un au moins des opérandes est -1.

Donnons la table des divers opérateurs, où les opérandes sont des booléens:

| | |
|---|---|
| not a | $-a$ |
| a and b | $ab(ab-a-b-1)$ |
| a or b | $ab(1-a-b-ab)$ |
| when b | $-b-b^2$ |
| a when b | $a(-b-b^2)$ |
| a default b | $a+(1-a^2)b$ |
| a $ 1 init x0 | $a^2x$, avec $x' = a+(1-a^2)x$ |

Dans le codage du retard, $x$ est une variable *d'état*, présente sur tous les signaux : $x$ est sa valeur à un instant donné $t$, $x'$ la valeur qu'elle prendra à l'instant suivant $t+1$, valeur que l'on détermine à l'instant $t$. La valeur initiale de $x$ est le $x_0$ du *init*.

L'expression de définition $a^2x$ montre que, à l'instant $t$:
- si $a$ est absent (nul), le signal retardé est également absent ; la prochaine valeur $x'$ reste égale à $x$
- si $a$ est présent ($a^2=1$), *a $ 1* prend la valeur actuelle de $x$, alors que le $x$ suivant prendra la valeur actuelle de $a$.

Si les signaux ne sont pas booléens, on ne code plus les valeurs mais seulement les horloges :

| | |
|---|---|
| c := a when b | $c^2 = a^2(-b-b^2)$ |
| c := a default b | $c^2 = a^2+(1-a^2)b^2$ |
| a ^= b | $a^2 = b^2$ |
| a ^* b | $a^2b^2$ |
| a ^+ b | $a^2+(1-a^2)b^2$ |

Par ailleurs, les opérateurs synchrones (*and, or*, mais aussi $+, -$, etc) imposent l'égalité d'horloge de leurs opérandes. Cela s'exprime en ajoutant des contraintes de la forme : $a^2 = b^2$.

Enfin, à une définition de signaux $a := b$ correspond l'égalité $a = b$ s'il s'agit de booléens, ou $a^2 = b^2$ sinon.

### Exercices

1. Coder en Z/3Z l'opération : $B := \hat{}A$
2. A, B, C étant des booléens, coder en Z/3Z l'opération : $C := A \neq B$ (ou exclusif)

## 7.3 Exemple : l'opérateur cell

Cet opérateur : $C := A$ *cell* $B$ *init x0* propage la valeur de A sur les valeurs *vrai* de B. Il est défini par :

```
| C ^= A ^+ when B
| C := A default (C$ init x0)
```

On suppose d'abord les signaux A et C booléens (B l'est toujours). Appliquons les définitions pour évaluer la variable associée à C :
$c^2 = a^2 + (1 - a^2)(-b - b^2)^2$
$c = a + (1 - a^2)c^2x$
$x' = c + (1 - c^2)x$

Dans le développement de la première équation, on remarque que :
$(-b - b^2)^2 = (-b - b^2)$, car $(-b - b^2)$ vaut 1 pour b=1 et 0 autrement.

Remplaçant $c^2$ dans la seconde équation, on obtient : $c = a + (1 - a^2)(-b - b^2)x$.

Reportons $c$ et $c^2$ dans la troisième : $x' = a + (1 - a^2)x$.

La variable $x$ se contente donc de mémoriser $a$ lorsqu'il est présent. $c$ est bien égal à $a$ lorsqu'il est présent ($a^2 = 1$), et sinon, lorsque $a = 0$ et $b = 1$, il vaut $x$, soit la dernière valeur de $a$. Lorsque $a = 0$ et $b = 0$ ou $b = -1$, on a $c = 0$, donc absent. C'est bien la définition du *cell*.

Si A et C ne sont pas booléens, on ne peut plus raisonner que sur leurs horloges. D'après leur définition, C se code du point de vue horloges par :
$c^2 = a^2 + x^2 - a^2.x^2$ et $x^2 = c^2$, soit : $c^2 = a^2 + (1 - a^2).c^2$ .
En substituant le $c^2$ côté droit, on obtient :
$c^2 = a^2 + (1 - a^2).(a^2 + (1 - a^2).(-b - b^2))$
Comme $(1 - a^2).a^2 = 0$ et $(1 - a^2)^2 = 1 - a^2$ , on retrouve l'expression ci-dessus : $c^2 = a^2 + (1 - a^2)(-b - b^2)$. Les deux lignes de définition sont bien cohérentes.

