Le langage SIGNAL : un exemple en segmentation automatique de la parole continue

Claude Le Maire

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

Collection Di J.

N° 1217

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

LE LANGAGE SIGNAL : UN EXEMPLE EN SEGMENTATION AUTOMATIQUE DE LA PAROLE CONTINUE

Claude LE MAIRE

Avril 1990

RR-1217

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Le langage SIGNAL: Un Exemple en Segmentation Automatique de la Parole Continue

Mars 1990

Publication Interne n° 527 - 112 Pages

Claude Le Maire

IRISA/INRIA, Campus de Beaulieu
35042 RENNES CEDEX, FRANCE

Résumé: Dans l'étude que nous présentons, nous appliquons les concepts de la programmation synchrone par le biais du langage SIGNAL et développons des outils afin de programmer un module complexe de segmentation acoustique en reconnaissance de la parole continue.
Toujours par le biais du langage SIGNAL, nous développons un environnement autour de ce module de segmentation, permettant à l'utilisateur d'observer ou d'intervenir pendant l'exécution du module.
Cette étude est une première étape vers la création d'un poste de traitement de signal basé sur la programmation synchrone.

Using SIGNAL for Automatic Segmentation of Continuous Speech

Abstract: In this report, we propose the use of synchronous programming concepts for continuous speech recognition; hence the use of the SIGNAL language. Our work consists in developing tools or mecanisms in order to facilitate the description and the implementation of a complex acoustic segmentation module. By the use of SIGNAL, we develop an environment for this module. So the user can observe the results or interact during the execution.
This study is a first step to the creation of a signal processing workstation based on synchronous programming.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Table des Matières

Chapitre Premier

Introduction

Dans les domaines de "l'informatique temps-réel", la notion de temps prend une importance particulière car elle est intrinsèquement liée aux applications d'un domaine. Ces applications ont été regroupées sous le terme générique de *systèmes réactifs.*
Ce terme a été introduit pour désigner des systèmes qui interagissent de façon répétitive avec leur environnement [13] . Une propriété importante de ces systèmes est leur déterminisme: ils répondent toujours de la même manière aux mêmes séquences d'entrées.
Les méthodes de programmation classiques s'appliquent mal aux systèmes réactifs. Cela est lié en grande partie aux notions de temps inhérentes à ces méthodes. Les langages classiques reposent en effet sur des mécanismes de communication asynchrones et ont une vue chronométrique du temps.

Les insuffisances de ces méthodes ont conduit au développement de plusieurs langages ou formalismes spécifiquement adaptés aux systèmes réactifs. C'est le cas des formalismes purement synchrones comme les langages SIGNAL [12] , ESTEREL [7] , LUSTRE [6] ou les Statecharts [14] . Ces formalismes ont une vue chronologique du temps et présentent une hypothèse de synchronisme plus ou moins forte. Nous considérons ici le langage SIGNAL développé à l'IRISA[1].

[1] Institut de Recherche en Informatique et Systèmes Aléatoires à Rennes

Le langage SIGNAL est un langage synchrone à flots de données pour la programmation de systèmes à temps-réel.
L'hypothèse de synchronisme se traduit sous deux aspects:

- en ce qui concerne les mécanismes internes du système: chaque action est instantanée (c.a.d. a une durée nulle)
- en ce qui concerne les communications avec le monde extérieur: les entrées et les sorties sont fixées à l'avance. L'ensemble des instants où les valeurs sont disponibles sur les ports d'entrée est totalement ordonné.

L'instantanéité de l'exécution des actions et l'ordre total des événements assurent le déterminisme.
D'un autre côté, la caractéristique flots de données permet de décrire facilement le parallélisme. L'ensemble des calculs à effectuer est représenté par un graphe orienté, dans lequel les arcs sont des chemins de données et les nœuds des opérations. Lors de l'exécution, un nœud est activé lorsque toutes ses entrées sont prêtes (règle flots de données), la notion de variable étant remplacée par celle de flots (suite de valeurs).

Différentes applications SIGNAL, par exemple [20] [11] [3] , ont permis de souligner trois points importants du langage:

- une application peut être développée de façon hiérarchique
- les systèmes d'équations sont immédiatement transposables
- des modules extérieurs écrits dans un autre langage peuvent être utilisés

Dans cette étude, nous nous plaçons dans le cadre de la reconnaissance de la parole continue, notre but final étant de créer un poste de traitement de signal basé sur la programmation synchrone [16] .
La définition d'un système de compréhension automatique de la parole nécessite, à côté des choix de représentation et d'utilisation des informations linguistiques, la mise en œuvre de stratégies et d'architectures logicielles particulières [10] .
Le dilemne qui se pose alors au concepteur est soit de définir des structures de données complexes (type blackboard) pour représenter les connaissances, soit porter l'effort sur une architecture du système et les structures de contrôle associées [15] . Le langage SIGNAL avec son approche synchrone permet de résoudre le premier point tout en présentant une structure hiérarchisée de l'application, la mise en œuvre du langage, transparente au concepteur, pouvant être effectuée sur différentes machines selon le code produit par le compilateur.

La segmentation, au même titre que l'étiquetage, fait partie du décodage acoustico-phonétique. Celui-ci peut être défini comme la transformation de l'onde vocale, en unités phonétiques étiquetées ou identifiées. Il occupe une place importante et est à l'heure actuelle un problème-clef dans le processus de reconnaissance automatique de la parole continue.
Nous nous sommes fixés en particulier les buts suivants:

- décrire une application grandeur nature: le module choisi manipule le temps de façon complexe et constitue une partie importante du décodeur acoustico-phonétique développé par l'équipe parole de l'IRISA
- utiliser SIGNAL dans sa version actuelle (version P1) en essayant de dégager (voire de résoudre) les insuffisances du langage pour ce genre d'application
- utiliser l'interface graphique du langage pour développer de façon hiérarchique l'application

Ce document présente tout d'abord le module de segmentation. Ce dernier s'appuie sur la méthode de divergence forward-backward [1] . Le chapître 2 la décrit de façon détaillée.
Puis nous présentons le langage SIGNAL en se plaçant du point de vue de l'utilisateur. Le noyau de base y est présenté ainsi que les principaux outils statiques et dynamiques développés dans la version P1 et le côté hiérarchique de SIGNAL.
Le chapître 3 décrit l'application proprement dite et sa mise en œuvre en SIGNAL. La façon de résoudre les problèmes posés par cette application est accompagnée de nombreuses figures tirées de l'interface graphique.
Le chapître suivant consiste à développer tout un environnement autour du module de segmentation de façon à faciliter la tâche de l'utilisateur; par exemple utiliser SIGNAL comme un outil de mise au point de programmes en temps-réel.
Enfin, dans le dernier chapître , nous faisons une brêve synthèse du travail effectué en présentant les perspectives à court-terme dirigées vers le poste de traitement de signal.
Les différentes boîtes à outils utilisées ne sont décrites qu'en annexe afin de ne pas surcharger la description de l'application.

Chapitre II

La méthode de divergence forward-backward

Un décodeur acoustico-phonétique prend en entrée le signal de parole et délivre en résultat une chaîne phonétique reconnue. Il occupe une place importante dans un système de reconnaissance de parole continue car à partir de cette chaîne phonétique, les modules supérieurs (étages lexicaux, syntaxiques, sémantiques,...) doivent reconstituer la phrase prononcée.
Le décodeur dont il est question ici est développé à l'IRISA dans le projet Automatique et Signal. Il se décompose en deux parties:

- un décodage statique du signal composé lui-même de deux parties:
  - un module de segmentation qui sépare le signal en différentes zones (les segments), les zones étant séparées par des frontières.
  - des identifications diverses par quantification vectorielle ou à partir de règles de décision... qui permettent d'étiqueter les frontières et les segments
- un décodage probabiliste où les observations sont les résultats du décodage statique et la sortie est l'écriture phonétique de la phrase prononcée

Dans ce chapitre, nous décrivons le module de segmentation. Il prend en entrée le signal de parole numérique et rend en sortie ce même signal segmenté (figure II.1). Ce module est considéré comme un pré-traitement et a pour but de détecter les changements brusques dûs à une modification de la source vocale, ou les changements plus doux dûs à une variation du conduit vocal (ou nasal). Il s'appuie sur la méthode forward-backward, elle-même basée sur un test statistique, le test de divergence Hinkley.

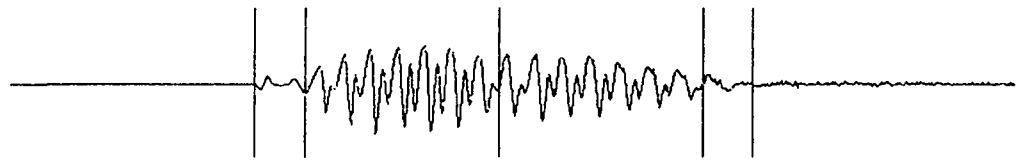

Figure II.1 : Le signal de parole numérique est découpé en zones.
N.B: le signal numérique est constitué d'une suite de valeurs (les échantillons). Ce qui est tracé ici est la courbe reliant ces valeurs.

II.1 Le test statistique de divergence Hinkley [2]

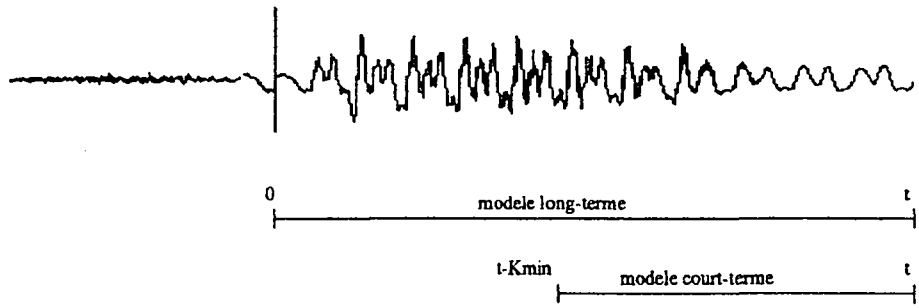

Figure II.2 : Localisation des modèles

Dans notre cas, deux modèles autorégressifs d'ordre M analysent le signal depuis la dernière frontière (figure II.2):

- un modèle autorégressif identifié séquentiellement sur une fenêtre croissante par la méthode de Burg [4] ("modèle long-terme"): il analyse le signal sur une fenêtre de longueur t qui croît à l'arrivée d'un échantillon
- un modèle autorégressif identifié sur une fenêtre glissante par la méthode d'autocorrélation [19] ("modèle court-terme"): il analyse le signal sur une fenêtre fixe comprenant les *Kmin* derniers échantillons

Une phase d'initialisation de longueur *Kmin* est donc nécessaire. Dans notre cas, *Kmin* est fixé à 20**F_ech* (F_ech, fréquence d'échantillonnage du signal de parole). Cela représente 256 échantillons et une durée de 20 ms si la fréquence vaut 12.8 KHz, un échantillon étant dans ce cas présent tous les 0.078 ms.

A chaque instant, une distance, la distance de Kullback, est mesurée entre les deux modèles. La valeur du test statistique de Hinkley à un instant t (W_t) est calculée à partir de cette distance à l'instant t (w_t), de la valeur du test à l'échantillon précédent (W_{t-1}) et d'un biais positif

$$W_t = \sum_{i=1}^{t}(w_i + biais) = W_{t-1} + w_t + biais.$$

Soit ($\max_{i \leq t} W_i$), le maximum courant du test statistique. Une frontière est détectée à l'instant N après franchissement d'un seuil de la différence entre la valeur courante du test et le maximum courant (figure II.3),

$$N = inf \{t \geq 1 / (\max_{1 \leq i \leq t} W_i) - W_t \geq seuil\}.$$

La frontière est positionnée à l'instant r correspondant à l'argument du maximum (cela correspond sur la figure au dernier instant où la différence entre le test et le maximum était nulle),

$$r = arg \{ \max_{1 \leq i < N} W_i \}.$$

Le fait que l'instant de détection d'une frontière ne soit pas le même que la position de cette frontière est très important. En effet le redémarrage (ou réinitialisation) ayant lieu à la frontière r, il est indispensable de retraiter les échantillons compris entre les instants r et N. Nous appellerons par la suite "reprise forward" le retraitement de ces échantillons.

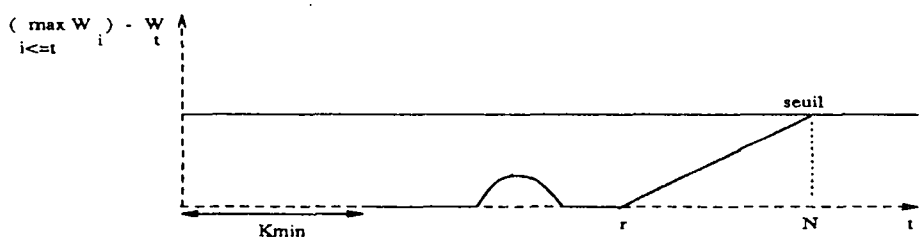

Figure II.3 : Test statistique de Hinkley: tracé de la courbe de la différence entre le maximum courant et la valeur courante du test.

II.2 Principe de la méthode de divergence forward-backward [1]

La nature de la statistique de Hinkley, à savoir sa non symétrie, a pour conséquence les omissions de certaines frontières, en particulier en présence de phonèmes nasalisés. Afin de pallier cette déficience, le test de divergence est introduit dans le sens rétrograde (d'où le nom forward-backward).

La valeur Lmin est introduite. La longueur totale de deux segments adjacents est obligatoirement supérieure à cette longueur, qui varie en fonction de l'omission recherchée. Lors de la détection d'une frontière r', si le segment [r,r'] est de longueur supérieure à Lmin, le test de divergence Hinkley lui est appliqué dans le sens rétrograde (fig. II.4.a) afin de repérer un éventuel oubli. Nous appellerons par la suite "reprise backward" le retraitement des échantillons de r' à r. A l'issue de ce test, deux cas se présentent:

- aucune détection n'est apparue: la frontière r' est validée et la segmentation redémarre à l'instant r'
- une ou plusieurs détections sont apparues aux instants r'_i (fig. II.4.b): la frontière r'_i la plus proche de r (avec $r'_i - r > Dmin$, longueur minimum d'un segment) est validée et la segmentation redémarre à cette frontière.

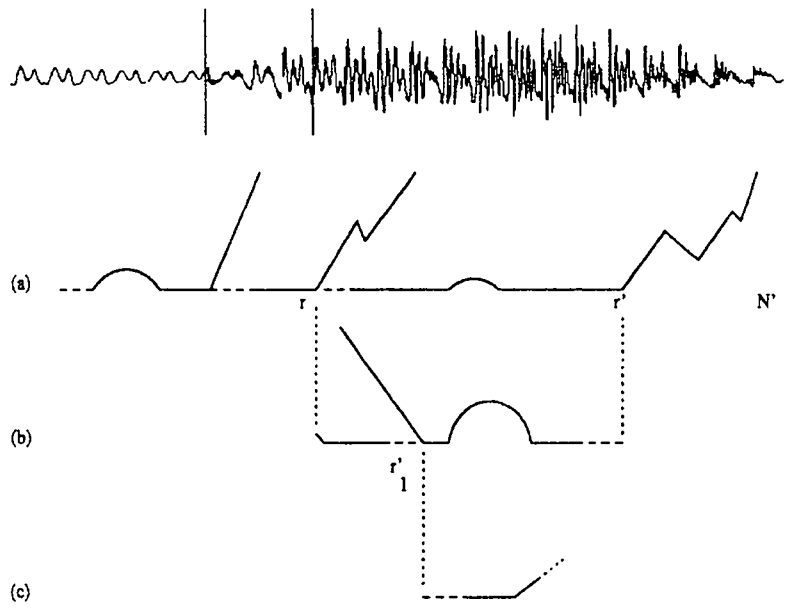

Figure II.4 : Le retour-arrière.

Les paramêtres Lmin et Dmin sont fixés en fonction des objectifs et dépendent d'un test de voisement effectué en parallèle à la segmentation forward:

- lors de la recherche de l'explosion d'une plosive non voisée, le segment précédent correspond à une zone silencieuse et le segment courant à une zone voisée, Dmin et Lmin sont choisis petits. Ils représentent respectivement la durée minimum d'une aspiration et celle d'un phonème voisé
- lors d'une recherche de frontière entre deux phonèmes voisés, le segment courant correspond à une zone voisée. Dmin et Lmin représentent respectivement la longueur minimale d'un phonème voisé et celle de deux phonèmes voisés articulés et sont donc choisis plus grands

Seuls ces cas d'omissions sont actuellement traités

II.3 Le module segmentation final

Le module de segmentation final est constitué de deux modules de segmentation opérant sur le même signal d'entrée:

- un module de segmentation sur le signal d'entrée filtré passe-haut (>4000Hz). Il est constitué du test de divergence Hinkley sur le signal filtré (le biais et le seuil sont fixes). Il rend en sortie des segments appelés segments passe-haut
- un module de segmentation s'appuyant sur la méthode de divergence forward-backward telle qu'elle a été décrite précédemment, avec une variante dans le sens direct pour rendre cette segmentation plus robuste: trois tests opèrent en parallèle (et non pas un seul), proposant chacun leurs propres détections. La frontière est localisée après concertation des résultats des trois tests. Ces trois tests sont les suivants:
  - le test de Hinkley sur le signal d'entrée (le biais et le seuil sont variables)
  - le test de voisement sur le signal d'entrée. Il permet de détecter de façon grossière les zones voisées-non voisées-silence, principalement durant la phase d'initialisation. Contrairement au test de Hinkley, il analyse le signal par blocs glissants (dans notre cas, blocs de 10 ms). A l'issue de la phase d'initialisation, ce test détecte si le segment en cours est une zone voisée ou non voisée ce qui permet de fixer deux familles de paramêtres biais-seuil du test de divergence Hinkley.

– le test de Hinkley activé sur le signal d'entrée filtré passe haut

Il rend en sortie les segments appelés forward-backward ainsi que les étiquettes de voisement associées à chaque segment forward-backward (fournies par le module de voisement).

N.B: Nous remarquons que les résultats du module de segmentation sur le signal d'entrée filtré peuvent remettre en cause ceux du module forward-backward, l'inverse n'étant pas vrai.

Chapitre III

Le langage SIGNAL

Dans ce chapitre, ne sont présentés que les aspects du langage pouvant faciliter la compréhension des chapitres ultérieurs: simplicité du noyau de base, extensibilité et développements autour du noyau, modularité et environnement graphique. Pour une présentation détaillée du langage (sémantique, calcul d'horloges, graphe de dépendances, compilation...) se reporter par exemple aux travaux [8] , [12] , [17] ou [18] .

III.1 Les objets de base

Les objets de base du langage sont les signaux. Un signal est une paire (flot, horloge) où le flot est une suite ordonnée de données typées de longueur non spécifiée et l'horloge associée à ce flot spécifie les instants auxquels les données sont disponibles.
Par exemple, le signal de parole peut être représenté par un tel objet. Le flot correspond aux valeurs des échantillons et l'horloge aux instants où les échantillons sont présents.
La notion de temps dans un système est définie par les relations entre les horloges des différents signaux du système, sans faire référence à un temps universel.

Un processus SIGNAL décrit à la fois les relations fonctionnelles et temporelles entre les signaux. Un processus est une boîte noire pouvant communiquer avec le monde externe ou d'autres processus par l'intermédiaire de signaux appelés ports d'entrée et ports de sortie.
Par exemple le module de segmentation présenté dans le chapitre précédent est un processus. Le port d'entrée est le signal de parole et les ports de sortie les segments.