## 7.4 Vérification de propriétés

Montrons sur un exemple comment peut se dérouler la vérification de certaines propriétés d'un processus SIGNAL, travaillant sur des booléens.

### 7.4.1 Spécification, et programme SIGNAL

Le processus a une entrée B de type booléen. Il produit une sortie C égale à B lorsque : soit B, soit sa valeur précédente B $, sont à *vrai*. La première valeur de B est toujours sortie.

Le code SIGNAL suivant se propose de mettre en oeuvre cette spécification :

```
  process NONFF =
   ( ? boolean B
     ! boolean C )
  (| C := B when D
   | D := B or P
   | P := B $ init true
   |)
where boolean D, P;
```

```
end
```

La propriété que l'on souhaite vérifier est :
*il n'y a jamais sortie de deux valeurs consécutives ayant la valeur faux.*

On rajoute dans le programme la valeur retardée de C :

```
| R := C $ init true
```

On ne doit jamais avoir C et R tous deux à *faux*. L'initialisation de R évite une succession *faux* - *faux* non significative si le premier B est *faux*.

### 7.4.2 Codage du programme

Le codage en Z/3Z du programme s'écrit :
$c = b(-d - d^2)$
$d = bp(1 - b - p - bp), d^2 = p^2 = b^2$
$p = b^2 x, x' = b + (1 - b^2)x, x_0 = 1$
$r = c^2 y, y' = c + (1 - c^2)y, y_0 = 1$

On note d'abord que si $b = 0$ (absent), toutes les variables sont à 0, sauf les variables d'état *x, y* qui conservent leur valeur. Ces instants où aucun signal n'est présent sont toujours supprimés des calculs.

Avec $b^2 = 1$, les 3 premières lignes se récrivent :
$p = x, x' = b, x_0 = 1$
$d = bx - x - b - 1$
$c = b(-d - 1)$, soit : $c = -x + bx + 1$

On vérifie immédiatement que si $b = 1$ (*vrai*), $c = 1$, c'est-à-dire présent et vrai.

Si $b = -1$ (*faux*), $c = x + 1$. Or, la *variable d'état* $x$ mémorise sur tous les instants la valeur précédente de $b$. Si $x = 1$ (B précédent *vrai*), $c = -1$, soit la valeur actuelle de B. Si $x = -1$ (deux B consécutifs *faux*), $c = 0$, c'est-à-dire *absent* : aucun C n'est produit.

### 7.4.3 Automate associé au programme

Une exécution SIGNAL est la succession des instants où au moins un signal est présent. À chacun de ces instants, et avant toute lecture d'un signal d'entrée, on dispose des *variables d'état x, y,...*, portant la valeur des signaux retardés, que ces signaux soient présents ou non à cet instant.

Ainsi, sur un B *faux* suivant un autre B *faux*, la variable $c$ va prendre la valeur 0, sa valeur retardée $r$ également, mais la variable d'état $y$ associée à $r$ a une valeur non nulle (sans doute -1) : c'est la dernière valeur prise par $C$, ou la valeur initiale $y_0$ si $C$ n'a encore pris aucune valeur.

Cet ensemble des valeurs (non nulles) des variables d'état constitue un *état* d'un automate associé au programme. L'ensemble $[x_0, y_0, ..]$ constitue l'état initial, disponible pour le premier instant du calcul.

Un état $[x, y, ...]$ étant fixé au début d'un instant du calcul, les entrées (telles que $b$) sont lues. Les signaux intermédiaires sont élaborés, dans un ordre établi par la compilation : $p = x$, puis $d$, puis la sortie $c$. Enfin, les nouvelles valeurs des variables d'état sont calculées :
$x' = b, y' = c + (1 - c^2)y$

$[x', y', ..]$ constituent le nouvel état vers lequel le système effectue une transition, étiquetée par la valeur de $b$. Cet état sera disponible au début de l'instant suivant.

Ce système de variables d'état, de transitions,... fait partie des *Systèmes Dynamiques Polynomiaux*, pour lesquels des outils mathématiques ont été développés [8].

Les propriétés vérifiables sur ces systèmes portent sur les états : quels états sont atteignables, le système est-il vivace (absence de blocage), etc.