III.2 La modularité

Un programme SIGNAL est représenté par un réseau statique de processus interconnectés, ces derniers pouvant être eux-mêmes composés de sous-processus. Le développement d'un environnement graphique [9] permet d'avoir une vue du programme soit textuelle soit graphique, ce dernier cas laissant apparaître une vue modulaire du programme (figure III.1).

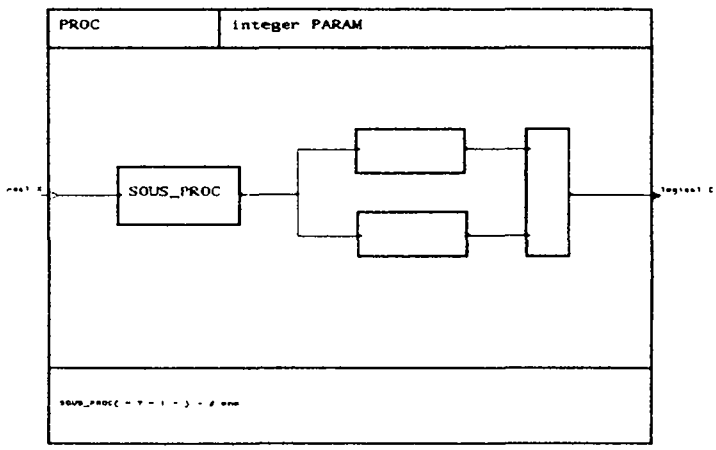

Figure III.1 : Vue graphique d'un processus de nom PROC. Les boîtes représentent les processus, les fils représentent les liens entre les signaux d'entrée/sortie. Dans cet exemple, X est un port d'entrée et C un port de sortie. Une zone est réservée pour le passage de paramètres (ou constantes): ici Param. De même une zone est réservée pour la déclaration des sous-processus: ici SOUS_PROC

Cet environnement comprend deux outils principaux: un éditeur graphique et un décompilateur graphique. L'éditeur permet de concevoir les programmes de manière ascendante ou descendante. Le décompilateur, en cours de réalisation, permet notamment de visualiser graphiquement des programmes textuels. Dans ce rapport, les vues graphiques sont utilisées pour illustrer la description des processus.

III.3 Le noyau du langage SIGNAL

Le noyau du langage SIGNAL comprend cinq instructions de base:

- un opérateur *composition* permet la construction de réseaux et exprime le parallélisme entre les processus. Soient deux processus P et Q, cet opérateur (on note **P | Q**) connecte les ports de sortie de P (resp. de Q) aux ports d'entrée de Q (resp. de P) possédant le même nom. P et Q ne doivent pas posséder de sorties de même nom (figure III.2). La composition, opérateur commutatif et associatif, est assimilable à l'union d'équations ou de systèmes d'équations permettant de former un nouveau système.

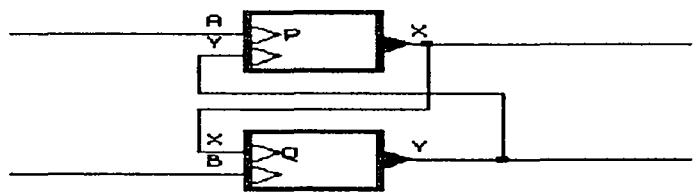

Figure III.2 : Un exemple graphique de P | Q

- trois processus statiques — les valeurs des ports de sortie dépendent des valeurs des ports d'entrée aux instants où celles-ci sont disponibles, sans tenir compte des valeurs passées — :

  - les *fonctions* sont des transformations instantanées élémentaires sur les données (arithmétiques, logiques, etc):
    $$p(x_1, ..., x_n) \Leftrightarrow \forall t : p(x_{1_t}, ..., x_{n_t})$$
    Tous les signaux impliqués sont synchrones (c.a.d. possèdent la même horloge)
  - le *filtrage* (on note **x:=y when c**) est un sous-échantillonnage conditionnel. La sortie x prend la valeur de y lorsque y et c (signal booléen) sont présents, c possédant la valeur vraie; sinon la sortie x n'est pas définie (figure III.3).

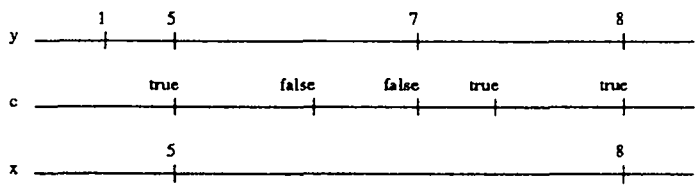

Figure III.3 : Exemple de diagramme temporel de l'instruction x:=y when c, x et y sont de types entiers

L'expression "x:=y when c" est asynchrone; l'horloge de x est inférieure aux horloges de y et c (c.a.d. moins fréquente)

- le *mélange* (on note **x:=u default v**) donne la priorité au signal u lorsque u est présent; sinon x prend la valeur de v si v est présent; sinon la sortie x n'est pas émise (figure III.4).
  L'expression "x:=u default v" est asynchrone; les horloges de u et v sont inférieures à celle de x

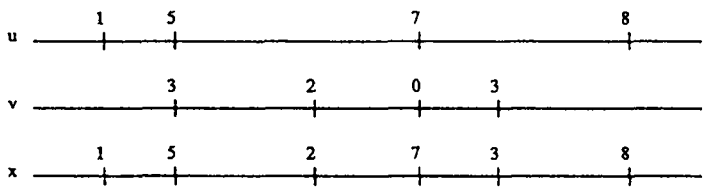

Figure III.4 : Exemple de diagramme temporel de l'instruction x:=u default v, x, u et v sont de types entiers

- un processus dynamique, le *retard* (on note **zx:=x $ 1**), exprime la connaissance du passé. Le retard se comporte comme un registre de type FIFO, de telle façon que $zx := x\$1 \Leftrightarrow \forall t > 1 : zx_t := x_{t-1}$ (x doit être initialisé à x_0).
  Les signaux x et zx sont synchrones (figure III.5).

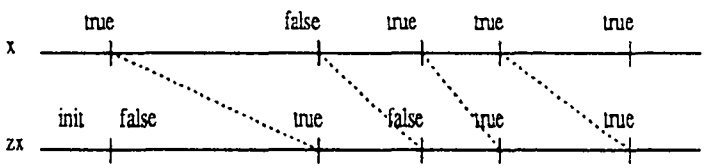

Figure III.5 : Exemple de diagramme temporel de l'instruction zx:=x $ 1, x et zx sont de types booléens

La construction des réseaux se fait par interconnexion de processus en identifiant les noms de ports. Cela impose l'utilisation de mécanismes de *renommage* (notés ?x:a pour les ports d'entrée, et !y:b pour les ports de sortie) et explique également les mécanismes de *masquage* de ports (P/x masque le port de sortie x, tandis que P!!x masque tous les ports de sortie sauf x) ou de *rebouclage* (P@x connecte la sortie x et l'entrée x de P).

Des opérateurs de composition dérivés sont également ajoutés (par exemple P;Q exprime la *composition séquentielle*).

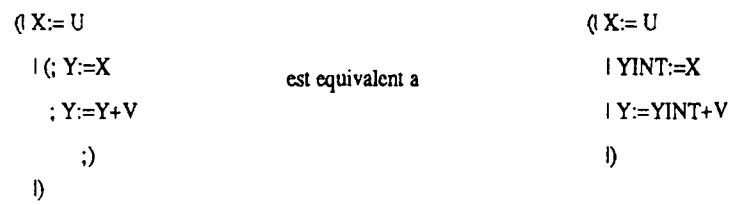

Figure III.6 : Exemple de composition séquentielle

A partir du noyau, différents outils ont été développés dans la version actuelle.

III.4 Les outils statiques développés autour du noyau.

De nombreux outils statiques ont été développés autour du noyau. Ils peuvent être classés en deux catégories, les expressions sur signaux et les expressions sur processus. Nous présentons ici les principaux:

- l'opérateur *événement* est un opérateur temporel (on note **event y**). L'expression "event y" produit un signal booléen, toujours vrai, à l'horloge du signal d'entrée y. Le signal produit définit ainsi l'horloge de y

- l'opérateur *mémorisation* est un opérateur temporel (on note **x:=y cell c**). L'horloge de x est la réunion de l'horloge de y et de l'horloge de c=vrai. Le signal x contient la dernière valeur disponible de y.
  N.B. Le signal x doit être initialisé à x_0

- la *composition conditionnelle* construit un nouveau processus à partir de deux autres processus (on note **if c then P else Q fi**). L'ensemble des noms d'entrée du processus résultant est formé par l'union des noms d'entrée de P et Q, et des signaux de c (expression booléenne). L'ensemble des noms de sortie est l'union des sorties de P et Q. Les sorties de P (resp. de Q) ne sont disponibles que lorsque c est présent et vrai (resp. faux) et les sorties de P et Q de même nom sont mélangées

- les *tableaux de processus* permettent de disposer de structures de répétition (on note **array i to n of P with <signaux d'entrée> end**). Les signaux d'entrée sont des vecteurs de longueur n (expression entière).

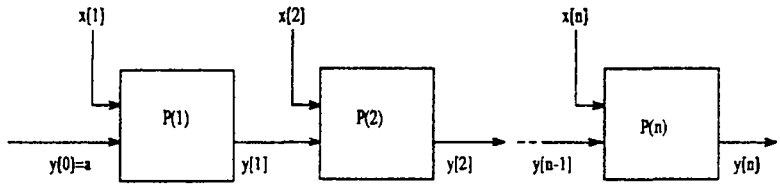

Figure III.7 : Block-diagramme d'un tableau régulier de processus

Dans le cas de la figure III.7, les signaux d'entrée sont les vecteurs x et y. Le signal y est construit pendant la répétition, il faut donc donner une valeur d'initialisation à y[0].
N.B. Il est possible d'effectuer la répétition de droite à gauche . Dans ce cas il faut initialiser y[n]

- la *synchronisation explicite* est un processus exprimant des contraintes sur les horloges de certains signaux (on note **synchro** $x_1, ..., x_n$). Les signaux d'entrée $x_1, ..., x_n$ sont contraints à être synchrones.
  N.B. Aucune sortie n'est produite.

III.5 Les outils dynamiques développés autour du noyau.

Quelques outils dynamiques d'expressions sur signaux peuvent être cités dans la version actuelle:

- le *décalage temporel* est une extension du retard (on note **x:=y $m window n**). Ce processus définit le signal x retardé de m par rapport au signal y et sur lequel on prend une fenêtre glissante de dimension n. Les signaux x et y sont synchrones.
  N.B. Lorsque m=1 et n=1, on retrouve le *retard*

- Quelques compteurs ont été développés. Le plus utilisé est le *compteur simple* (on note **x:=#c**). Ce compteur compte le nombre d'occurences du signal booléen c à vrai. Les signaux x et c=vrai sont synchrones.

Chapitre IV

La description du module segmentation en SIGNAL

Dans ce chapitre, nous développons l'approche synchrone du module de segmentation et sa mise en œuvre en SIGNAL. Cette mise en œuvre a nécessité la création de deux bibliothèques d'outils SIGNAL:

- une bibliothèque de processus SIGNAL facilitant la programmation des parties synchronisations des algorithmes
- une bibliothèque de processus standards de traitement de signal facilitant la programmation des modules de traitement des données.

Chacun de ces processus peut être utilisé dans plusieurs modules. Pour cette raison et aussi pour gagner en clarté, ils ne sont pas décrits dans ce chapitre mais en annexe.

La méthode de programmation suivie lors de cette étude peut se résumer en quatre points:

(1) Mise en évidence des caractéristiques temporelles de l'algorithme

(2) Utilisation de l'aspect modulaire de SIGNAL pour décrire l'application

(3) Séparation des aspects contrôle et traitement de données à l'intérieur de chaque module

(4) Construction du programme par réunion de tous les modules développés

IV.1 Les caractéristiques temporelles de l'algorithme forward-backward

Le module de segmentation présenté au chapitre 2 est un algorithme qui manipule le temps de façon complexe. Afin de s'en convaincre, nous soulignons les caractéristiques temporelles ainsi que les difficultés de synchronisation rencontrées lors de la description de l'algorithme.
L'algorithme s'articule autour de quatre tests. Trois tests s'exécutent en parallèle lorsque la segmentation est en mode direct (ou forward). Il s'agit du test de Hinkley (HINKLEY), du test de voisement (VOISEMENT) et du test de Hinkley sur le signal filtré (HINKLEY_FILTRE). Un test de Hinkley (HINKLEY_BAK) s'exécute lorsque la segmentation est en mode rétrograde (ou retour-arrière ou backward).
Les problèmes principaux concernent la gestion des informations produites par les quatres modules, et la gestion délicate de la mémoire lors des périodes de reprise.

La gestion des informations.

- l'appel au retour-arrière est bloquant pour la segmentation forward. Il ne peut donc pas y avoir de parallélisme entre HINKLEY_BAK d'une part et les trois autres tests d'autre part. Cela impose de commuter ces tests lors des passages forward-backward et backward-forward
- les trois tests de la segmentation forward s'exécutent en parallèle. HINKLEY et HINKLEY_FILTRE traitent le signal échantillon par échantillon, tandis que VOISEMENT traite le signal par blocs glissants ce qui rend difficile la coordination de ces tests
- la concertation entre les trois tests, en mode forward, doit avoir lieu "le plus souvent possible" afin de prendre une décision "au plus tôt".

La gestion de la mémoire.

- une période de reprise, quelle soit en forward ou backward, impose une réutilisation d'échantillons déja émis. Cela implique le stockage de ces échantillons afin d'éventuellement les réutiliser.
- HINKLEY_FILTRE est un module indépendant. Il doit gérer lui-même ses propres périodes de reprise, tout en intervenant sur celles de la segmentation forward-backward

- HINKLEY_BAK doit également gérer des périodes de reprise puisque le retour-arrière est un test de Hinkley
- l'imbrication de périodes de reprise est autorisée puisqu'un retour-arrière peut avoir lieu pendant une reprise
- chacun des trois tests (sens direct) propose ses propres frontières. Mais lorsqu'un test propose une frontière, celle-ci n'est pas forcément validée immédiatement, mais peut-être mise en attente des résultats des deux autres tests. Dans ce cas, il faut suspendre le test correspondant. Il faut donc gérer les phases de contrôle de suspensions de processus
- enfin, les biais et seuils de HINKLEY doivent être réglés en fonction de l'étiquette de voisement proposée par VOISEMENT.

Les problèmes temporels que nous venons d'exposer peuvent être en partie résolus en SIGNAL en suréchantillonnant le signal d'entrée. Entre deux échantillons du signal d'entrée, on réinjecte les échantillons nécessaires aux reprises.
Lors d'une détection (après concertation), il faut distinguer deux cas:

- `pas de retour-arrière`: réinjection, en mode direct, des échantillons dus à la reprise forward (figure IV.1)
- `retour-arrière`: *réinjection des échantillons* dus à la reprise backward (retour-arrière) jusqu'à la dernière frontière validée, puis des échantillons dus à la reprise forward après le retour-arrière. Les figures IV.2 et IV.3 illustrent respectivement les cas où le retour-arrière n'a rien détecté et le cas où il a détecté une frontière

L'horloge du signal suréchantillonné est la réunion des horloges de trois signaux: le signal d'entrée, le signal de reprise forward et le signal de reprise backward. Un tel outil dynamique n'existe pas en SIGNAL, ce qui justifie la création d'un processus *AUTOREPRODUCTION* (annexe B.13 et paragraphe IV.2.3).

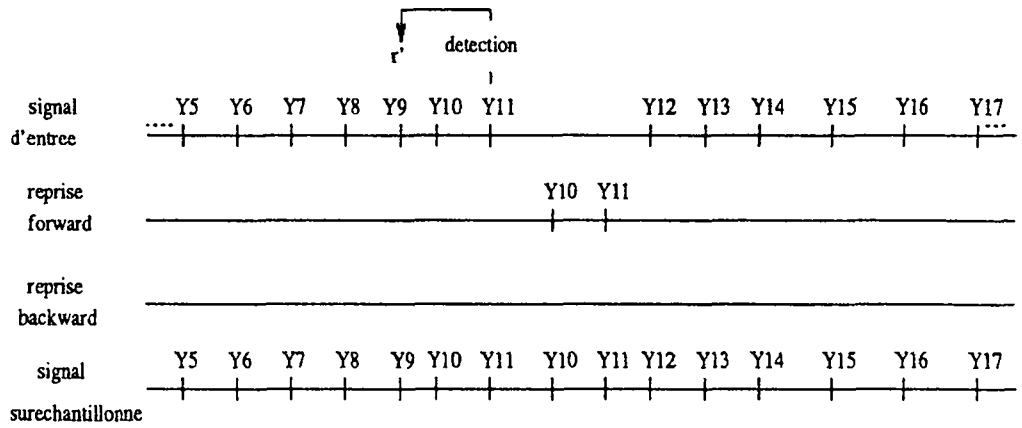

Figure IV.1 : Pas de retour-arrière: une frontière est détectée à l'arrivée de l'échantillon Y11, la frontière étant localisée en Y9. Il n'y a pas de retour-arrière. La frontière est donc placée en Y9 et il faut réinjecter les échantillons arrivés après Y9 (dans notre cas Y10 et Y11).

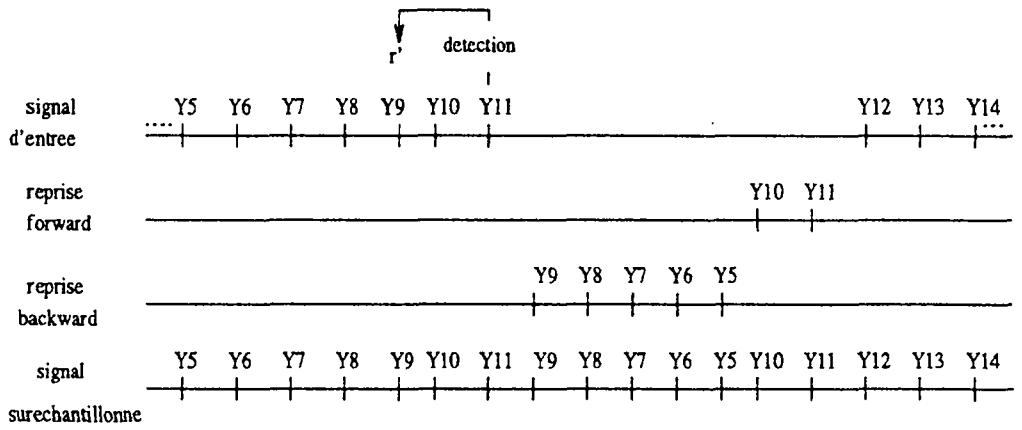

Figure IV.2 : Retour-arrière: une frontière est détectée à l'arrivée de l'échantillon Y11, la frontière étant localisée en Y9. Il y a un retour-arrière. La frontière n'est donc pas validée. On réinjecte les échantillons dans le sens rétrograde jusqu'à la dernière frontière (dans notre cas Y9 Y8 Y7 Y6 Y5, la dernière frontière est supposée placée en Y4). Le retour-arrière n'ayant rien détecté, la frontière est validée et placée en Y9. Il faut donc redémarrer à cette frontière et réinjecter les échantillons arrivés après Y9 (dans notre cas Y10 et Y11).

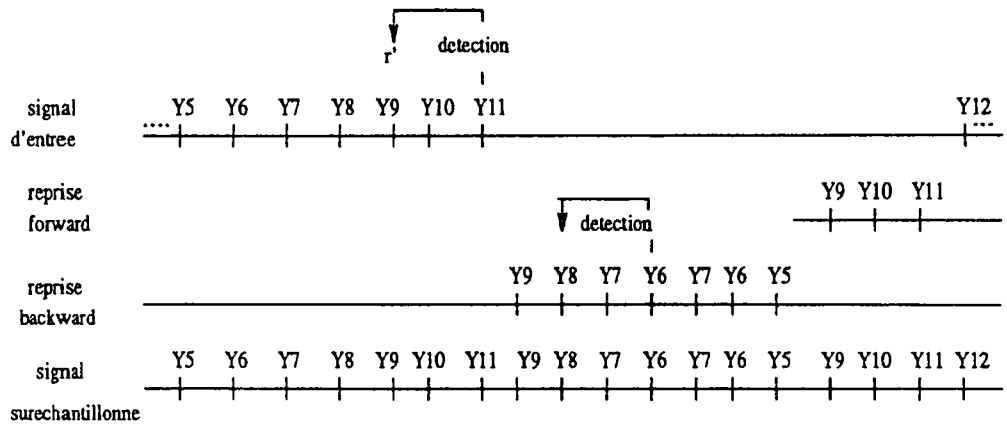

Figure IV.3 : Retour-arrière: une frontière est détectée à l'arrivée de l'échantillon Y11, la frontière étant localisée en Y9. Il y a un retour-arrière. La frontière n'est donc pas validée. On réinjecte les échantillons dans le sens rétrograde jusqu'à la dernière frontière (dans notre cas Y9 Y8 Y7 Y6 Y5, la dernière frontière est supposée placée en Y4). Une frontière est détectée à l'instant Y6 et localisée en Y8 pendant le retour-arrière. On met en attente cette frontière située en Y8. Et on redémarre le retour-arrière en Y8 (d'où la réinjection de Y7 et Y6 pendant le retour-arrière). A la fin du retour-arrière, la frontière située en Y8 est validée. Le redémarrage a lieu en Y8 (réinjection forward de Y9 Y10 Y11).

IV.2 L'aspect modulaire de l'algorithme et la description des modules

Avant de présenter le module de segmentation dans son ensemble, nous introduisons la notion de *point de reprise potentiel*:

le point de reprise potentiel

- lors du test de Hinkley (figure II.3), le point de reprise potentiel correspond au dernier instant où la différence entre le test et le maximum est nulle (points cerclés sur la figure IV.4). Ainsi lors d'une détection, le point de reprise potentiel est le point où l'on situe la frontière.

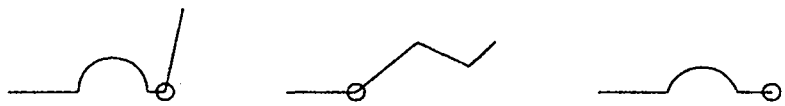

Figure IV.4 : Points de reprise potentiels lors de tests de Hinkley

- lors du test de voisement – traitement du signal par blocs glissants – le point de reprise potentiel correspond à la borne de l'intervalle où se situe la différence de voisement (s=silence, nv=non voisé sur la figure IV.5)

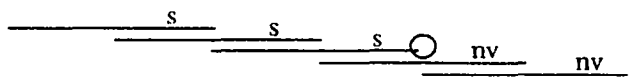

Figure IV.5 : Point de reprise potentiel lors d'un test de voisement. Les traits horizontaux représentent les blocs analysés par le test de voisement.

Le point de reprise potentiel va nous permettre de prendre la décision de détection "au plus tôt" sans attendre que les trois modules détectent une frontière.

Prenons l'exemple où le test de voisement détecte une frontière. Après concertation, cette frontière est tout de suite validée car les points de reprises potentiels des tests de Hinkley et de Hinkley sur le signal filtré sont situés après la frontière de voisement (figure IV.6). Il n'y a pas besoin d'attendre que ces deux tests détectent une frontière.

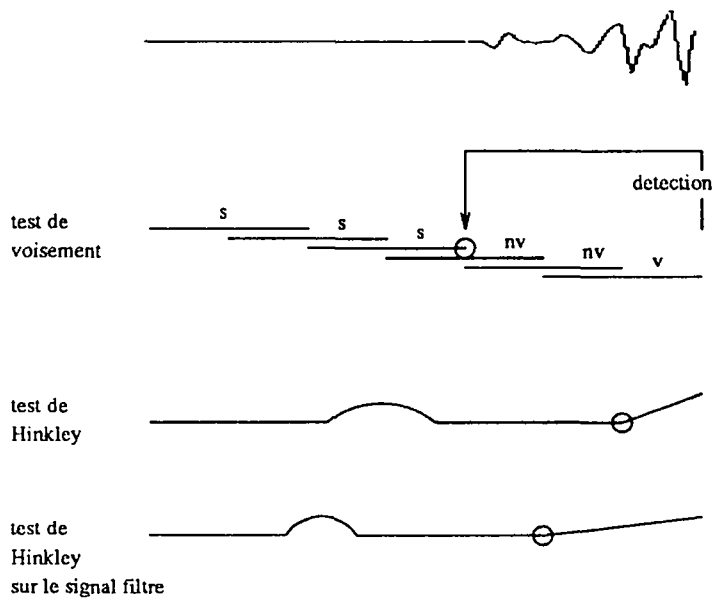

Figure IV.6 : La frontière détectée par le test de voisement est validée. Les étiquettes de voisement sont "s" pour silence, "v" pour voisé et "nv" pour non voisé.

Le programme principal: le module de segmentation acoustique

Ce processus représente le module de segmentation acoustique final décrit au chapitre 2. Il prend en entrée le signal de parole **PAROLE** et délivre en sortie deux sortes de segments, les segments forward-backward et les segments passe-haut (paragraphe II.3). En SIGNAL, nous construisons donc deux modules (figure IV.7):

- un module autonome *MODULE_FILTRE* rendant en sortie les segments passe-haut **NUMF**. Il prend en entrée le signal **PAROLE** et la réponse impulsionnelle **H** (nous n'avons pas parlé plus tôt de la réponse impulsionnelle car elle n'intervient que pour filtrer le signal de parole). Il rend également en sortie les points de reprise potentiels du test de Hinkley sur le signal filtré
- un module *MODULE_FORWARD_BACKWARD* rendant en sortie les segments forward-backward **NUM**. Il prend en entrée le signal **PAROLE** et les signaux de sortie de *MODULE_FILTRE*. Il rend également en sortie les étiquettes de voisement **VOIS** associées à chaque segment forward-backward.

Le paramètre **KMIN** représente la longueur de la fenêtre d'initialisation.
Le paramètre **LIMITEBAK**, qui peut être négatif, indique que le retour-arrière est effectué non pas jusqu'à la frontière r (figure II.4), mais jusqu'à $r-$**LIMITEBAK**.
Le paramètre **MAXTAB** est la taille de la fenêtre glissante dans laquelle est stocké **PAROLE**. Il dépend de **LIMITEBAK**.

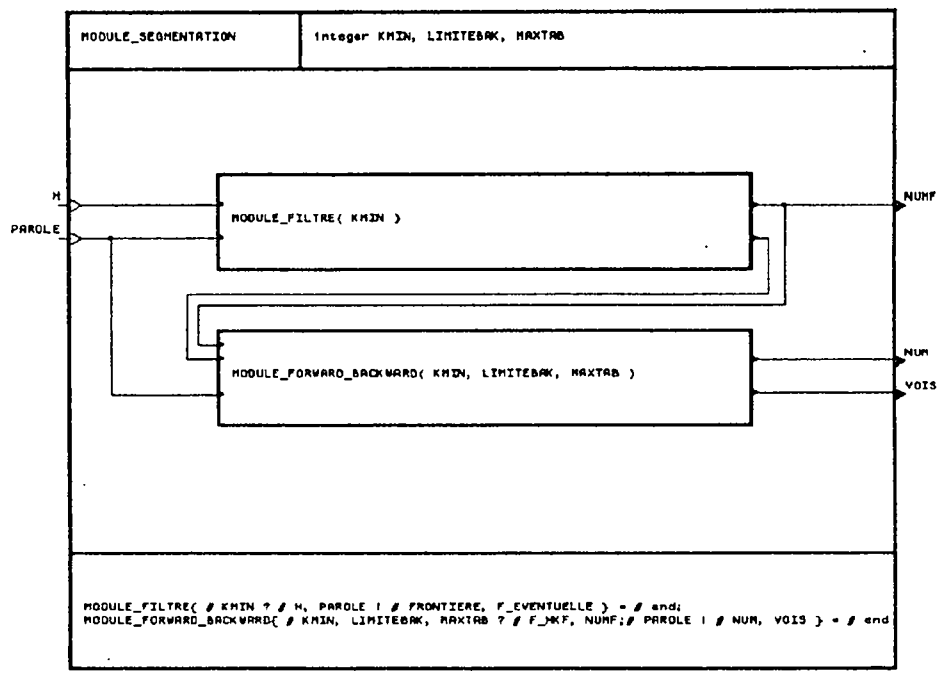

Figure IV.7 : Le module de segmentation

Avant de présenter et détailler les deux modules principaux *MODULE_FORWARD_BACKWARD* et *MODULE_FILTRE* qui composent le module final, nous décrivons deux façons de programmer le test de Hinkley:

- une première façon est de considérer que tout le contrôle inhérent aux reprises est effectué à l'extérieur du module de Hinkley
- une seconde façon est de gérer ce contrôle à l'intérieur même du module

La première façon de procéder sera utilisée dans le processus *MODULE_FORWARD_BACKWARD*, l'autre dans *MODULE_FILTRE*.

IV.2.1 Le module de Hinkley

Le processus *TEST_HINKLEY*.

Ce processus effectue le test de Hinkley sur un signal **Y** sans gérer les périodes de reprise. Il rend en sortie le point de reprise potentiel **F_EVENTUELLE** (à chaque évènement **Y**) et un signal booléen **DETECTION** indiquant s'il y a ou non détection. Le test est réinitialisé de l'extérieur par le signal **RAZ**. Le biais et le seuil sont représentés par les signaux du même nom.
Le test de Hinkley peut se diviser en quatre phases:

- estimation des modèles autorégressifs (processus *MODELE*, annexe A.6)
- calcul de la distance et de la statistique de Hinkley (processus *STAT_HINKLEY*, annexe A.7)
- calcul du maximum courant (processus *MAX*, annexe A.8)
- calcul du point de reprise potentiel et localisation de la frontière lors d'une détection

En SIGNAL, nous créons une boîte noire pour chaque phase et nous assemblons ensuite ces quatre boîtes (figure IV.8).

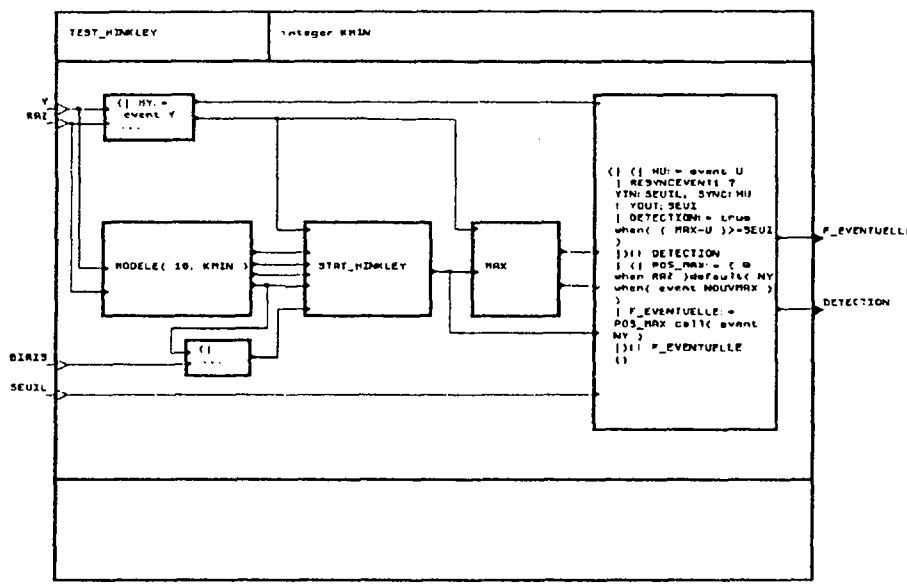

Figure IV.8 : Le processus *TEST_HINKLEY*

Les signaux de sortie des deux modèles autorégressifs, l'énergie résiduelle et l'erreur, sont des entrées de *STAT_HINKLEY* et permettent de calculer la distance entre les deux modèles et ainsi la statistique **U** de Hinkley. Ils ne sont émis que lorsque le test est actif, c'est à dire après une période d'initialisation de longueur **KMIN** gérée dans *MODELE*: **VCOURT**, **ERRCOURT** sont issus du modèle court-terme et **VLONG**, **ERRLONG** sont issus du modèle long-terme.
Le point de vue temporel de ce processus est décrit sur la figure IV.9.

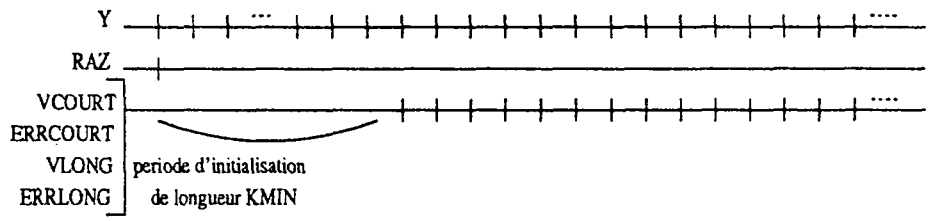

Figure IV.9 : L'entrelacement de signaux du processus *MODELE*

Le biais est resynchronisé à l'horloge de ces signaux actifs.
Le maximum **MAX** de la statistique est calculé dans *MAX* à partir de la statistique **U**.

Une boîte noire (située en haut à gauche sur la figure IV.8) calcule les signaux **RAZTEST** et **NY**.
Le signal **NY** est le compteur d'événements **Y** depuis la réinitialisation **RAZ**. Il est utilisé pour localiser le point de reprise potentiel.
Le signal **RAZTEST** est présent à la fin de chaque période d'initialisation (vrai lorsque **NY=KMIN**). Il permet de réinitialiser les processus *STAT_HINKLEY* et *MAX*.

Le signal **DETECTION** de détection est émis lorsque la différence entre **MAX** et **U** dépasse le seuil (le seuil est resynchronisé à l'horloge de ces signaux dans la boîte noire).
Pour localiser le point de reprise potentiel, il suffit de localiser le maximum puis de mémoriser ce maximum à l'horloge de **Y**, le maximum étant simplement la valeur de **NY** lors d'un nouveau maximum.

Le processus *HINKLEY_OPT.*

Ce processus effectue le même test de Hinkley, la gestion des reprises étant interne (pas de signal de réinitialisation externe) et tranparente à l'extérieur du module. Nous parlons ici des reprises dites forward car le retour-arrière n'intervient pas pour programmer le test de Hinkley. Pour rendre la gestion transparente, il faut gérer la reprise (ou l'éventuelle reprise) de façon parallèle au calcul de la statistique et du maximum. Pour cela nous utilisons la méthode suivante:

- un processus primaire effectue les quatre phases du test de Hinkley (*MODELE*, *STAT_HINKLEY*, *MAX* et le calcul des points de reprise potentiels et du signal de détection).
- un processus secondaire effectue en parallèle la phase d'estimation des modèles (*MODELE*). Mais il est réinitialisé à chaque nouveau point de reprise potentiel. Ainsi lors d'une détection il est prêt à continuer la phase d'estimation des modèles et à effectuer les trois autres phases
- un commutateur primaire/secondaire permet, lors d'une détection, de commuter les processus: l'ancien processus primaire devient secondaire et est réinitialisé, l'ancien processus secondaire devient primaire

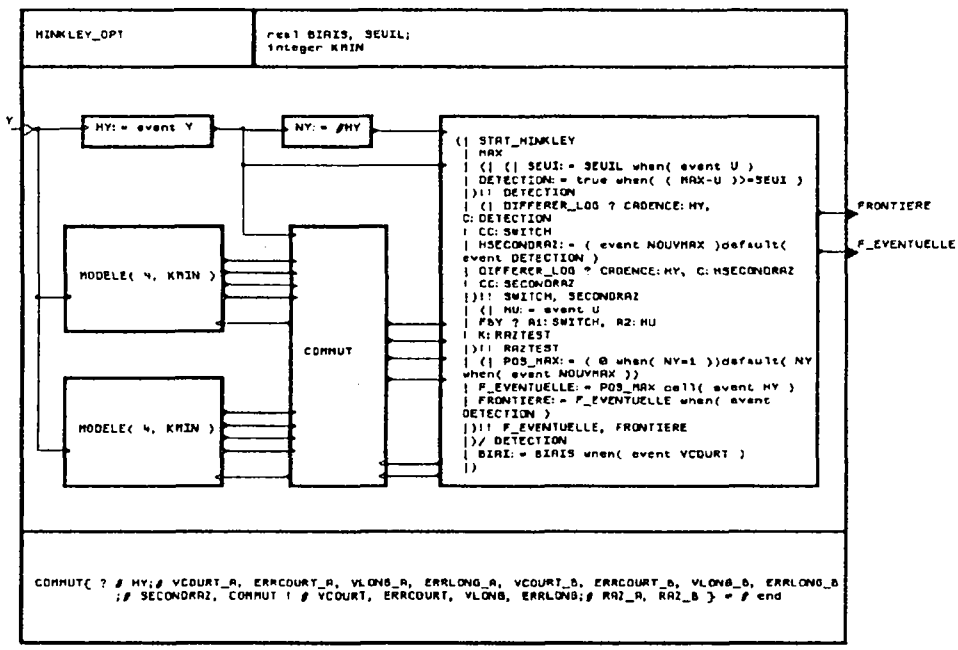

Figure IV.10 : Le processus *HINKLEY_OPT*

Le processus *COMMUT* représente le commutateur primaire/secondaire. Il utilise pour cela le processus *FLIPFLOP* (annexe B.9) qui agit comme une bascule. Le signal **MAIN_A** indique lequel des processus A ou B est prioritaire; il prend la valeur vrai lorsque le processus A est primaire et faux lorsqu'il est secondaire. Il est resynchronisé à l'horloge de **Y**:

```
(| (| FLIPFLOP ?switch:commut !b:bascule,zb:zbascule /zbascule
   | RESYNCEVENT3 ?yin:bascule,sync:hy !yout:main_a
   |)/bascule
 | if main_a
   then
      (| vcourt:=vcourt_a
       | errcourt:=errcourt_a
       | vlong:=vlong_a
       | errlong:=errlong_a
       | raz_b:=secondraz
       |)
   else
      (| vcourt:=vcourt_b
       | errcourt:=errcourt_b
       | vlong:=vlong_b
       | errlong:=errlong_b
       | raz_a:=secondraz
       |)
   fi
|)/main_a
```

En sortie, les signaux **VCOURT**, **ERRCOURT**, **VLONG** et **ERRLONG** prennent les valeurs des signaux du processus primaire. Le signal **SECONDRAZ** de réinitialisation est connecté au signal **RAZ** du processus secondaire. D'un point de vue temporel, lors d'une détection, le signal **SWITCH** est présent et la commutation a lieu, le signal **SECONDRAZ** est présent et égal à vrai ce qui réinitialise l'ancien processus primaire devenu secondaire.

Les trois autres phases et les signaux liés à la commutation et à la réinitialisation sont regroupés dans une boîte noire (figure IV.11).