### 7.4.4 La vérification

Notre programme doit être à nouveau adapté, pour que la propriété à vérifier (pas deux sorties consécutives à *faux*) soit représentable en terme d'états. La variable $y$ porte déjà le dernier $c$ produit. Il nous faut aussi l'avant dernier :

```
| R := C $ init true
| S := R $
```

avec une valeur initiale quelconque. Le codage Z/3Z s'écrit :
$r = c^2 y, y' = c + (1 - c^2)y, y_0 = 1$,
$s = r^2 z, z' = r + (1 - r^2)z$ et aussi $s^2 = r^2 = c^2$

Un état du système sera donc un triplet $[x, y, z]$, soit 8 états théoriquement possibles. L'état initial est $[1, 1, z_0]$. On demande de vérifier que les états de la forme $[x, -1, -1]$ sont inaccessibles quelles que soient les données.

Le calcul est faisable manuellement :
- on a vu que si $b = 1$, alors $c = 1$, et donc $y' = 1, z' = r = y$ ; comme par ailleurs $x' = b = 1$, l'état d'arrivée est $[1, 1, y]$, toujours correct quelque soit l'état de départ
- si $b = -1, c = x + 1$ ; il faut alors détailler $x$ :
- - si $x = -1, c = 0, y' = y, z' = z$ : on boucle sur l'état courant
- - si $x = 1, c = -1, y' = -1, z' = r = y$, soit l'état d'arrivée : $[-1, -1, y]$ ; cet état est incorrect si $y = -1$, et serait accessible en partant d'un état de la forme $[1, -1, z]$. Mais ni l'état initial, ni les états d'arrivée calculés ci-dessus ne comportent cette configuration.

Les états incorrects sont donc inaccessibles, et la *propriété vérifiée*.

**Remarque :**

on pourrait être tenté de vérifier directement la propriété, en ajoutant un signal :

```
| V := (not C) and (not R)
```

Ce signal $V$ devrait être toujours *faux* ou absent ($v \leq 0$). Son codage s'écrit :
$v = cr(cr + c + r - 1)$
Si $b = 1$, alors $c = 1, r = y, v = -y^2 = -1$
Si $b = -1$ et $x = -1$, alors $c = 0$ et $v = 0$
Mais si $b = -1$ et $x = 1$, alors $c = -1, r = y, v = -y$, et l'on ne peut pas conclure. Seule l'étude des états accessibles permet de montrer qu'avec ces hypothèses, on a toujours $y = 1$.

### 7.4.5 Exercices

1. Représenter les divers états réellement accessibles à partir de l'état initial, avec les transitions étiquetées par $b$. Il y a en a très peu !
   Modifier les initialisations pour réduire encore le nombre d'états accessibles.
2. Démontrer que la requête : *"le premier B est toujours sorti"* est bien vérifiée.

3. Modifier l'exemple traité ci-dessus pour ne sortir B que *lorsqu'il change*. Le premier B est également sorti. La propriété à vérifier est que *toute valeur sortie - à partir de la seconde - est la négation de la précédente*.

# Chapitre 8

# Grammaire d'un sous-ensemble de SIGNAL

Les non-terminaux sont entre chevrons $< .. >$ . Les noms en *o-xx* indiquent une entité optionnelle, les *l-yy* une liste. Les terminaux sont entre guillemets.

```
<programme> => <processus>

<processus> => "process"  <ident>  "="
        <o-decl-parametres>
        "(" "?" <o-l-declarations>
            "!" <o-l-decl-init>  ")"
        "(|" <l-equations> "|)"
        <o-decl-locales>

<o-decl-parametres> =>
        |  "{" <o-l-declarations> "}"
```

Les paramètres sont considérés dans le reste du processus comme des constantes.

```
<o-l-declarations> =>  | <declaration>
        |  <declaration>  ";"  <o-l-declarations>

<declaration>  =>  <type>  <l-ident>

<type>  =>  "event" | <type-init>

<type-init> => "boolean" | "integer" | "real" | "long" | ....

<l-ident> => <ident> | <ident> "," <l-ident>

<o-l-decl-init> =>   |  <decl-init>
        |  <decl-init>  ";" <o-l-decl-init>

<decl-init> => "event" <l-ident> | <type-init>  <l-ident-init>
```

Un *ident* de type *event* ne peut être initialisé.

```
<l-ident-init> => <ident> <o-initialisation>
        |  <ident> <o-initialisation>  ","  <l-ident-init>

<o-initialisation> =>
        |  "init"  <expression-constante>

<l-equations> => <equation>
        |  <equation>  "|"  <l-equations>

<equation>  =>  <contrainte-horloge>
        |   <ident>  ":="  <expression>
        |   <ident>  ":="  <appel-processus>
        |   "("  <l-ident>  ")"  ":="   <appel-processus>

<contrainte-horloge> =>
     <expression-horloge> "^=" <expression-horloge>
  |  <expression-horloge> "^=" <contrainte-horloge>

<expression-horloge> =>
     <expression> | <expression> <op-horloge> <expression-horloge>

<expression>  =>  <exp-1>
        |   <expression> "default" <exp-1>

<exp-1>  => <exp-2>
        |  <exp-1>  "when"  <exp-3>
        |  <exp-2>  "cell"  <exp-2>

<exp-2>  => <exp-3>
        |  "when" <exp-3>

<exp-3>  =>  <exp-4>
        |  "^"  <ident>

<exp-4>  =>  <exp-5>
        |   "not"  <exp-5>
        |   <exp-4>  "or"   <exp-4>
        |   <exp-4>  "and"  <exp-4>
```

Cette règle et la suivante sont laissées ambigües pour n'avoir pas à détailler les priorités de tous les opérateurs.

```
<exp-5>  => <exp-simple>
        |  "-" <exp-simple>
        |  <ident>  "$" <o-initialisation>
```

```
        |  <ident>  "$"  <exp-simple-constante> <o-initialisation>
        |  <exp-5> <op-binaire> <exp-5>

<op-binaire>  =>  "<" | "<=" | ">" | ">=" | "=" | "/="
        |   "+" | "-"  | "*" | "/"  | "**"

<op-horloge> => "^+" | "^*" | "^-"

<exp-simple> => <constante>
        |  <ident>
        |  <appel-fonction>
        |  "("  <expression>  ")"

<constante>  =>  "true" | "false" | nombre entier  |  nombre reel

<exp-constante>  =>  <exp-4>

<exp-simple-constante>  =>  <exp-simple>
```

Expressions constantes : sur constantes ou identificateurs de paramètres

```
<appel>  =>  <ident> <o-parametres> "("  <o-l-expressions> ")"

<appel-fonction>  =>  <ident>  "("  <l-expressions>  ")"

<o-l-expressions>  =>
        |  <l-expressions>

<l-expressions>  =>  <expression>
        |   <expression>   ","  <l-expressions>

<o-parametres>  =>
        | "{"  <l-parametres> "}"

<l-parametres>  =>  <exp-constante>
        |   <exp-constante>  ","  <l-parametres>

<o-decl-locales> =>
        |   "where" <o-l-decl-locales> "end"

<o-l-decl-locales> =>  <l-decl-init>
        | <l-decl-init> ";" <o-l-decl-locales>
        | <l-decl-init> ";" <o-l-decl-processus>
        | <o-l-decl-processus>

<o-l-decl-processus>  =>
        |   <processus>
```

```
        |   <fonction>
        |   <processus>  ";"  <o-l-decl-processus>
        |   <fonction>   ";"  <o-l-decl-processus>

<fonction>  =>  "function"  <ident>  "="
        "(" "?" <l-declarations>  "!"  <type>  <ident>  ")"
```

# Bibliographie

[1] Paul Le Guernic. *SIGNAL : description algébrique des flots de signaux*, dans Architecture des machines et systèmes informatiques, AFCET, 1982.

[2] Thierry Gautier. *Conception d'un langage flot de données pour le temps réel.* Thèse Université de Rennes 1, 1984.

[3] Patricia Bournai, Bruno Chéron, Thierry Gautier, Bernard Houssais, Paul Le Guernic. *SIGNAL Manual.* RR 1969, INRIA, 1993.

[4] Nicolas Halbwachs et al. *The synchronous data flow programming language LUSTRE.* Proceedings IEEE, 79-9, 1991.

[5] Boussinot, De Simone. *The ESTEREL language.* Proceedings IEEE, 79-9, 1991.