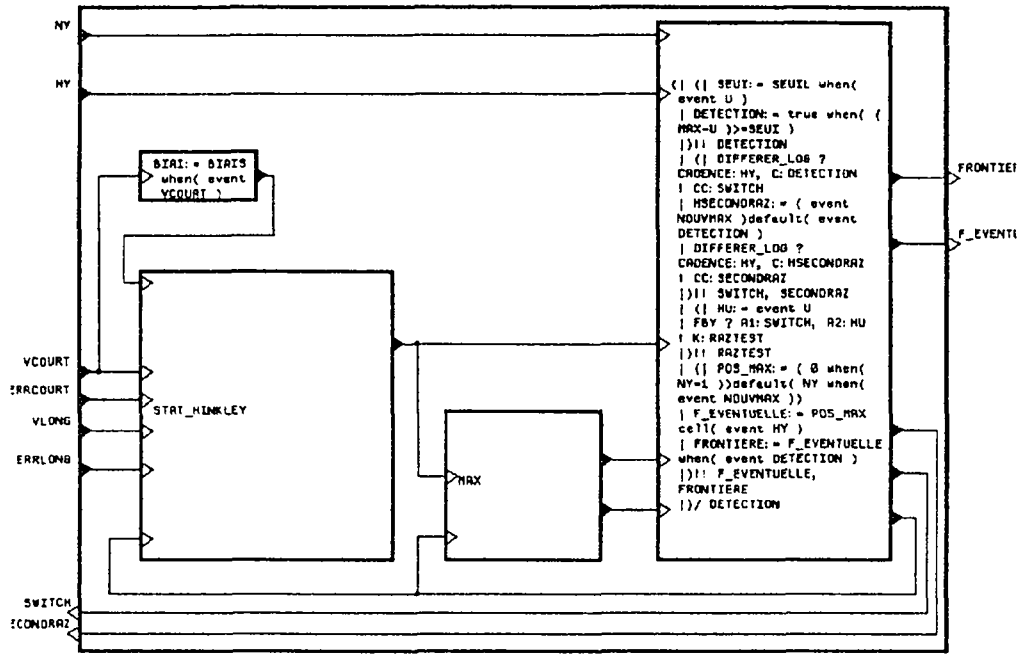

Figure IV.11 : Les trois phases effectuées par le processus primaire. Cette boîte noire est représentée de façon textuelle sur la figure IV.10.

Le traitement des trois phases est identique à celui effectué dans *TEST_HINKLEY*. Toutefois en sortie le signal **DETECTION** est remplacé par le signal **FRONTIERE**: d'un point de vue horloge rien n'est changé puisque **FRONTIERE** n'est émis que lors d'une détection. **FRONTIERE** contient la valeur de localisation de la frontière, c'est à dire la valeur du point de reprise potentiel lors d'une détection.

Les calculs des signaux liés à la commutation et à la réinitialisation sont liés aux signaux de détection et de nouveau maximum, c'est pourquoi ils sont regroupés. Il s'agit de **SECONDRAZ** (réinitialisation du processus secondaire), **SWITCH** (vrai lors d'une commutation primaire/secondaire) et **RAZTEST** (réinitialisation des tests).

Le processus *DIFFERER* (annexe B.12) diffère d'un top d'horloge les valeurs d'un signal C par rapport à une horloge plus rapide appelée **CADENCE**. Le signal **DETECTION** est différé en **SWITCH** et le signal **HSECONDRAZ**, vrai lors d'une détection ou lors d'un nouveau point de reprise potentiel, est différé en **SECONDRAZ**. Ainsi **SWITCH** et **SECONDRAZ** sont émis dès le top d'horloge suivant la détection ou le nouveau point de reprise potentiel.

```
(| DIFFERER_LOG ?cadence:hy, c:detection !cc:switch
 | hsecondraz:= ( event nouvmax) default ( event detection)
 | DIFFERER_LOG ?cadence:hy, c:hsecondraz !cc:secondraz
 |)!!switch,secondraz
```

Le signal **RAZTEST** est produit lorsque le test devient actif, c'est à dire lors de la première émission du signal **U** après une détection. On ne peut pas utiliser pour cela le compteur **NY** à cause de la commutation. Cela necessite de "reporter" le signal de réinitialisation au prochain événement **U** (processus *FBY* décrit en annexe B.10):

```
(| hu:= event u
 | FBY ?a1:switch, a2:hu !k:raztest
 |)!!raztest
```

IV.2.2 Le processus *MODULE_FILTRE*

Ce processus effectue le test de Hinkley sur le signal **PAROLE** précédemment filtré. Deux modules le composent:

- un module *FILTRAGE* rendant en sortie le signal filtré **YFILT**. Il prend en entrée un signal **Y**
- un module *HINKLEY_OPT* effectuant le test de Hinkley sur un signal **Y** (pas nécessairement filtré).

Dans notre cas les deux modules sont assemblés comme le montre la figure IV.12.

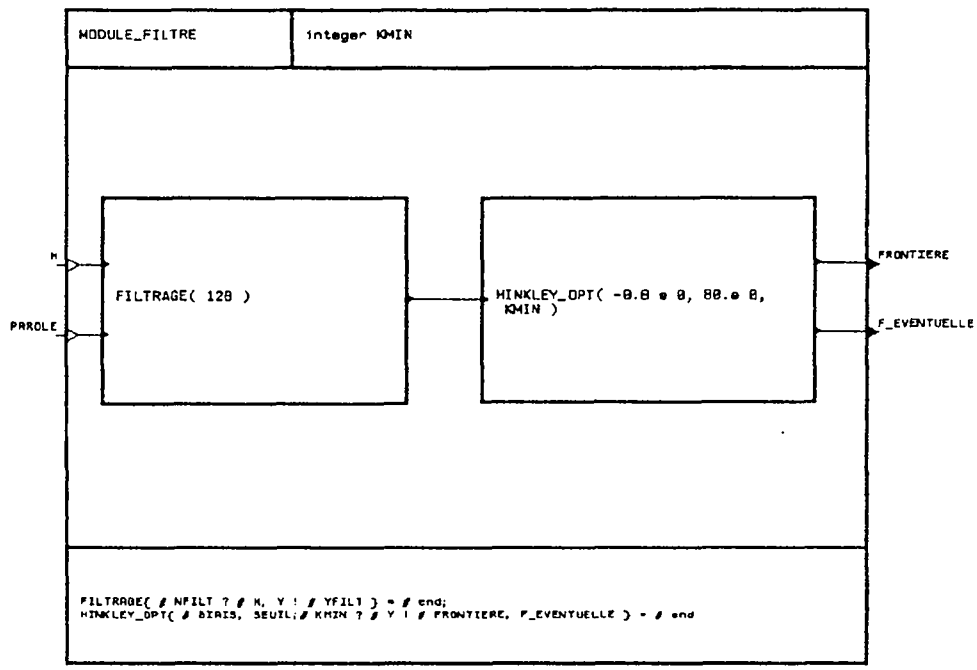

Figure IV.12 : Le processus *MODULE_FILTRE*

Le signal d'entrée **Y** de *FILTRAGE* est connecté à l'entrée **PAROLE**. Tandis que le signal d'entrée **Y** de *HINKLEY_OPT* est connecté à la sortie de *FILTRAGE* réalisant ainsi le test de Hinkley sur le signal filtré.

Le module de Hinkley utilisé ici est *HINKLEY_OPT* car le "module filtré" est autonome, gérant lui-même les reprises forward. Ainsi à l'extérieur de ce module, la gestion est transparente.

Le module *FILTRAGE*

Ce processus filtre le signal de parole d'entrée **Y** (figure IV.13). Cela consiste en un appel au processus standard *CONVOLUTION_LINEAIRE* (annexe A.10). Le signal de sortie **YFILT** est synchrone à **Y**.

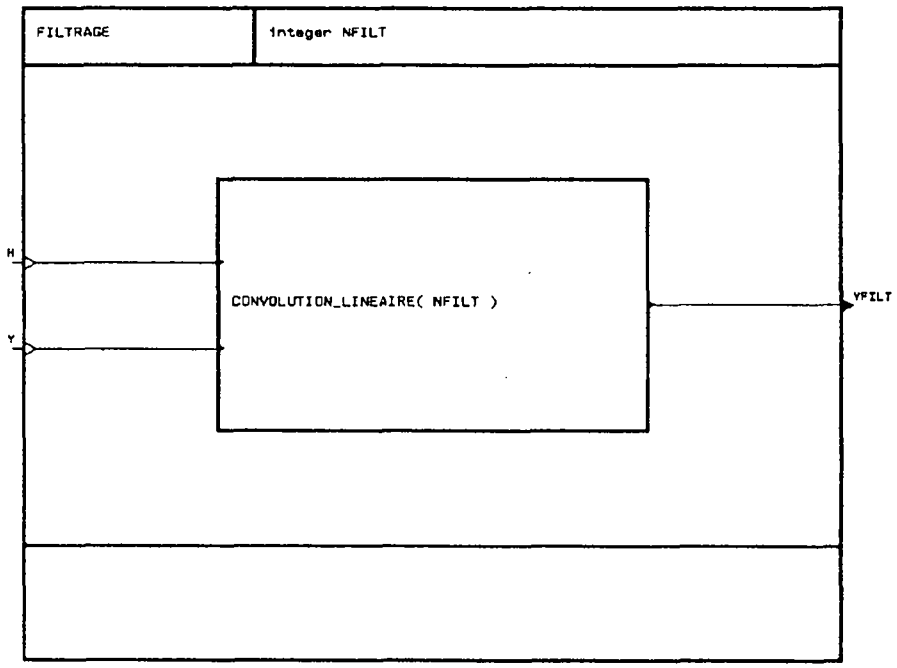

Figure IV.13 : Le processus *FILTRAGE*

Le paramètre **NFILT** indique que **YFILT** dépend des **NFILT** échantillons **Y** précédents. **H** est la réponse impulsionnelle.

IV.2.3 Le processus *MODULE_FORWARD_BACKWARD*

Ce processus segmente le signal de parole par la méthode de divergence forward-backward décrite au chapitre 2. Il est construit de façon modulaire et hiérarchique en tenant compte des caractéristiques temporelles de l'algorithme (figure IV.14). Il comprend quatre modules principaux:

- un module de contrôle *GESTION_MEMOIRE* gèrant le suréchantillonnage du signal de parole et la fenêtre glissante dans laquelle il est stocké. cette gestion distingue trois "périodes":
  - la période forward sans reprise. Dans ce cas les trois processus *MODULE_FILTRE*, *TEST_HINKLEY* et *TEST_VOISEMENT* s'exécutent en parallèle
  - la période forward avec reprise. Dans ce cas, le signal de parole n'est pas présent et est réinjecté. Seuls *TEST_HINKLEY* et *TEST_VOISEMENT* s'exécutent en parallèle[1]
  - la période backward. Dans ce cas, seul le test de Hinkley s'exécute

  En SIGNAL, les tests de Hinkley pour les périodes forward et backward peuvent être représentés par le même module, ici *TEST_HINKLEY*
- un module de traitement de données effectuant le test de Hinkley. Le module de Hinkley utilisé ici est *TEST_HINKLEY* car la réinitialisation et la gestion des reprises sont externes. Cela est du au fait que la localisation dépend de trois tests
- un module de traitement de données *TEST_VOISEMENT* segmentant le signal en des zones voisées, non voisées ou de silence. Il prend en entrée le signal **Y** et traite ce signal par blocs glissants
- un module de contrôle *GESTION_INFO* traitant les données issues des trois tests et communiquant les longueurs de segments au module de gestion de la mémoire.

[1] L'horloge du signal d'entrée **PAROLE** est liée à l'intérieur du processus *GESTION_MEMOIRE* puisque ce signal est suréchantillonné, fixant ainsi l'horloge de ce signal. Par conséquent, à l'extérieur de *MODULE_FORWARD_BACKWARD* l'horloge de ce signal n'est plus libre. Cela ne pose pas de problème au processus *MODULE_FILTRE* car celui-ci consomme le signal de parole au fur et à mesure de son arrivée, sans tenir compte des tops d'horloge pouvant intervenir entre deux échantillons. Cela explique le fait que *MODULE_FILTRE* soit inactif pendant les périodes de reprise. Cette phase de contrôle d'inactivité est donc totalement transparente au programmeur.

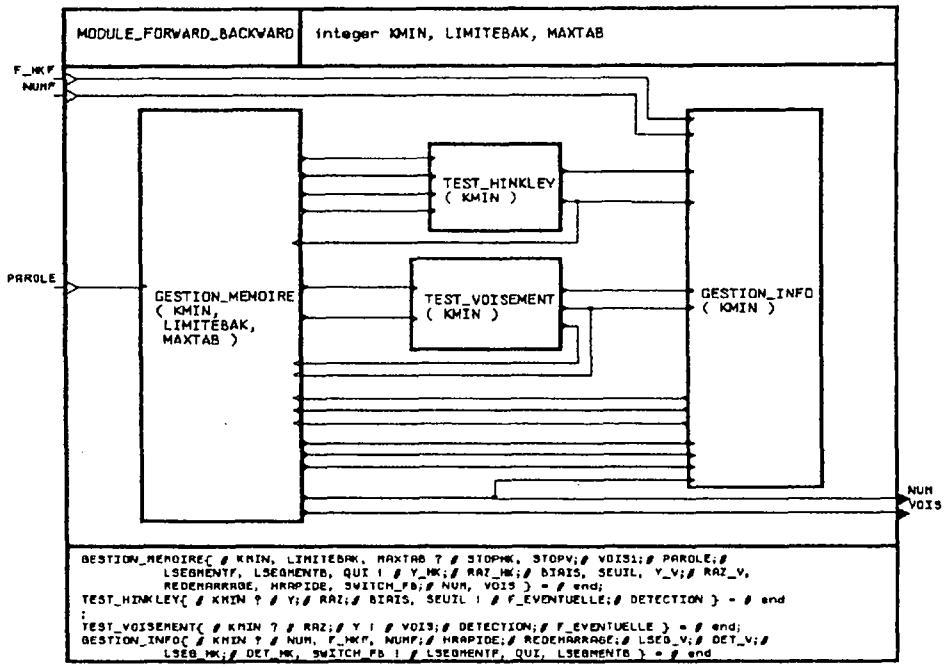

Figure IV.14 : Le processus *MODULE_FORWARD_BACKWARD*

Le module *GESTION_MEMOIRE*

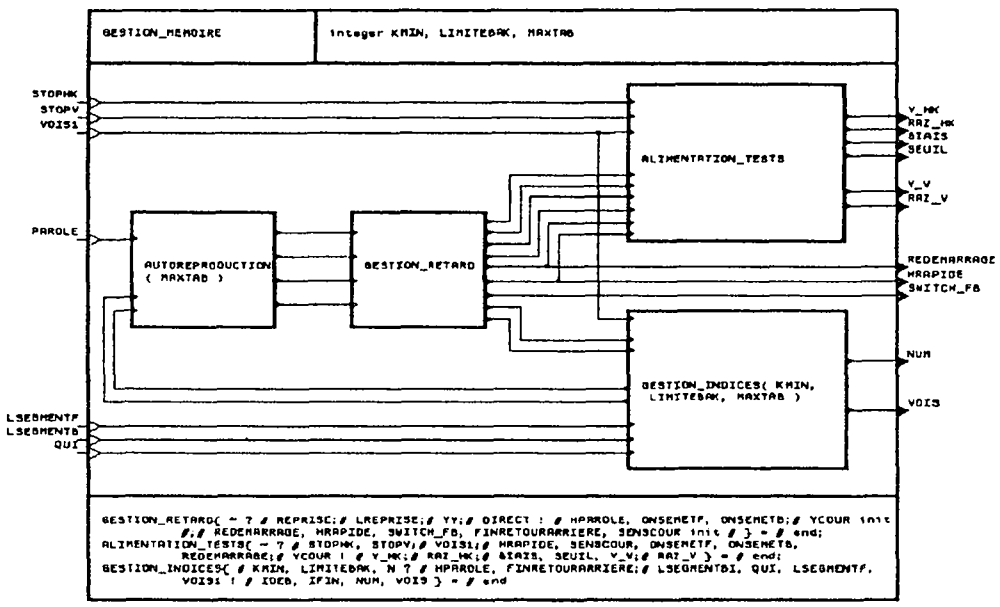

Figure IV.15 : Le processus *GESTION_MEMOIRE*

Ce processus est construit à partir de quatre modules (figure IV.15):

- le module *AUTOREPRODUCTION* (annexe B.13) suréchantillonne le signal **PAROLE**, stocké dans une fenêtre glissante. Une période de reprise démarre lorsque les indices d'entrée **IDEB** et **IFIN** sont présents. Le processus réinjecte alors les échantillons de la fenêtre glissante compris entre **IDEB** et **IFIN**. **PAROLE** sera de nouveau présent lorsque la réinjection sera terminée[1].
- le module *GESTION_RETARD* retarde les signaux issus de *AUTOREPRODUCTION* d'un top d'horloge par rapport à l'horloge la plus rapide (l'horloge du signal suréchantillonné). Il est indispensable de retarder d'un top d'horloge les décisions prises par les tests de Hinkley et de voisement car ce sont ces décisions qui permettent d'alimenter ces mêmes tests
- le module *GESTION_INDICES* calcule les indices et les frontières forward-backward à partir des longueurs des segments
- le module *ALIMENTATION_TESTS* alimente les entrées des tests de Hinkley et de voisement à partir des signaux retardés de *GESTION_RETARD*

[1]une reprise est dite forward (resp. backward) lorsque **IDEB**<**IFIN** (resp. >). Une reprise peut démarrer pendant une reprise: il suffit de recevoir **IDEB** et **IFIN** pendant la réinjection

Le processus *GESTION_RETARD* .
Les signaux d'entrée sont calculés un top d'horloge avant d'être utilisés:

- **REPRISE**: vrai si on se trouve en période de reprise, faux sinon
- **LREPRISE**: nombre d'échantillons restant à réinjecter lors de la reprise
- **YY**: signal de parole suréchantillonné
- **DIRECT**: sens de la reprise (synchrone à **IDEB** et **IFIN**).

Les signaux **YY**, **REPRISE** et **LREPRISE** sont synchrones et possèdent l'horloge la plus rapide.

Les signaux de sortie sont les signaux courants diffusés pour l'alimentation des tests, retardés d'un top sur l'horloge la plus rapide:

- **HPAROLE**: vrai lorsqu'on n'est pas en période de reprise (courante)
- **ONSEMETF**: vrai lors d'un redémarrage en mode forward
- **ONSEMETB**: vrai lors d'un redémarrage en mode backward
- **YCOUR**: signal suréchantillonné courant
- **REDEMARRAGE**: vrai lors d'un redémarrage
- **HRAPIDE**: horloge du signal suréchantillonné
- **SWITCH_FB**: vrai lors d'un redémarrage en mode forward (resp. backward) alors qu'on se trouvait en mode backward (resp. forward)
- **FINRETOURARRIERE**: vrai lorsque le dernier échantillon est réinjecté pendant le retour arrière
- **SENSCOUR**: sens courant (vrai pour forward et faux pour backward)

Pour retarder et calculer ces signaux, on utilise des processus déjà décrits (figure IV.16).

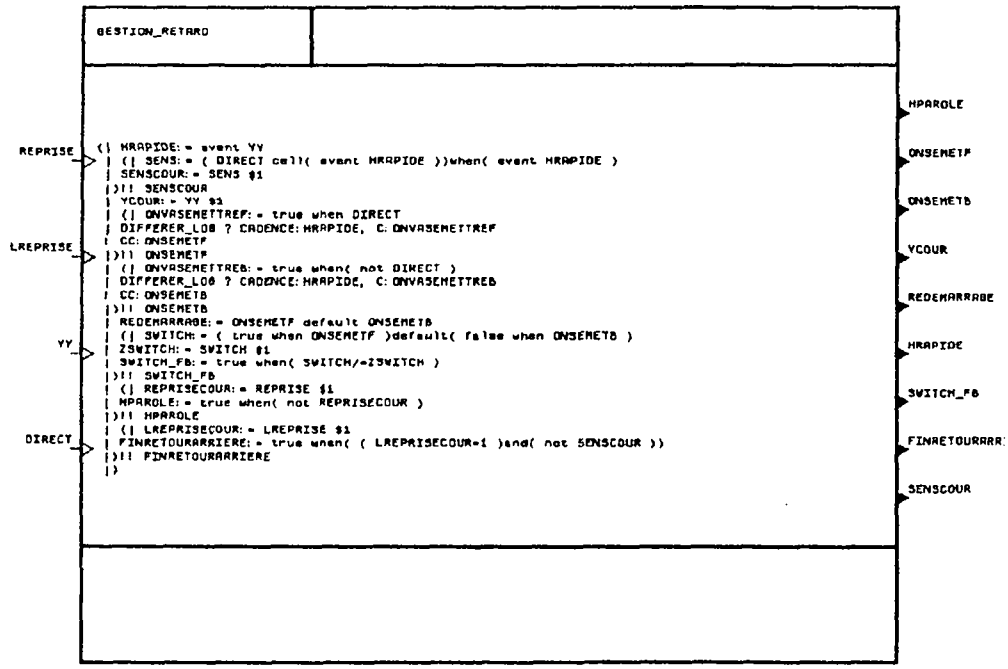

Figure IV.16 : Le processus *GESTION_RETARD*

Le processus *GESTION_INDICES.*
La longueur du segment forward **LSEGMENTF** et la décision de voisement **VOIS1** associée à ce segment permet de décider ou non d'un retour arrière:

```
(| okvoise:= vois1=2      % okvoise est vrai lorsque le segment est voise.
 | hlsegmentf:= event lsegmentf
 | RESYNCEVENT3 ?yin:okvoise,sync:hlsegmentf !yout:voise
     % okvoise est resynchronise a l'horloge de lsegmentf.
 | voiseprec:= vois $1
 | lmin:= (((4*kmin) when ((qui=1) or voiseprec)) default (2*kmin))
 |      when ( event lsegmentf)    % la longueur minimale de deux seg-
     % -ments adjacents depend de la decision de voisement du segment
     % precedent et du test qui a provoque la d
 | okb:= (lsegmentf≥lmin) and voise
     % lorsque lsegmentf est juge "trop long" et lorsque le segment
     % est voise, alors decision de retour arriere.
|)!! voiseprec,okb
```

[1] **QUI**=0 pour le test de Hinkley, **QUI**=1 pour le test de voisement, **QUI**=2 pour le test de Hinkley sur le signal filtré

Les indices **IDEB** et **IFIN** sont calculés différemment selon la période:

```
(| ideb:= irep  default ib  default ipdtb  default irepab
 | ifin:= jrep  default jb  default jpdtb  default jrepab
 | zideb:= ideb $1      % dernier ideb
 | zifin:= ifin $1      % dernier ifin
 |)
```

- **IREP, JREP**: indices de reprise forward après une période forward
- **IB, JB**: indices de reprise backward après une période forward
- **IPDTB, JPDTB**: indices de reprise backward pendant une période backward
- **IREPAB, JREPAB**: indices de reprise forward après une période backward

Le texte SIGNAL du calcul de ces indices ne pose pas de difficultés. Il se trouve en annexe avec le texte complet du programme.

Le calcul des frontières forward-backward est immédiat. Une frontière est validée à la fin d'un retour-arrière ou lors d'une détection forward sans retour-arrière.

```
(| ideb:= lsegmentvalide:= (lsegmentf  when (not okb))  default lsegmentb
 | znum:= num $1
 | num:= znum + lsegmentvalide
 | hnum:=  event num
 | RESYNCEVENT2 ?yin:vois1,sync:hnum !yout:vois
     % la decision de voisement est resynchronisee a l'horloge de num.
 |) !! num,vois
```

Le processus *ALIMENTATION_TESTS.*
Le test de Hinkley est alimenté par **YCOUR** et est réinitialisé à chaque **REDEMARRAGE**. Lorsque ce test détecte une frontière, les deux autres tests peuvent éventuellement continuer leurs traitements si *GESTION_INFO* attend les résultats de ces deux tests pour valider la frontière. Il faut dans ce cas interrompre le test de Hinkley. Cela est réalisé par le processus *INTERRUPT* (annexe B.11) et le signal **STOPHK** (connecté au signal de détection de la frontière de Hinkley). Les signaux **BIAIS** et **SEUIL** dépendent du sens courant et de la décision de voisement.
Le test de voisement est alimenté par **YCOUR** seulement dans le sens forward et est réinitialisé lorsque **ONSEMETF** est présent. De la même façon que pour Hinkley, il faut pouvoir l'interrompre.

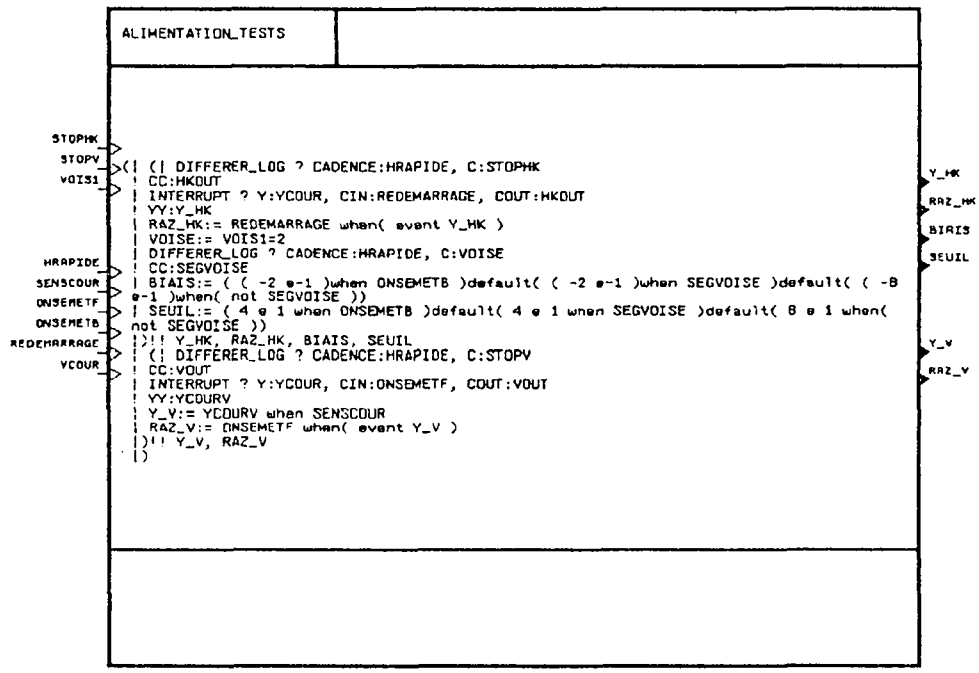

Figure IV.17 : Le processus *ALIMENTATION_TESTS*

Le module *TEST_VOISEMENT*

Ce processus effectue le test de voisement par blocs glissants sur le signal **Y**. Le test est réinitialisé de façon externe par le signal **RAZ** (figure IV.18).

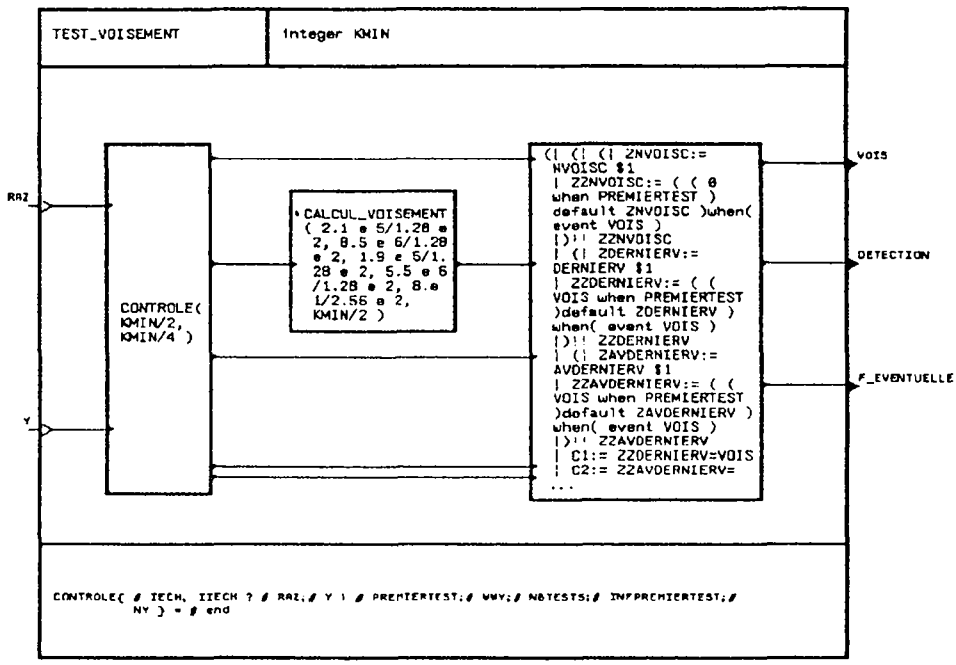

Figure IV.18 : Le processus *TEST_VOISEMENT*

Le processus *CONTROLE* stocke le signal **Y** dans une fenêtre glissante de longueur **IECH=KMIN**/2 et rend en sortie la fenêtre **WY** lorsque le test est actif (tous les **IIECH=KMIN**/4 échantillons). Il rend également en sortie d'autres signaux comme le montre le programme SIGNAL ci-après et la figure IV.19 d'entrelacements de quelques signaux. Le processus *COMPTEUR_MODULO_AFTER_RAZ* est décrit en annexe B.7.

```
(| (| hy:= event y
   | COMPTEUR_MODULO_AFTER_RAZ (iiech) ?cadence:hy
     !nmod:nymod % nb d'echantillons y depuis raz, modulo iiech=kmin/4
   | COMPTEUR_AFTER_RAZ ?cadence:hwy,raz:premiertest !n:nbtests
   |)/hy
 | (| actif:=(ny≠iiech) and (nymod=iiech) % le test est actif tous les
     % iiech echantillons des que ny=iech (fin de la periode d'initialisation)
   | wy:=y window iech
   | wwy:=wy when actif     % fenetre sous-echantillonnee a actif=vrai
```

```
    |)/actif,wy
  | infpremiertest:=ny<iech % indique si la periode d'initialisation est finie
  | (| hwwy:= event wwy
    | premiertest:=(ny=iech) when hwwy    % vrai lors du premier test
    | COMPTEUR_AFTER_RAZ ?cadence:hwwy,raz:premiertest !n:nbtests
      % nb de fois ou le test est actif depuis raz
    |)/hwwy
  |)
```

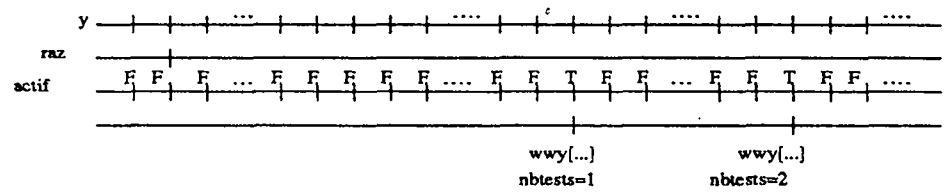

Figure IV.19 : L'entrelacement de signaux du processus CONTROLE

La fenêtre **WY** est ensuite analysée par le processus CALCUL_VOISEMENT (annexe A.9) qui rend en sortie une décision de voisement **VOIS**.

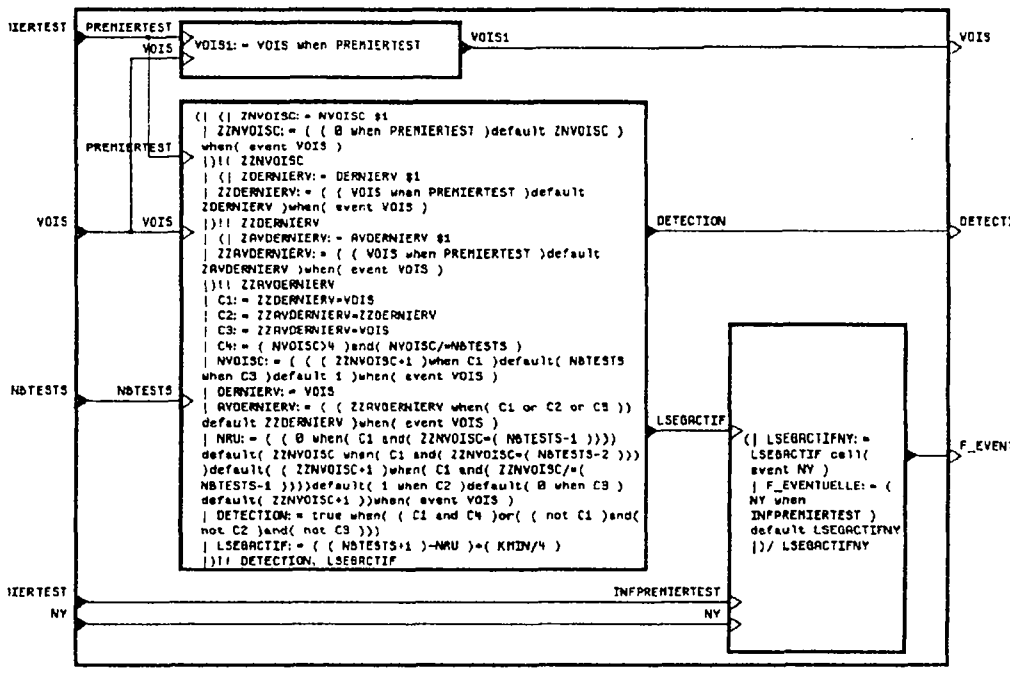

Figure IV.20 : La décision de voisement et la détection de frontières. Cette boîte noire est représentée de façon textuelle sur la figure IV.18.

La boîte noire traitant le signal **VOIS** rend en sortie la décision de voisement **VOIS** lors du premier test, le signal **DETECTION** lorsque le test de voisement détecte une frontière et la frontière éventuelle **F_EVENTUELLE** (figure IV.20).
La figure IV.21 nous donne les deux cas où le test de voisement détecte une frontière. Le cas (a) lorsque trois décisions différentes sont signalées par *CALCUL_VOISEMENT*. Le cas (b) lorsque deux décisions différentes sont trouvées, la deuxième portant sur plus de quatre blocs.
La longueur du segment **LSEGACTIF** est le nombre de blocs fois la taille d'un bloc (**KMIN**/4). **F_EVENTUELLE** est synchrone à **Y**. Elle vaut **NY** jusqu'au premier test, ou **LSEGACTIF** mémorisé à l'horloge de **Y**.

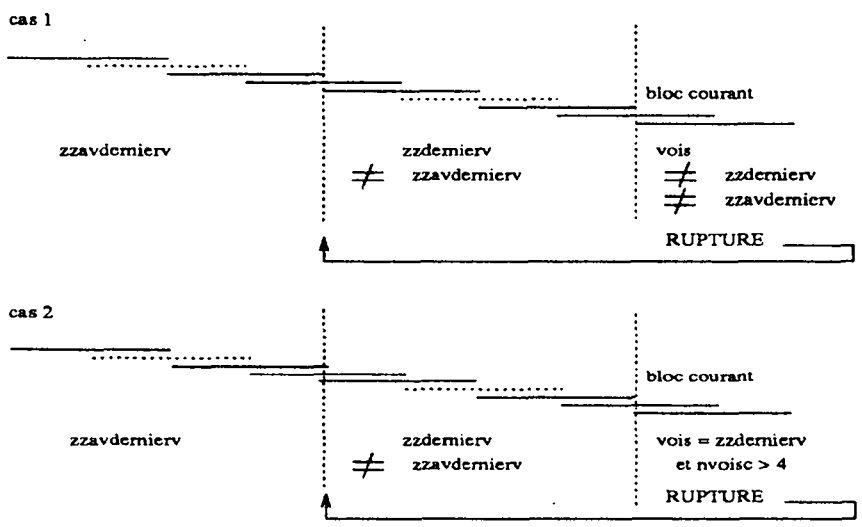

Figure IV.21 : La détection de frontières par le test de voisement. **NVOISC** est le nombre de blocs où la décision de voisement vaut **ZZDERNIERV**. Tous ces signaux sont synchrones à **WY** (test actif).

Le module *GESTION_INFO*

Ce processus gère l'information issue des modules de tests (figure IV.22).

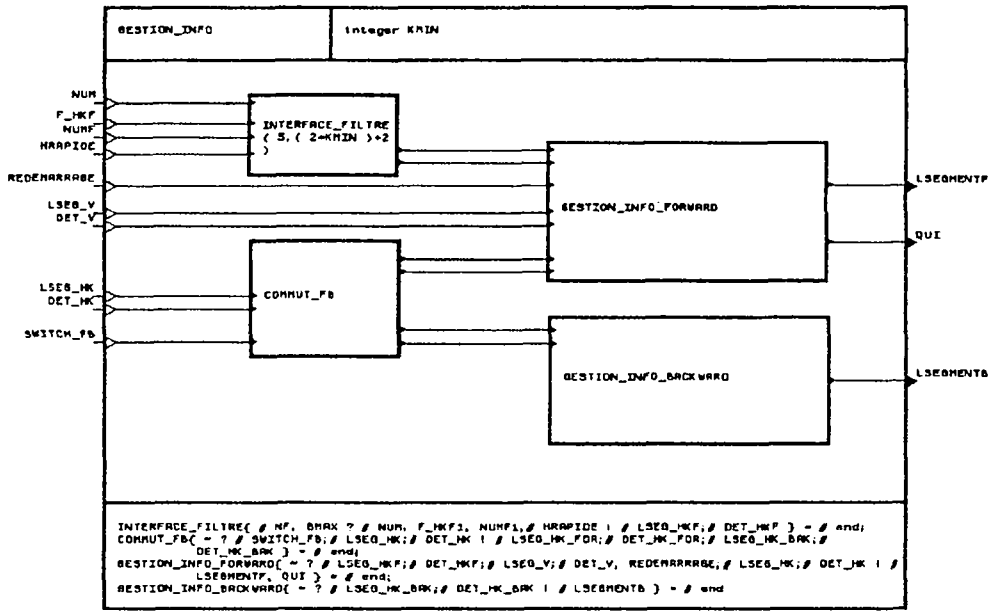

Figure IV.22 : Le processus *GESTION_INFO*

Le processus *COMMUT_FB* commute les signaux issus du test de Hinkley lors de passages forward-backward ou backward-forward. Ces signaux sont aiguillés vers le processus *GESTION_INFO_BACKWARD* lors d'un retour-arrière (**MAINFOR** présent et égal à faux) ou vers le processus *GESTION_INFO_FORWARD* lors d'un traitement en mode direct (**MAINFOR** présent et égal à vrai). Le processus *FLIPFLOP* (annexe B.9) est utilisé pour commuter les signaux.

```
(| (| FLIPFLOP ?switch:switch_fb !b:bascule,zb:zbascule /zbascule
   | hlseg_hk:= event lseg_hk
   | RESYNCEVENT3 ?yin:bascule,cadence:hlseg_hk !yout:mainfor
   | mainfor:=(bascule cell ( event lseg_hk)) when ( event lseg_hk)
   |)!! mainfor
 | lseg_hk_for:=lseg_hk when mainfor
 | det_hk_for:=det_hk when mainfor
 | lseg_hk_bak:=lseg_hk when (not mainfor)
 | det_hk_bak:=det_hk when (not mainfor)
 |)/mainfor
```

Le processus *GESTION_INFO_BACKWARD* filtre **LSEG_HK_BAK** lors d'une détection **DET_HK_BAK** pendant un retour-arrière:

```
lsegmentb:=lseg_hk_bak  when det_hk_bak
```

Le processus *INTERFACE_FILTRE* rend comparable les résultats du module filtré, qui rend des valeurs absolues, avec les autres modules, qui rendent des valeurs relatives au dernier **NUM.** Il doit tout d'abord différer les sorties du module filtré (renommées ici **NUMF1** et **F_HKF1**) d'un top par rapport à **HRAPIDE** afin de les resynchroniser avec les sorties des deux autres tests. Les frontières du module filtré sont ensuite alignées sur celles des autres tests:

```
(| DIFFERER_INT ?cadence:hrapide,c:numf1 !cc:numf
 | DIFFERER_INT ?cadence:hrapide,c:f_hkf1 !cc:f_hkf
 | DIFFERER_INT ?cadence:hrapide,c:num !cc:dnum
     % le num qui nous interesse pour la comparaison est le dernier
     % num valide, et non pas le prochain. "num $ 1" ne convient pas
     % puisqu'on ne sait pas encore si num va etre present.
     % Il faut donc differer num par rapport a hrapide
 | hredemarrage:= event dnum
 | (| wnumf:=numf window nf
    | RESYNCEVENT5(nf) ?win:wnumf,sync:hredemarrage !wout:wwnumf
    | zero:= 0  when ( event hredemarrage)
    | synchro hredemarrage,val
    | array i to nf
      of val:=(wwnumf  when (wwnumf>(dnum+bmax+1)))  default val
      with val[.]:zero,wwnumf
      end
    | frontieref:=val[1]
    | presentf:=:=val[1]≠0    % lors d'un redemarrage, on recherche, si elle
      % existe, la plus petite frontiere numf superieure a dnum+bmax+1.
   |)!!frontieref,presentf
 | (| mminnumf:=dnum+bmax+1
    | hnumf:= event numf
    | RESYNCEVENT2 ?yin:mminnumf,sync:hnumf !yout:minnumf
    | det_hkf:= presentf  default (not ( event dnum))
          default numf>minnumf
      % si la recherche a echoue lors d'un redemarrage, il faut reinitialiser
      % det_hkf a faux. Une nouvelle detection n'est prise en compte,
      % que si elle est superieure a dnum+bmax+1.
   |)!!det_hkf
```

```
| (| hf_hkf:= event f_hkf
   | RESYNCEVENT3 ?yin:det_hkf,sync:hf_hkf !yout:ddet_hkf
   | RESYNCEVENT2 ?yin:dnum, sync:hf_hkf !yout:ddnum
   | lseg_hkf:= (frontieref-dnum when presentf)
        default (0 when (not presentf))
        default (f_hkf-ddnum when (not ddet_hkf))
     % la valeur de la frontiere eventuelle filtree comparee aux
     % autres tests est une valeur relative a dnum.
   |)!!lseg_hkf
|)/hredemarrage
```

Le processus *GESTION_INFO_FORWARD* analyse les sorties **LSEG_HK**, **LSEG_V** et **LSEG_HKF** des trois tests, ainsi que les signaux de détection **DET_HK**, **DET_V** et **DET_HKF** présents lors d'une détection d'un test.