[6] Claude Le Maire. *Environnement de programmation synchrone en reconnaissance de la parole.* Thèse Université de Rennes 1, 1990.

[7] Loïc Besnard. *Compilation de SIGNAL : horloges, dépendances, environnement.* Thèse Université de Rennes 1, 1992.

[8] Michel Le Borgne. *Systèmes dynamiques polynomiaux sur des corps finis.* Thèse Université de Rennes 1, 1993.

[9] Bruno Dutertre. *Spécification et preuve de systèmes dynamiques.* Thèse Université de Rennes 1, 1992.

[10] Eric Rutten, Paul Le Guernic. *Sequencing Data Flow Tasks in SIGNAL.* PI 778 IRISA, 1993.

[11] Patricia Bournai, Paul Le Guernic. *Un environnement graphique pour le Langage SIGNAL.* PI 741 IRISA, 1993.

# Chapitre 9

# Corrigé des exercices

### Opérateurs synchrones

```
process APPAREIL =
  ( ? event BOUTON
    ! boolean ENMARCHE )
 (| BOUTON ^= ENMARCHE
  | ENMARCHE := not (ENMARCHE$ init false)
  |)

process MOYENNES =
  {integer N}
  ( ? real A
    ! real S, MOY, MOYN )

 (| S := (S$ init 0.0) + A      % synchronise S et A %

  | CPT := (CPT$ init 0) + 1
  | MOY := S / CPT   % synchronise CPT avec S, donc avec A %

  | SN := (SN$ init 0.0) - (A$N init [ {to N}: 0.0 ]) + A
  | MOYN := SN / N
  |)
 where
  integer CPT;
  real SN;
end

% PREMIER vrai sur 1ere occurrence de A, faux ensuite %
  | PREMIER := NHA$ init true
  | NHA := not (^A) % horloge de A, valeur toujours false %
 where
   boolean NHA;
```

### Opérateur when unaire

```
 (| ZA := A$ init 0
  | MONTE := A > ZA or (A = ZA and ZMONTE)
      % MONTE prend l'horloge de A %
      % maintien de la valeur de MONTE en cas de palier %
  | ZMONTE := MONTE$ init true
  | SOMMET := when (ZMONTE and not MONTE)
                % on montait, et l'on ne monte plus %
  |)
 where
  integer ZA;
  boolean MONTE, ZMONTE
end
```

### Opérateur when binaire

Exercice 1 : *true when ^A* se simplifie en *^A*.
*not (^A)* est toujours *false*, donc *when not (^A)* toujours absent.

Exercice 2 :

```
  | MAXLOCAL := A$ when (ZMONTE and not MONTE)
ou
  | MAXLOCAL := A$ when SOMMET
```

Exercice 3 :

```
  (| CPT ^= A
   | CPT := (CPT$ init 0) modulo N + 1
   | S := A when CPT = 1
   |)
 where
   integer CPT;
```

Exercice 4 :

```
 (| PREMB := B when (PREMIER or B /= B$)
  | PREMIER := NHB $ init true
  | NHB := false     % operateur or => horloge de B %
  |)
where
  boolean NHB, PREMIER;
```

Exercice 5 :

L'expression *^S1 and ^S2* est incorrecte, car elle impose que S1 et S2 aient la même horloge.
Solution: **^S1 when ^S2**

### Opérateur default

Exercice 2 :

```
B := true when condition default false
```

n'est valide que si l'horloge de B est fixée par ailleurs, et si le compilateur peut montrer qu'elle inclut (ou est égale à) l'horloge de la condition.
**Simplification:** *true* peut toujours être supprimé. On ne peut écrire $B := condition$ que si $B$ et *condition* ont exactement la même horloge.

Exercice 3 :

*3 when ^S1 when ^S2 default 1 when ^S1 default 2 when ^S2*

Exercice 4 :

```
| ZMAX := MAX$ init 0
| MAX := A when A > ZMAX default ZMAX
    % A > ZMAX suffit a‘ rendre A et ZMAX (et donc MAX) synchrones %
```

Exercice 5 :

```
| CPT ^= ENTREE default SORTIE
| ZCPT := CPT$ init 0
| CPT := ZCPT when ENTREE when SORTIE
         default ZCPT + 1 when ENTREE
         default ZCPT - 1 when SORTIE
```

La contrainte sur les horloges est bien nécessaire : l'horloge de l'expression en partie droite est égale à l'intersection des horloges de *ZCPT* et de *ENTREE default SORTIE*; elle n'aurait aucune raison d'être égale à celle de *CPT*. Le dernier *when SORTIE* est facultatif.

On peut aussi factoriser :

```
| CPT := (ZCPT when SORTIE default ZCPT + 1) when ENTREE
         default ZCPT - 1
```

Exercice 6 : soustraction d'event A - B

```
| NON_B_A := not B default A
| A_B := A when NON_B_A
```

Exercice 7 :

```
| ENSEMBLE := ^S1 when ^S2
| UNSEUL := when (not ENSEMBLE default ^S1 default ^S2)
```

*ENSEMBLE* est présent et *vrai* lorsque S1 et S2 sont simultanés. L'expression parenthèsée vaut donc *faux* dans ce cas, et *vrai* si l'un seulement des signaux est présent. Le *when* permet de n'extraire que ces instants *vrai*.

Exercice 8 :

```
process ALTERN =
   % produit alternativement une valeur de A et une valeur de B %
  ( ? integer A, B
    ! integer S )

(| NUM := (3 - (NUM$ init 2)) when ^A when ^B
      default 1 when ^A default 2 when ^B
 | S := (A when NUM = 1 default B) when NUM /= (NUM$ init 0)
 |)
where
  integer NUM
end
```

NUM associe à chaque occurrence de A ou/et B un numéro : 1 pour A seul, 2 pour B seul ; et s'ils sont présents tous deux, "l'inverse" du numéro précédent ; NUM est donc le numéro du signal d'entrée qui doit être effectivement produit... lorsqu'il est différent du précédent.

Il est permis d'initialiser NUM$ à des valeurs différentes selon le contexte : dans la définition de NUM, la valeur 2 donne NUM=1 si A et B sont tous deux présents au premier instant ; dans la définition de S, la valeur 0 permet au premier NUM d'être pris en compte qu'il soit 1 ou 2. Avec 3 valeurs, NUM ne pouvait donc être un booléen.

### Opérateur cell

Exercice 1 :

```
process BALAYAGE = % balayage d'une image ligne par ligne %
  {integer NL, NC} % dimensions de l'image %
 ( ? boolean PIXEL
   ! integer LIGNE, COLONNE)
}
 (| ZCOL := COLONNE$ init 0
  | COLONNE := ZCOL modulo NC + 1
  | COLONNE ^= PIXEL
  | ZL := L$ init 0
  | L := ZL modulo NL + 1
  | L ^= when COLONNE = 1
  | LIGNE := L cell ^PIXEL
  |)
 where
    integer ZL, ZCOL, L;
end
```

Exercice 2 : **Resynchronisation :**

Si **A** est exactement sur un **H**, il doit être sorti aussitôt ; sinon, il est déplacé sur le **H** qui le suit immédiatement.

On peut construire un **A cell H** que l'on restreint aux instants intéressants :

```
  AH := A when H
        default (A cell H) when PREMIERH
```

Le booléen PREMIERH ne doit être vrai que sur le premier H suivant un A isolé. Derrière A et H simultanés, il doit être faux, pour éviter la répétition du A.

```
 | PREMIERH := SURA $1 init false
 | SURA := not H default ^A
```

Une autre solution consiste à limiter la répétition du A au seul instant nécessaire : sur le H suivant un A isolé. C'est ce que ferait **A cell PREMIERH**, qui produit également tous les A. Pour éliminer les A isolés, on restreint ce signal aux seuls instants de H :

```
  AH := (A cell PREMIERH) when H
```

Cette expression montre clairement que AH n'est produit que sur des instants de H.

Exercice 3 : **Passage à niveau.**

Une seule voie :

```
FERMER := AVANT default not APRES
```

Si plusieurs impulsions par capteur :

```
CAPTEURS := AVANT default not APRES
FERMER := CAPTEURS when CAPTEURS /= (CAPTEURS$ init false)
```

Avec deux voies, soient F1 et F2 les signaux de fermeture en provenance de chacune des voies : *FERMER := F1 or F2* imposerait F1 et F2 synchrones. Leur regroupement par *F1 default F2* n'est pas plus acceptable, car en cas de simultanéité, le second signal est perdu : en particulier, si F1 autorise l'ouverture à l'instant où F2 demande la fermeture, la barrière serait effectivement ouverte !