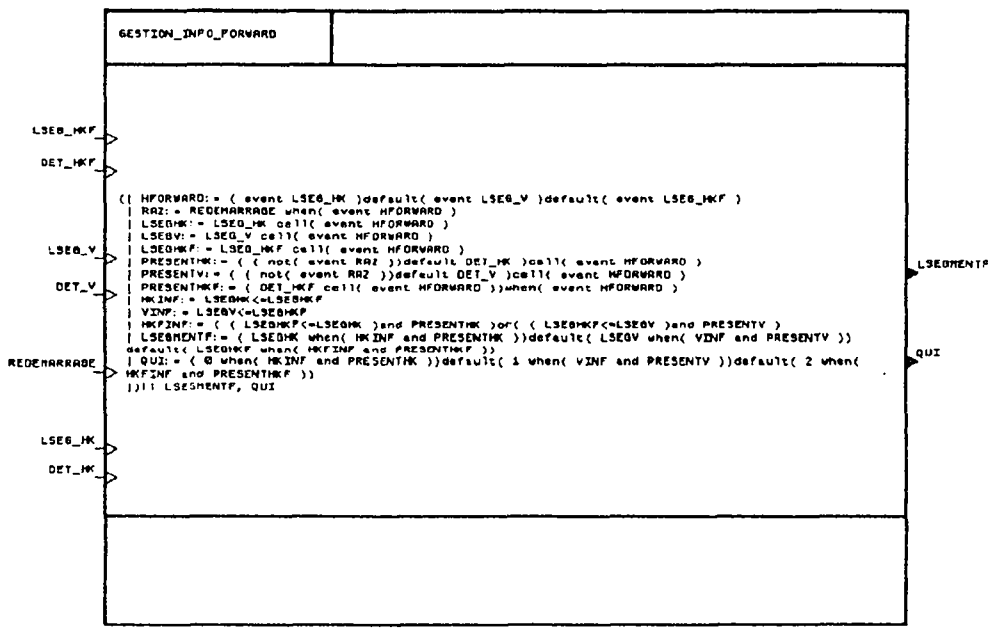

Figure IV.23 : Le processus *GESTION_INFO_FORWARD*

Le signal **QUI** indique lequel des trois tests a été à l'origine de la détection. La décision de valider une frontière dépend des situations courantes dans les autres modules. Toutefois, le texte SIGNAL est assez souple pour tester sans difficultés diverses façons de valider. Il suffit de modifier **HKINF**, **VINF** ou **HKFINF** pour privilégier certains tests par rapport à d'autres. Dans notre cas, par exemple, il suffit que la frontière éventuelle de *TEST_HINKLEY* soit inférieure à celle du module filtré pour qu'elle soit validée.

IV.3 Analyse des résultats

IV.3.1 Mise au point du programme

En ce qui concerne la mise au point des programmes, il faut distinguer:

- la mise au point du comportement temporel du programme: les erreurs de synchronisation sont détectées par le compilateur SIGNAL
- la mise au point du programme lui-même à partir du code produit par le compilateur: elle est énormément réduite du fait de la mise au point précédente

Comportement temporel

Le compilateur SIGNAL vérifie statiquement et de façon complète le comportement temporel du programme (absence de blocage...) grâce à un mécanisme de calcul d'horloges qui transforme un programme en un système d'équations.
En effet l'analyse statique effectuée permet notamment de prouver la correction des synchronisations exprimées dans les algorithmes en résolvant un système d'équations construit sur le corps $Z/3Z$ (entiers modulo 3).
D'un point de vue utilisateur, les résultats de cette analyse apparaîssent sous deux formes:

- un fichier de même nom que le programme SIGNAL suffixé par "tra". Il permet l'analyse des erreurs de synchronisation (égalités d'horloges non prouvables...); par exemples contraintes sur les valeurs de variables externes, programme non déterministe...
- un fichier de même nom que le programme SIGNAL préfixé par la lettre "D". Il est produit lors de la détection de cycles (signaux→horloges, signaux→signaux ou horloges→signaux) et permet de retrouver aisément l'origine d'un cycle (ensemble circulaire de définitions).

Résultats numériques

A ce stade les synchronisations sont correctes puisque la mise au point précédente a été effectuée. La question qui se pose est la nécessité de développer un outil spécial pour la mise au point du programme généré par le compilateur. Dans notre cas nous avons utilisé le langage SIGNAL lui-même pour mettre au point un programme SIGNAL:

- la transformation d'un signal interne en port de sortie du programme, laissant ainsi une trace de ce signal pendant l'exécution. En effet les valeurs du signal sont alors stockées dans un fichier du nom du signal préfixé par la lettre "W". Ces transformations sont aisées grâce à l'interface graphique

- l'utilisation d'outils SIGNAL pour la mise au point pendant l'exécution du programme: visualisation de courbes, suivi des horloges de certains signaux, etc. Ce point est développé dans le chapître suivant. Les techniques que nous avons développées devraient être intégrées dans un environnement pour obtenir un véritable débogueur.

IV.3.2 Facilités de programmation

Nous appelons "programme FORTRAN d'origine", le module de segmentation acoustique développé par l'équipe PAROLE à l'IRISA. Ce programme, écrit en langage séquentiel FORTRAN, effectue le même traitement que le programme SIGNAL appelé *MODULE_SEGMENTATION*.

L'algorithme en lui-même est conçu de façon séquentielle. La structure très simplifiée de l'algorithme est la suivante:

```
filtrage de tout le signal de parole;
calcul du premier segment filtre (r_filtre);
REPETER TANT QUE pas_fin_de_tableau
    <initialisations>;
    FAIRE JQA detection
        tous les kmin/4 echantillons: test de voisement;
            SORTIR PAR detection (lors d'une detection de frontiere r)
        test de divergence Hinkley;
            SORTIR PAR detection (lors d'une detection de frontiere r)
        incrementer l'indice de tableau;
    REFAIRE
    SORTIE detection:
      rinf:=pluspetit(r,r_filtre)
      si rinf<kmin alors retour-arriere (resultat: frontiere r') fsi
      si r_filtre < r' alors chercher le prochain segment filtre fsi
FREPETER
```

La gestion des données dans le programme d'origine consiste à stocker le signal de parole dans un tableau au début de l'exécution. Les inconvénients de ce choix sont que d'une part la taille du tableau doit être prévue très grande et d'autre part le signal de parole doit être stocké et connu avant l'exécution du programme.
Il est possible de gérer ces données de la même façon que le programme SIGNAL,

c'est à dire en les traitant échantillon par échantillon, mais dans ce cas il faut gérer la fenêtre glissante qui contient le signal de parole.

Nous pouvons noter quelques points qui nous paraîssent importants, sans oublier toutefois que ce programme a été conçu en vue d'une mise en œuvre séquentielle:

- le parallélisme entre les différents tests est inexistant
- la gestion du contrôle est entièrement à la charge du programmeur. Par exemple programmation et gestion des fenêtres glissantes (test de voisement), des périodes d'initialisations (test de Hinkley), des entrées/sorties...
- la décision de détection n'est pas prise au plus tôt. C'est à dire par exemple que les tests de hinkley ou de voisement poursuivent leurs traitements jusqu'à détection d'une frontière afin de comparer celle-ci à la frontière filtrée. Dans le cas où la frontière filtrée est choisie, ce traitement était en partie inutile
- il n'y a aucune structure hiérarchique et modulaire, les initialisations et traitements des différents tests étant par exemple répartis sur l'ensemble du programme

Il est difficile en FORTRAN de traiter les données au plus tôt car la gestion du contrôle devient alors contraignante. Cela explique la programmation ci-dessus. En règle générale, le programmeur FORTRAN choisi la programmation dans laquelle la gestion du contrôle est la moins importante. Dans le cas contraire il se trouve confronté à la programmation d'un véritable système de contrôle et retrouve alors les difficultés résolues par le compilateur SIGNAL tels que l'optimisation de l'indiçage des fenêtres glissantes, le traitement des vecteurs, etc.
De plus le fait qu'il n'y ait aucune structure hiérarchique et modulaire pénalise le concepteur désirant tester différents modules ou par exemple rajouter un test supplémentaire.

IV.3.3 Code obtenu

Le compilateur SIGNAL génère du code FORTRAN, simulant ainsi une exécution synchrone. Avant de donner les chiffres des exécutions du programme SIGNAL et du programme d'origine de l'équipe parole, il faut citer les différences entre les deux algorithmes:

- les entrées/sorties du programme SIGNAL ont été modifiées pour être adaptées à celles du programme séquentiel d'origine. Ainsi les entrées sont lues par blocs et stockées dans un tableau au début de l'exécution. Toutefois le programme SIGNAL reste désavantagé car la lecture d'un échantillon du signal de parole est réalisé par un appel à une subroutine. Or même si la subroutine a été modifiée pour lire dans un tableau au lieu d'un fichier, l'appel en lui-même est pénalisant en temps d'exécution

- le programme SIGNAL n'a pas été conçu pour une exécution séquentielle mais en vue d'une mise en œuvre sur des architectures parallèles. La remarque vaut en premier lieu pour le processus *HINKLEY_OPT*. La conception "temps-réel" de ce module autonome fait que les processus primaire et secondaire effectuent tous les deux la phase "estimation des modèles". Par conséquent, les calculs inhérents à l'estimation des modèles sont effectués en double (à l'ordre 4 sur le signal filtré), ce qui présente un gros désavantage pour le programme SIGNAL lors d'une exécution séquentielle sur un seul processeur.

Les tests qui suivent ont été réalisés sur une VAXstation 3200 sous VMS, dans les mêmes conditions d'utilisation; les temps indiqués sont les temps CPU, incluant les temps de SWAP. La phrase analysée est *"Il se garantira du froid avec ce bon capuchon"*.

	tests sur 15 blocs soit 3840 échantillons (0,3 s de parole)		toute la phrase: 170 blocs soit 43520 échantillons (3,4 s de parole)	
	ordre 2	ordre 16	ordre 2	ordre 16
programme SIGNAL[1]	22"	54"	4'37"	11'41"
programme SIGNAL[2]	17"	23"	3'34"	4'49"
programme FORTRAN	6"	11"	1'08"	2'09"

Tableau comparatif des temps d'exécution entre le programme SIGNAL (génération de code FORTRAN, version P1, simulant l'approche synchrone) et le programme séquentiel FORTRAN d'origine. L'ordre indiqué est l'ordre des modèles autorégressifs *AUTO* et *BURG*.

[1] *AUTO* et *BURG* sont écrits en SIGNAL.
[2] *AUTO* et *BURG* sont laissés en langage FORTRAN d'origine.

La deuxième ligne du tableau indique les temps d'exécution pour le programme SIGNAL dans lequel nous avons laissé en FORTRAN les deux modules d'identification des modèles autorégressifs. Pour plus de détails sur ces possibilités offertes par SIGNAL, voir [16] . Il faut tout de même savoir que la possibilité est donnée au programmeur d'utiliser des modules entiers non nécessairement écrits en SIGNAL. Il suffit simplement de les déclarer en tant que processus externes, leur utilisation étant similaire aux autres processus.

Tailles et temps de compilation

	taille
programmes FORTRAN générés par le programme SIGNAL	142 blocs
programme FORTRAN d'origine	43 blocs

Tableau comparatif des tailles des fichiers FORTRAN. Un bloc correspond à 512 octets.

le programme SIGNAL	
programme textuel	135 blocs (1168 lignes)
programme graphique	817 blocs
temps de compilation	1'19" temps CPU

Taille des programmes SIGNAL et temps de compilation. Les blocs sont de 512 octets. Le temps de compilation est mesuré en temps CPU de façon identique aux précédents. Il comprend le calcul d'horloges, du graphe de dépendances et la génération du code FORTRAN.

IV.3.4 Analyse et perspectives

Il est important de rappeler que cette application n'avait pas pour but d'effectuer les traitements le plus rapidement possible, mais de décrire l'application entière en SIGNAL. La rapidité d'exécution d'un programme est liée au développement du langage SIGNAL et les améliorations qui en découlent sont laissées aux concepteurs du langage.

Pour conclure ce chapître. nous citons quatre points qui nous semblent importants:

- la version du compilateur du langage que nous avons utilisée est une version prototype appelée P1. Elle doit être incessamment remplacée par une version actuellement en cours de test et appelée H2. Dans cette nouvelle version, le contrôle des applications est organisé de façon hiérarchique ce qui permet une optimisation du code produit par le compilateur (imbrication de "if", optimisations supplémentaires...). Nous donnons à titre indicatif des chiffres sur un des exemples tests. L'exemple pris est un programme SIGNAL simulant une montre chronomètre. La vitesse de calcul de l'exécutable est multipliée par 2,5 (resp. 3) lorsque les codes FORTRAN, produits par les compilateurs des versions P1 et H2, sont compilés avec (resp. sans) l'option d'optimisation du compilateur FORTRAN.
  L'algorithme résolvant le système d'équations des horloges (c.a.d. le calcul d'horloges) est lui-même plus efficace ce qui entraîne des temps de compilation moins importants. Cette efficacité est liée au fait que la résolution est basée sur la notion de profondeur (hiérarchie). Sur le même exemple test les temps de compilation pour le calcul d'horloges sont passés de 18 mn (P1) à 1 mn 45 sec (H2), soit un gain de facteur 10 pour le calcul d'horloge proprement dit. Le gain est ramené à un facteur 4 après la production de code (cela est lié au parcours en vue d'optimisation)
  La hiérarchisation du code permet de réduire ce code. Sur le même exemple, le sous-programme FORTRAN engendré occupe 200 blocs VMS (100 KO) (version P1) et 127 blocs (64 KO) (version H2), soit un gain de 37 % en encombrement

- les communications (entrées/sorties...) pourraient être modifiées en vue d'une mise en œuvre séquentielle très optimisée. Il est en effet aisé par exemple de remplacer les appels d'entrée/sorties par la lecture dans un tableau et d'optimiser les indexations sur ce tableau

- les temps d'exécution sont énormément réduits lorsque l'on utilise les modules FORTRAN pour identifier les modèles. Ces processus ne comportent pas de phase de contrôle, mais manipulent essentiellement des vecteurs et des tableaux réguliers de processus. Si la version H2 doit permettre d'améliorer ces temps, une analyse plus approfondie reste à faire en ce qui concerne les traitements sur vecteurs et la possibilité d'optimiser ces traitements

- enfin une étude est actuellement en cours pour réduire le rapport taille du programme graphique/taille du programme textuel.

Chapitre V

Développements en SIGNAL autour du module de segmentation

L'approche synchrone permet de traiter les échantillons du signal de parole au fur et à mesure de l'exécution du programme. De la même façon, elle permet d'émettre des résultats dès leur production pendant cette exécution. L'idée de ce chapître est d'exploiter ces possibilités pour offrir à l'utilisateur un ensemble d'outils qui vont lui permettre de visualiser des résultats (intermédiaires ou finals) ou d'intervenir manuellement pendant l'exécution du programme.

Pour ce faire, une boîte à outils a été développée (annexe 3). Toutefois elle reste insuffisante et sera prochainement complétée. L'environnement autour du module de segmentation acoustique a été développé sous le système de fenêtrage SunView sur SUN; l'interface graphique étant actuellement développé sous SunView (il doit être prochainement porté sur X_WINDOWS).

V.1 Observations sans modifier l'exécution d'un programme

Nous appelons "programme source", un programme SIGNAL classique tel qu'il a été défini dans le chapitre précédent. Dans notre cas, il s'agit du programme *SEGMENTATION* dont le texte est en annexe 4.
Nous appelons "programme source modifié", le programme source auquel ont été rajoutés des processus liés à l'exécution du programme (affichage...).

Le processus *SONDE*.
Nous appelons processus *SONDE*, un processus SIGNAL ayant un seul signal en entrée et ne possédant pas de signaux de sortie. Un tel processus peut être attaché à un signal du programme source sans modifier l'exécution de ce programme.
Nous pouvons bien sur greffer dans le programme source autant de processus *SONDE* que de signaux. L'interface graphique permet de visualiser aisément de tels processus (figure V.1).

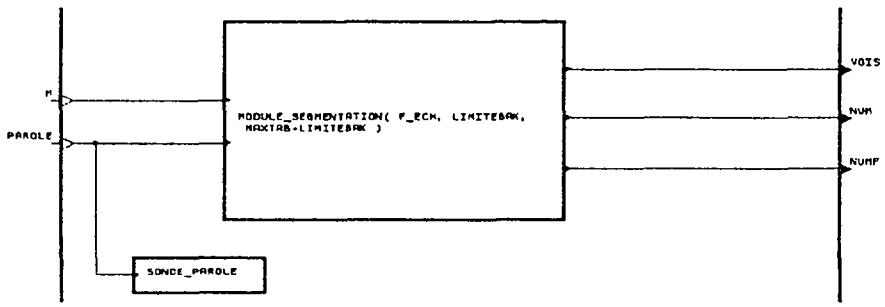

Figure V.1 : Le processus *SONDE_PAROLE* est associé au signal de parole.