Le *or* donne bien la solution logique, à condition d'amener les valeurs des signaux sur une horloge commune :

```
process BARRIERE = % passage a niveau sollicite par 2 voies %
  { ? boolean F1, F2  % VRAI : doit fermer; FAUX : peut ouvrir %
    ! boolean FERMER }  % VRAI : on ferme; FAUX : on ouvre %

 (| MEM1 := F1 cell ^F2 init false
  | MEM2 := F2 cell ^F1 init false
  | FERMER := MEM1 or MEM2
  |)
where
  boolean MEM1, MEM2;
end
```

### Intervalles

```
  | FO := S when H                      % [ .. [ %
  | OF := ZS when H                     % ] .. ] %
  | FF := (FIN default S) when H        % [ .. ] %
  | OO := (not DEBUT default S) when H  % ] .. [ %
```

### Automates

```
process CHRONO =
 ( ? event MA, TR, H   % Marche/Arret, Temps interm./RaZ, Top %
   ! integer AFF )

(| CPT ^= AFF ^= TOURNE ^= AFFCPT ^= MA ^+ TR ^+ H

 | TOURNE := not ZTOURNE when MA default ZTOURNE
 | AFFCPT := not ZAFFCPT when TR when TOURNE
            default TR when not TOURNE default ZAFFCPT
 | CPT := 0 when TR when ZAFFCPT and not TOURNE
        default ZCPT + 1 when H when TOURNE default ZCPT
 | AFF := CPT when AFFCPT default AFF$ init 0
 | ZTOURNE := TOURNE$ init false
```

```
 | ZAFFCPT := AFFCPT$ init true
 | ZCPT := CPT$ init 0
 |)
where
  boolean TOURNE, ZTOURNE, AFFCPT, ZAFFCPT;
  integer CPT, ZCPT;
end
```

Cette autre version utilise des états numérotés, avec 0 = arret/affichage actif (état initial), 1 = tourne/affichage actif, 2 = tourne/affichage bloqué, 3 = arrêt/affichage bloqué.

```
(| S ^= CPT ^= AFF ^= MA ^+ TR ^+ H
 | S :=      1 when MA when ZS=0
    default (0 when MA default 2 when TR) when ZS=1
    default (1 when TR default 3 when MA) when ZS=2
    default (0 when TR default 2 when MA) when ZS=3
    default ZS
 | CPT := 0 when TR when ZS=0
     default ZCPT+1 when H when ZS=1 or ZS=2
     default ZCPT
 | AFF := CPT when S <= 1 default AFF$ init 0
 | ZS := S$ init 0
 | ZCPT := CPT $ init 0
 |)
where
  integer S, ZS, CPT, ZCPT
end
```

### Sur-échantillonnage

**Extraction dans un entier N = 0..99 du chiffre dizaine, puis unité**

```
(| BASCULE := not BASCULE$ init false
 | CHIFFRE ^= BASCULE
 | N ^= when BASCULE

 | CHIFFRE := N/10 default (N cell ^BASCULE)-10*(CHIFFRE$)
 |)
where
  boolean BASCULE
end
```

On ne peut écrire $N - 10 * CHIFFRE$ puisque N et CHIFFRE n'ont pas la même horloge.

```
    % produit entre chaque N une suite de N event %
process NINTER =
  ( ? integer N
```

```
    ! event S )

(| CPT_DECR := N default ZCPT_DECR - 1
 | ZCPT_DECR := CPT_DECR$ init 0
 | N ^= when ZCPT_DECR = 0
 | S := when ZCPT_DECR /= 0
 |)
where
 integer CPT_DECR, ZCPT_DECR
end
```

## Compilation SIGNAL

**Clocks Constraints** : dans la ligne
$MAX := TAILLE\ when\ TAILLE > ZMAX$, la comparaison oblige TAILLE et ZMAX à être synchrones ; or cette équation définit l'horloge de MAX (et donc de ZMAX) comme **extraite** de celle de TAILLE (à cause du when) : c'est bien ce que l'on souhaite, puisque l'on ne veut MAX que lorsqu'il change. TAILLE ne doit donc pas être comparé directement à ZMAX, mais à une valeur "prolongée" du maximum actuel.

**Dependency cycle in the graph** : la **valeur** de MEMAX (comparée à celle de TAILLE) sert à calculer **l'horloge** de MAX et ZMAX. Si ZMAX est présent, on peut obtenir sa valeur (retard). Or, cette valeur est nécessaire... pour calculer la valeur de MEMAX, égale à ZMAX lorsque ZMAX est présent.

**Version correcte :**

```
 | MEMAX := MAX default ZMEMAX
 | ZMEMAX := MEMAX $ init 0
 | MAX := TAILLE when TAILLE > ZMEMAX
```

où la comparaison fixe l'horloge de ZMEMAX (et donc de MEMAX) à celle de TAILLE. La valeur retardée de MEMAX est disponible dès le début du calcul.

**Hiérarchie des horloges :**

```
   A = LG = ZLG
      |
      |
  [A=0] = FIN = TAILLE = MEMAX = ZMEMAX
      |
      |
  [TAILLE > ZMEMAX] = MAX
```

## Calculs de propriétés

- $B := {}^{\wedge}A$ se code par $b = a^2$
- $C := A \neq B$ se code par $c = -ab$ et $c^2 = a^2 = b^2$
- *Premier B toujours sorti ?*
  À partir de l'état initial $[1, 1, z_0]$, on calcule $c = -x + bx + 1$ égal à b pour $x = 1$.

**Sortie des valeurs de B lorsqu'il change**

Négligeons d'abord le cas de la première valeur.

```
| C := B when B /= P
| P := B $
| R := C $ | S := R $
```

Il faut montrer que R est toujours la négation de C, ou bien, en pensant au codage sous forme d'états, S doit être la négation de R.

En posant $b^2 = p^2 = 1$, et avec le codage ci-dessus pour l'opérateur $\neq$, on obtient :
$c = b(bp - 1), p = x$, donc : $c = x - b, c^2 = xb - 1, 1 - c^2 = -xb - 1$.
On vérifie : si $b = x, c = 0$;
si $b = -x, c = -b - b = b, c^2 = -1 - 1 = 1$.
À l'horloge de C, on a :
$r = c^2 y, y' = c + (1 - c^2)y$,
$s = c^2 z, z' = r + (1 - c^2)z$
soit, pour les variables d'état :
$x' = b$,
$y' = x - b - (xb + 1)y$,
$z' = (xb - 1)y - (xb + 1)z$.
Les états valides sont ceux où $z = -y$. Reprenons les deux cas :
Si $b = x, x' = x, y' = -(1 + 1)y = y, z' = z$ : état inchangé.
Si $b = -x, x' = -x, y' = -x, z' = y$ : dans les états d'arrivée, on a donc toujours $x' = y'$ ; et toute transition depuis un état $[k, k, z]$ donne un état d'arrivée $[-k, -k, k]$ correct. Tous les états à partir du second seront donc corrects.

Pour que le comportement soit correct dès le premier changement d'état, il suffit que $x_0 = y_0$. On choisira par exemple les initialisations :

```
| C := B when B /= P
| P := B $ init true
| R := C $ init true  | S := R $
```

Si l'initialisation de C avait été à *false*, et les premiers B à *true*, le premier changement *true* - *false* pour B n'aurait pas été reconnu comme valide (deux valeurs successives de C à *false*).

**Sortie de la première valeur de B**

Dans la programme ci-dessus, le premier B n'est pas sorti s'il est à *true* : l'automate boucle alors sur l'état initial. Ajoutons un signal D, qui n'est *vrai* que sur le premier instant :

```
| C := B when D or B /= P
| D := F $ init true
| F := false
```

Ceci entraine l'ajout d'une variable d'état que l'on appelle $v$ :
$f = -1, d = f^2 v = v, v' = f = -1, v_0 = 1$
Appelons $t$ l'opérande droit du *when* :
$t = bx(1 - v) - v - 1), t^2 = 1$,
$c = (v - 1)x + vb$, où l'on a bien $c = b$ pour $v = 1$.

La première transition est : $[x_0, y_0, z_0, v_0 = 1] \rightarrow [b, b, y_0, -1]$, soit une configuration dont on a vu qu'elle conduisait à un état valide. À partir du second instant, avec $v = -1$, on retrouve $c = x - b$, soit le système précédent.