Le programme *AFFICHAGE*.
En parallèle, nous développons en SIGNAL un programme *AFFICHAGE* indépendant associé au programme source. Les ports d'entrée de ce processus sont les signaux associés à tous les processus *SONDE* du programme source modifié. Il n'existe pas de ports de sortie.
Ce processus est créé pour visualiser "en temps réel" les résultats de l'exécution du programme; par exemple afficher des courbes, visualiser des frontières ou des horloges de signaux, gérer l'écran (par exemple de façon circulaire), etc. Afin de faciliter la programmation de ce processus, une bibliothèque d'outils SIGNAL est développée (annexe 3). Ces processus utilisent des procédures externes d'affichage.

Application au programme *SEGMENTATION*.
Dans la version du langage que nous avons utilisée, la compilation séparée n'est pas autorisée. Toutefois, afin de mettre en œuvre les notions développées auparavant, et de ce fait "simuler" la compilation séparée, nous modifions:

- d'une part les interfaces des processus *SONDE*. Un processus *SONDE_X* met l'horloge de **X** à vrai lorsque X est présent, et à faux sinon
- d'autre part l'interface du processus *AFFICHAGE* afin de l'adapter aux processus *SONDE*.

La figure V.2 donne une vision globale du processus *AFFICHAGE*. Les boîtes noires connectées directement aux ports d'entrée seront inutiles dans la version acceptant la compilation séparée.

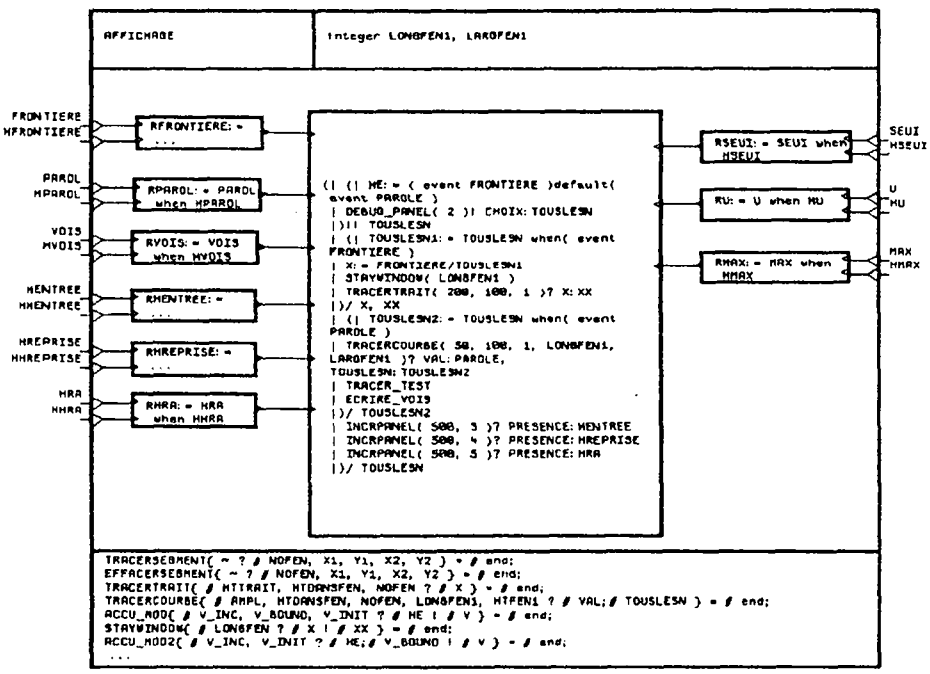

Figure V.2 : Le processus *AFFICHAGE*.

Le texte SIGNAL du processus *AFFICHAGE* visible sur la figure n'est pas expliqué car il se déduit d'une utilisation très simple des processus de la bibliothèque.

V.2 Interventions "en temps-réel" sur l'algorithme.

Les processus *DEBUG*. Nous appelons processus *DEBUG*, un processus prenant en entrée un signal interne ou externe d'un programme, et rendant en sortie le même signal sous-échantillonné. Nous pouvons considérer qu'un tel processus est attaché à un lien entre deux ports connectés (ou au port diffuseur du lien). Un tel processus nous permet de "couper" ou non le lien entre deux signaux pendant l'exécution du programme, et ainsi d'intervenir de façon synchrone sur ce lien.

L'interface graphique permet de visualiser un tel processus (figure V.3). L'exemple ci-dessous consiste à pouvoir interdire le retour-arrière pendant l'exécution du programme (signal **OKB** du processus *GESTION_INDICES*).

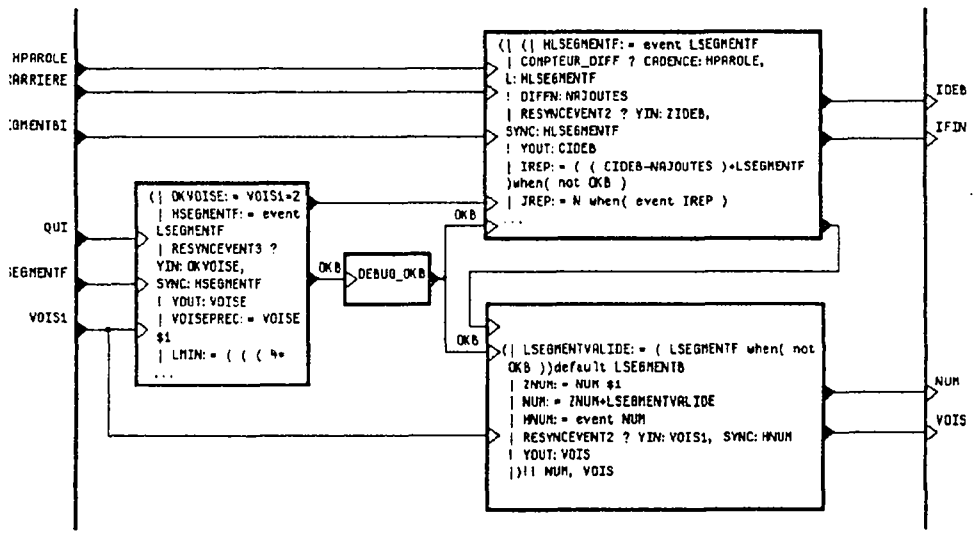

Figure V.3 : Le processus *DEBUG_OKB* est positionné entre le port diffuseur **OKB** et le port récepteur du lien. Cette figure représente un gros plan du processus *GESTION_INDICES* déjà présenté.

Les processus *DEBUG* sont insérés dans le programme source pour former une partie du programme source modifié. Ils sont développés en SIGNAL et utilisent la boîte à outils développée en annexe 3.
Dans la version comprenant la compilation séparée, ces processus *DEBUG* pourront être intégrés au programme *AFFICHAGE* qui possèdera alors des ports de sortie vers le programme source.

Un processus *DEBUG* peut appeler une procédure externe. Cette procédure, liée au clavier ou à la souris, nous permet d'intervenir sur le déroulement de l'exécution; par exemple autoriser ou refuser le passage de certains signaux, régler les seuils, autoriser ou refuser l'utilisation de certains modules, etc.

V.3 Autres outils.

D'autres outils peuvent être créés. Ils permettent également d'intervenir pendant l'exécution du programme et améliorer le "confort" de l'utilisateur. Très peu ont été développés, le travail restant à approfondir. Nous citons ici l'exemple du processus *PACE_MAKER* et du processus permettant de générer du signal homogène.

V.3.1 Le processus *PACE_MAKER*

Ce processus consiste à suréchantillonner ou souséchantillonner le signal d'entrée **HE**. Le signal de sortie est le signal **HES** (figure V.4).

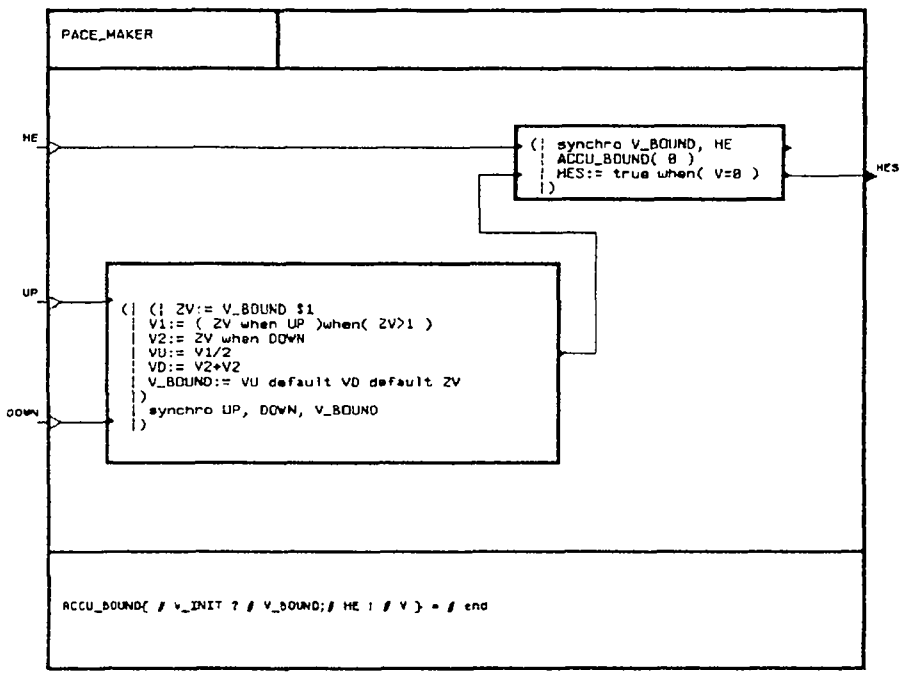

Figure V.4 : Le processus *PACE_MAKER*.

Le processus *ACCU_BOUND* est un compteur modulo dans lequel la valeur du modulo **V_BOUND** est un signal (annexe B.6).
La première boîte noire calcule le signal **V_BOUND** à partir des valeurs des signaux **UP** et **DOWN**. Le modulo est divisé par 2 lorsque le signal **UP** est présent et égal à vrai, et que le modulo était supérieur à 1. Le modulo est doublé lorsque le signal **DOWN** est présent et égal à vrai.
Le signal **HES** est produit lorsque le compteur *ACCU_BOUND* atteint la valeur

du modulo, les signaux **HE** et **V_BOUND** étant synchrones.
N.B: lorsque **UP** et **DOWN** ne sont jamais présents, le signal **HES** est produit à chaque événement **HE**.

Application au processus *SEGMENTATION.*
PACE_MAKER est utilisé pour ralentir ou accélérer l'exécution du programme. Il suffit pour cela de synchroniser l'horloge la plus rapide du programme (dans notre cas l'horloge du signal **LREPRISE**) avec le signal **HES**. Ainsi l'horloge de **LREPRISE** est imposée par les signaux **HE**, **UP** et **DOWN**.
Il suffit ensuite de relier les signaux **UP** et **DOWN** avec la souris par exemple, pour pouvoir ralentir ou accélérer l'horloge de **LREPRISE** manuellement pendant l'exécution du programme.

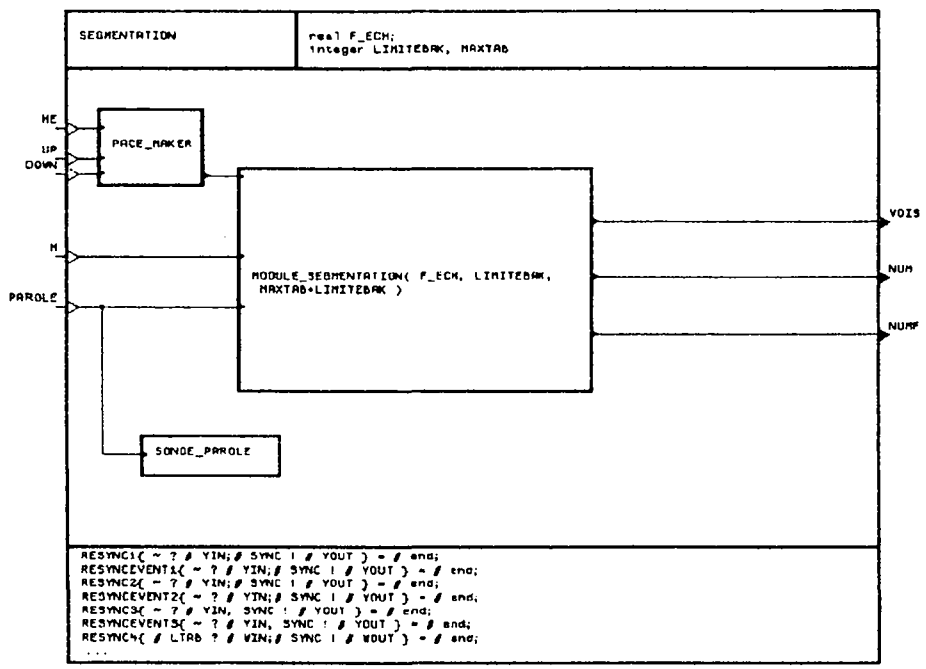

Figure V.5 : Une utilisation du *PACE_MAKER*.

V.3.2 Génération de segments homogènes du signal

Le processus *GENESIGNAL* génère des segments homogènes du signal (signal **Y**). Il est inséré de façon à connecter la sortie **Y** de *GENESIGNAL* à l'entrée **PAROLE** de *SEGMENTATION* (figure V.6).

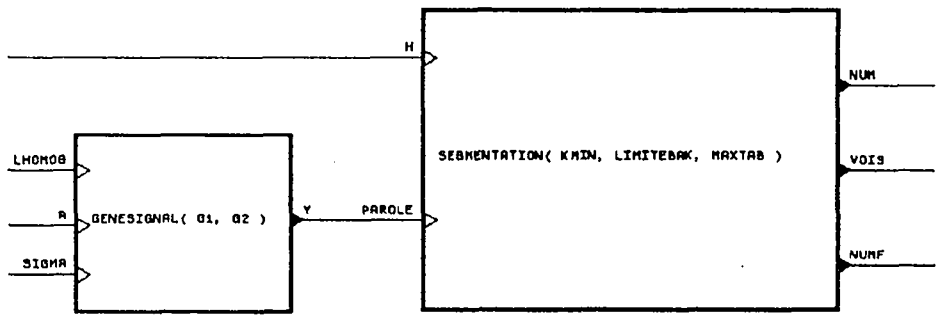

Figure V.6 : L'entrée **PAROLE** est générée par *GENESIGNAL*.

GENESIGNAL utilise un mécanisme de suréchantillonnage pour générer le signal **PAROLE** (processus *OVERSAMPLING*). *OVERSAMPLING* prend en entrée un signal **L**. Soit **HL** l'horloge de ce signal et v la valeur du signal **L** à un instant t. Le mécanisme de suréchantillonnage permet de créer (v-1) instants supplémentaires entre deux tops de **HL**, créant ainsi une horloge plus rapide **H'**. **H'** est l'horloge du signal de sortie **N** (figure V.7).

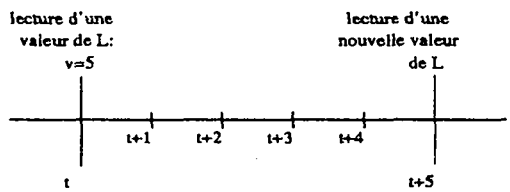

Figure V.7 : Exemple de suréchantillonnage avec v=5.

Le texte SIGNAL correspondant est le suivant:

```
(| synchro l,hl
 | n:=l default (zn-1)
      % n decompte l valeurs.
 | zn:= n $ 1
 | hl:= true when (zn=1)
 |)/zn,hl
```

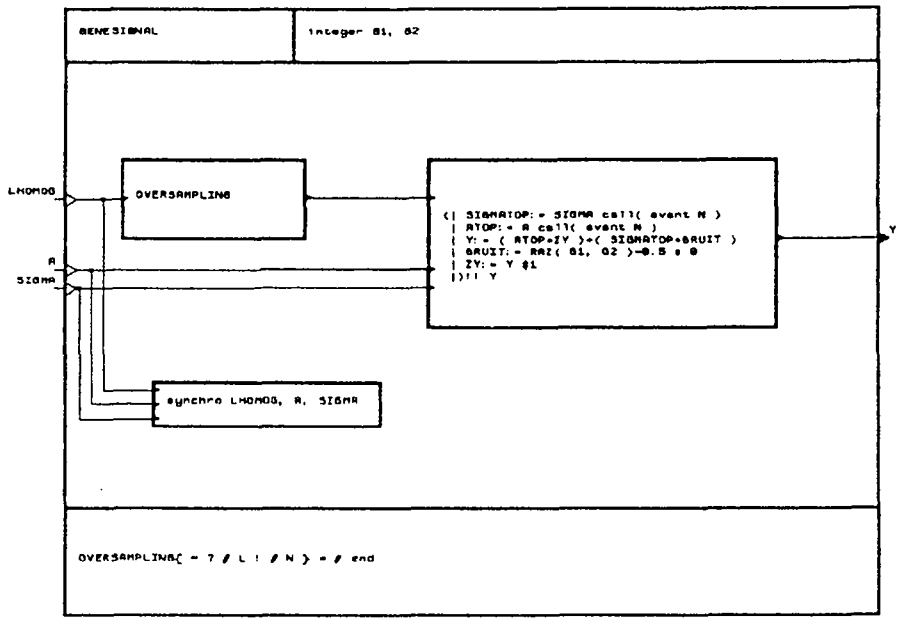

Figure V.8 : Le processus *GENESIGNAL*.

La création du signal **Y** est ensuite possible grâce à la fonction *RANDOM*, générateur de nombres aléatoires suivant une loi gaussienne (N(0,1)). Le signal **Y** peut correspondre à un processus autoregressif gaussien d'ordre p,

$$y_t = \sum_{i=1}^{p} a_i y_{t-i} + e_t \sigma \quad , \forall t \, ;$$

La fonction *RANDOM* génère un signal **BRUIT** (e_t); la valeur finale de **Y** est une fonction des signaux **BRUIT**, **A** (a_i), **SIGMA** (variance σ), et son horloge est celle de **N**.
Les instants de changements de modèles sont donnés par **LHOMOG**.

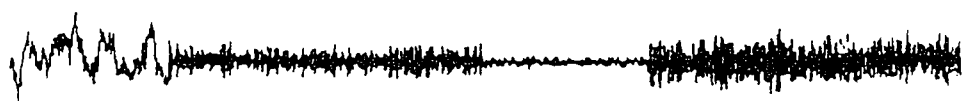

Figure V.9 : Exemple de signal généré pour un ordre p=1. Les valeurs prises dans cet exemple par **LHOMOG** sont (500,1000,500,1000) générant ainsi 3000 échantillons de signal. Les valeurs de **A** sont (0.95,-0.8,0.5,-0.7). Les valeurs de **SIGMA** sont (30.,25.,15.,50.).

V.4 Application à la segmentation automatique de la parole continue

Les procédures externes utilisées par les différents outils ont été écrites en langage C et utilisées sous le gestionnaire de fenêtre SunView sur SUN.
L'exécution du programme consiste d'abord à créer un environnement sous SunView. La figure V.10 représente ce que l'utilisateur peut faire, voir ou tester pendant l'exécution du programme. Nous pouvons distinguer quatre zones:

- une zone de commentaires des courbes située dans une fenêtre à gauche des courbes
- une zone pour tracer les courbes
- une zone pour suivre l'évolution des horloges, située en bas à gauche. Dans notre cas, ce sont les horloges permettant de suivre le déroulement des réinjections du signal
- une zone pour intervenir pendant l'exécution, située en bas à droite. Par exemple, utiliser le *PACE_MAKER*, exécuter le programme échantillon par échantillon (aux horloges du signal d'entrée ou du signal suréchantillonné), modifier l'ordre des modèles, interdire le retour-arrière, supprimer un (ou deux) des trois processus de segmentation, etc.

N.B: Les procédures externes sont facilement tranportables car il s'agit de fonctions classiques d'affichage.

Dans l'immédiat, nous allons étudier la possibilité d'intégrer de tels outils (*SONDE* et *DEBUG*) dans l'interface graphique. Par exemple en associant ces processus à des ports du programme. Ainsi le programmeur aura une vision non modifiée de son programme source, tout en ayant la possibilité de "descendre" dans un port pour lui associer un processus.

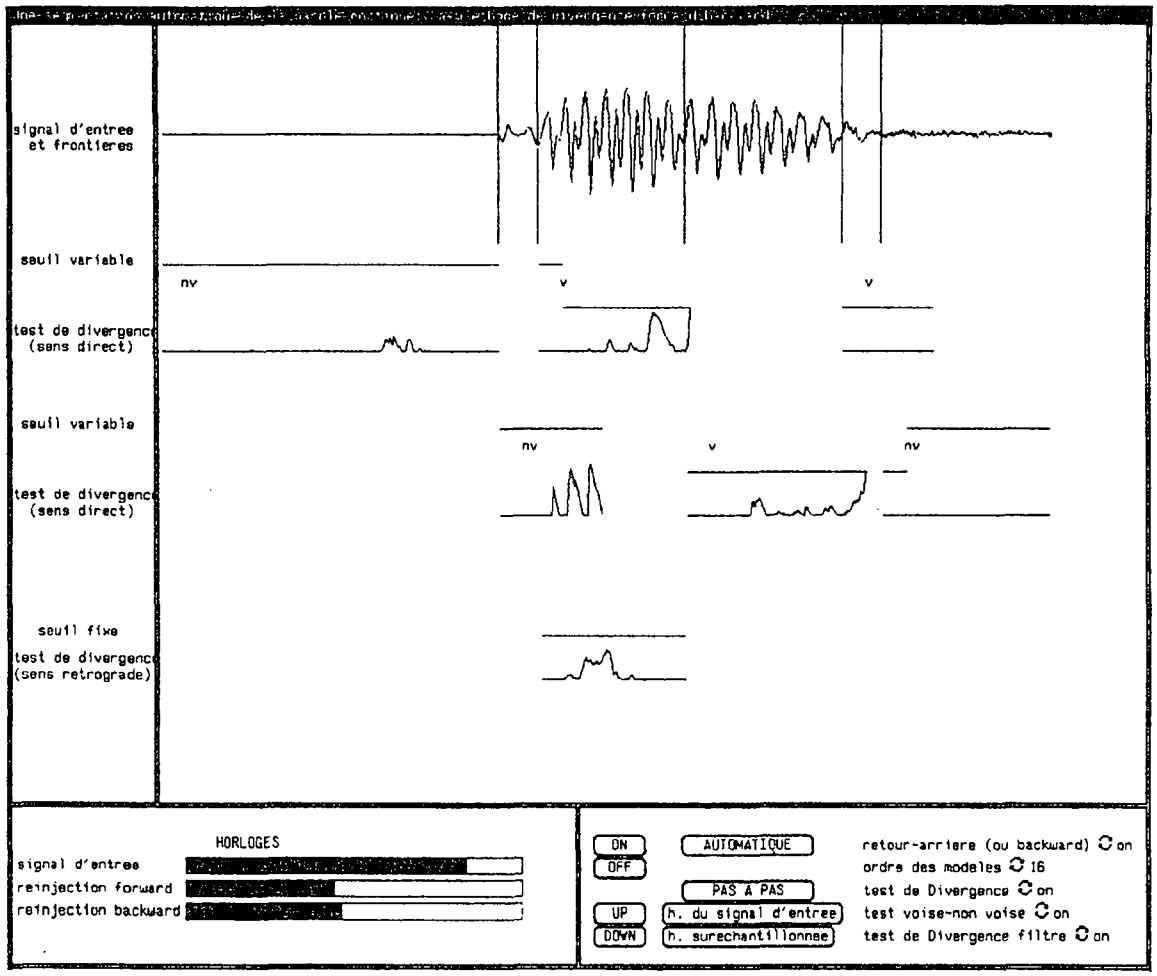

Figure V.10 : Environnement synchrone autour d'un module de segmentation.

Chapitre VI

Conclusion

Dans ce rapport, nous proposons d'appliquer les concepts de la programmation synchrone à la reconnaissance de la parole continue. Pour cela, nous utilisons SIGNAL, un langage synchrone à flots de données pour la programmation de systèmes à temps-réel.
Le module de segmentation qui est décrit ici est un module qui manipule le temps de façon complexe. Ce module est un module clef du décodeur acoustico-phonétique développé par l'équipe parole de l'IRISA. Lorsque l'on sait que le décodage acoustico-phonétique est lui-même un des problèmes clefs du domaine de la reconnaissance de la parole et que rien n'est proposé de véritablement satisfaisant en ce qui concerne la programmation et la mise en œuvre de ce genre d'application, on comprend aisément l'intérêt porté à une telle approche.

Le langage SIGNAL permet une description interne du parallélisme et une approche synchrone adaptée à la classe d'algorithmes étudiés. Il possède un outil graphique permettant d'avoir une vue modulaire et hiérarchique du programme. On peut ajouter une vérification complète du comportement temporel d'un programme (absence de blocage...) [12] [8] grâce à un mécanisme de calcul d'horloges qui transforme un programme en un système d'équations.
Actuellement, le compilateur génère du code FORTRAN simulant ainsi l'approche synchrone. Nous avons comparé les résultats de l'exécution de ce code avec le programme FORTRAN d'origine (équipe parole). Pour l'instant ils sont défavorables à SIGNAL, bien que le code FORTRAN produit ait été considérablement optimisé en cours d'étude; cette application ayant apporté sa contribution "au débogage" du compilateur actuel. La nouvelle version H2 du langage, avec un nouveau calcul d'horloges permettra sans doute (très prochainement) une meilleure optimisation du code produit. Cela devrait permettre d'égaler les performances du programme source, bien que SIGNAL ne soit pas prévu pour une exécution séquentielle.

Un translateur de programmes SIGNAL en OCCAM permettant une future mise en œuvre sur un réseau de transputers est en cours de réalisation. De plus des études sont en cours pour répartir le graphe obtenu à la compilation (caractéristique flots de données) sur une architecture multi-processeurs. Pour toutes ces raisons, SIGNAL se présente comme un langage spécifique au traitement de signal et bien adapté pour spécifier et mettre en œuvre des applications en parole.

Toutefois si cette étude a permis de mettre en évidence le côté hiérarchique, la transformation immédiate de systèmes d'équations et la puissance de SIGNAL en général (due en grande partie au fait qu'un processus SIGNAL peut recevoir des entrées complètement asynchrones), elle ne doit pas cacher les carences actuelles du langage dans ce genre d'applications, à savoir principalement le manque d'outils dynamiques mis à disposition du programmeur. Une prochaine étude plus poussée doit permettre de résoudre ce problème.
Nous avons déploré également le fait de ne pas pouvoir disposer de constantes locales ou de ne pas disposer de la compilation séparée. Mais ces deux points seront corrigés dans la prochaîne version.

En ce qui concerne les perspectives à court-terme, nous pouvons distinguer:

- le développement d'outils dynamiques pour le langage
- le développement en SIGNAL du décodeur acoustico-phonétique dans son ensemble: la partie statique du décodeur étant actuellement programmée et en cours de test
- les développements de l'interface graphique permettant d'automatiser les observations et les interventions de l'utilisateur pendant l'exécution
- le développement du poste de traitement de signal. A cette fin, l'équipe parole doit très prochainement commencer à utiliser ce poste et nos premiers résultats. Il ne fait aucun doute que cela nous permettra de l'améliorer considérablement en cernant les vrais besoins des concepteurs.

Remerciements: Je tiens à remercier Bruno Chéron pour sa compétence et sa disponibilité aussi bien en ce qui concerne les problèmes liés au compilateur SIGNAL que pour ceux plus portés vers la programmation proprement dite. Je lui associe toute l'équipe SIGNAL.
Enfin je tiens à remercier mon chef de projet Paul Le Guernic, Régine André-Obrecht et Albert Benveniste pour les nombreuses discussions et réunions qui m'ont permis d'avancer dans mon travail et sans lesquelles rien n'aurait été possible.

Bibliographie

[1] R. ANDRE-OBRECHT : *A New Approach for the Automatic Segmentation of Continuous Speech Signals* ; IEEE Trans. on ASSP, ASSP-36 No 1, pp. 29-40, January 1988.

[2] M. BASSEVILLE, A. BENVENISTE : *Sequential Detection of Abrupt Changes in Spectral Characteristics of Digital Signals* ; IEEE Trans. on Information Theory, Vol. 29, No 5, pp. 709-723, September 1983.

[3] C. BECHON : *Description temps réel des algorithmes du traitement du signal dans le langage SIGNAL: un exemple* ; Thèse, Université de Rennes 1, France, 1989.

[4] A. BENVENISTE : *Algorithmes simples d'estimation en treillis pour les séries longues* ; Outils et modèles mathématiques pour l'automatique, l'analyse des systèmes et le traitement du signal, Vol. 2, CNRS, Ed. 1982.

[5] A. BENVENISTE, B. LE GOFF, P. LE GUERNIC : *Hybrid Dynamical Systems theory and the language SIGNAL* ; Research report 838, INRIA France, Rocquencourt, April 1988.

[6] J.L. BERGERAND, P. CASPI, N. HALBWACHS, D. PILAUD, E. PILAUD : *Outline of a real-time Data-Flow Language* ; in Real-Time Systems Symposium, San Diego, December 1985.

[7] G. BERRY, L. COSSERAT : *The ESTEREL Programming Language and its Mathematical Semantics* ; Research report 327, INRIA FRANCE, 1984.

[8] L. BESNARD : *Mise en œuvre séquentielle de SIGNAL un langage orienté flot de données, synchrone pour applications temps-réel* ; Thèse, Université de Rennes 1, France, 1990, à paraître.

[9] P. BOURNAI, V. KERSCAVEN, P. LE GUERNIC : *Un environnement graphique pour la conception d'applications temps-réel* ; in Didier Plateau, éditeur, Ingéniérie des interfaces homme-machine, IN2-INRIA-LRI, Cargese, Corse, France, Mai 1989.

[10] CALLIOPE : *La parole et son traitement automatique* ; MASSON, Paris, Milan, Barcelone, Mexico, 1989.

[11] F. DECHELLE, Y. SOREL : *Utilisation du langage SIGNAL pour la spécification et la mise en œuvre d'algorithmes de traitement du signal* ; GRETSI, NICE, pp. 657-660, Juin 1987.

[12] T. GAUTIER, P. LE GUERNIC, L. BESNARD : *SIGNAL: a declarative language for synchronous programming of real-time systems* ; in Gilles Kahn,editor, Functional programming languages and computer architecture, Lecture Notes in Computer Science, Portland, Oregon, USA, Volume 274, pp.257-277, September 1987.

[13] D. HAREL, A. PNUELI : *On the Development of Reactive Systems* ; In K.R.Apt editor, Logics and Models of Concurrent Systems, Springer-Verlag, New York, Volume 13, pp.477-498, 1985.

[14] D. HAREL : *Statecharts: A visual Approach to Complex Systems* ; Science of Computer Programming, Volume 8-3, pp.231-275, 1987.

[15] J.P. HATON : *Intelligence artificielle et compréhension automatique de la parole: Etat des recherches et comparaison avec la vision par ordinateur* ; TSI, Volume 4, No 5, pp.265-287, 1985.

[16] C. LE MAIRE, R. ANDRE-OBRECHT, P. LE GUERNIC : *Développement d'un poste de traitement de signal basé sur la programmation synchrone: application au traitement de la parole* ; XVIIIèmes Journées d'Etudes sur la Parole, Montréal, mai 1990.

[17] P. LE GUERNIC, A. BENVENISTE, P. BOURNAI, T. GAUTIER : *SIGNAL—A Data Flow-Oriented Language for Signal Processing* ; IEEE Trans. on ASSP, ASSP-34 No 2, pp. 362-374, April 1986.

[18] P. LE GUERNIC, A. BENVENISTE : *Real-time, Synchronous, Data Flow Programming: the Language SIGNAL and its Mathematical Semantics* ; INRIA, Rennes, France, Research report no 620, February 1987.

[19] J.O. MARKEL, A.H. GRAY : *Linear Prediction of Speech* ; Springer-Verlag, New York, 1976.

[20] Y. SOREL, P. WOLF : *Outils de mise au point de machines pour le traitement du signal. Etude d'un cas: réalisation d'un modem 4800 bps* ; GRETSI, NICE, pp. 1029-1034, Mai 1985.

Annexe A

Bibliothèque de modules standards

A.1 processus *NB_PASSAGES_ZERO*

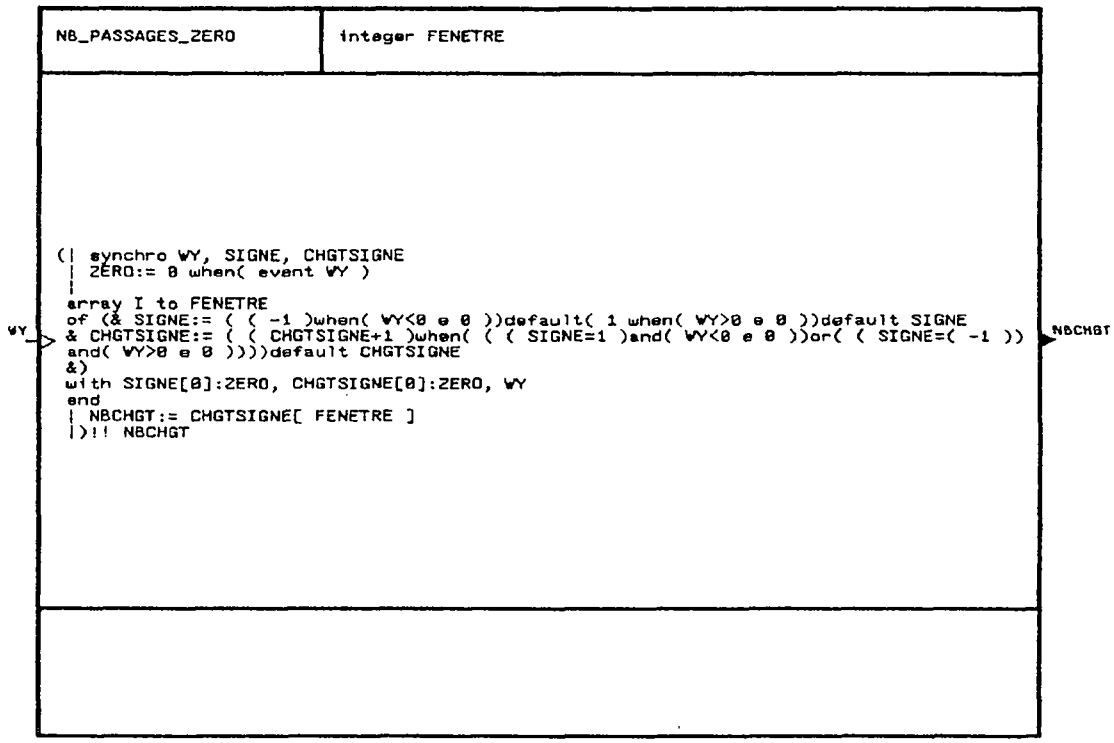

Figure A.1 : Le processus *NB_PASSAGES_ZERO*

Ce processus compte le nombre de passages par zéro (**NBCHGT**) sur une fenêtre de signal de parole (**WY**) de longueur **FENETRE**.
Tous les signaux sont synchrones à **WY**. Ce processus effectue un simple parcours de I=1 à **FENETRE**. **SIGNE** vaut +1 (resp. -1) si l'échantillon est positif (resp. négatif) et vaut le signe de l'échantillon précédent s'il est nul. **CHGTSIGNE** est incrémenté à chaque passage négatif-positif ou positif-négatif. **NBCHGT** récupère la valeur finale de **CHGTSIGNE**.

A.2 processus *AUTOCORRELATION1*

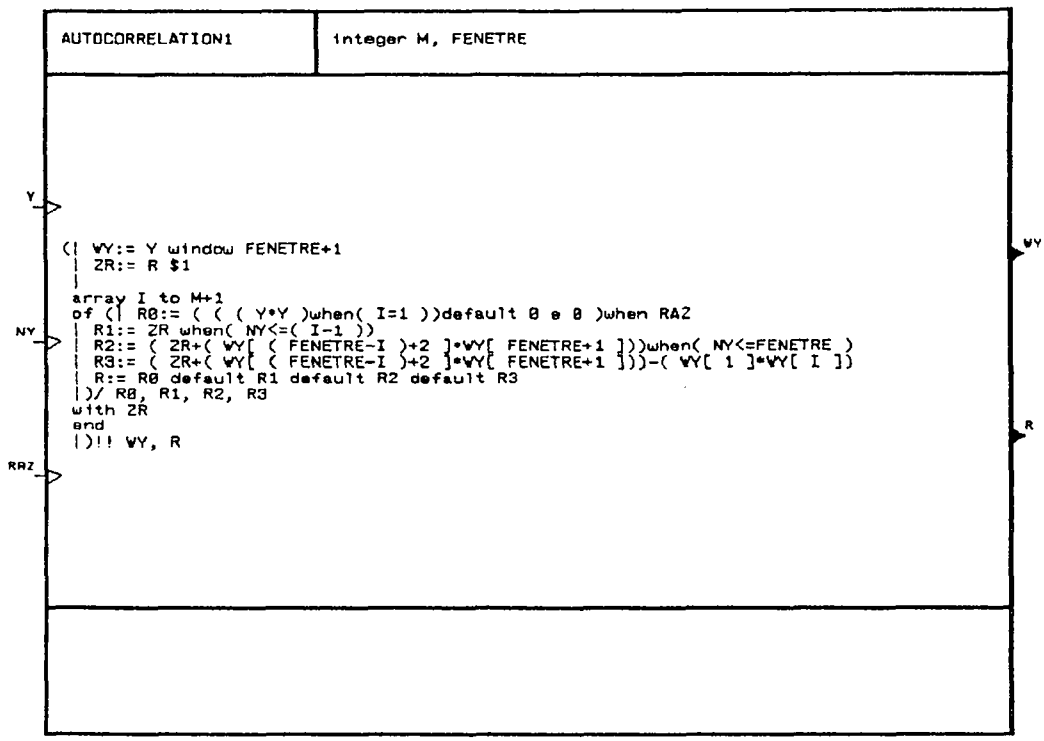

Figure A.2 : Le processus *AUTOCORRELATION1*

Ce processus calcule le vecteur des covariances (ordres 0 à M) à partir d'un signal de parole **Y**. Le vecteur **R** est synchrone à **Y**, mais les valeurs qu'il fournit ne seront significatives à l'extérieur qu'après une période d'initialisation **FENETRE**.
Le processus stocke **Y** dans une fenêtre glissante de longueur **FENETRE**. Cette fenêtre est fournie en sortie du processus. Il effectue un parcours pour calculer les (M+1) éléments du vecteur. Les cas à différencier sont, dans l'ordre prioritaire:

- lors d'une réinitialisation (**RAZ**=vrai). Dans ce cas, seule l'énergie (ordre 0) est significative, les autres valeurs étant nulles (signal **R0**)
- lorsque le nombre d'échantillons **NY** n'est pas suffisamment important pour calculer tous les ordres (signal **R1**)
- lorsque la fenêtre glissante n'est pas remplie (signal **R2**)
- lorsque la période d'initialisation est terminée (cas général, signal **R3**).

A.3 processus *AUTOCORRELATION2*

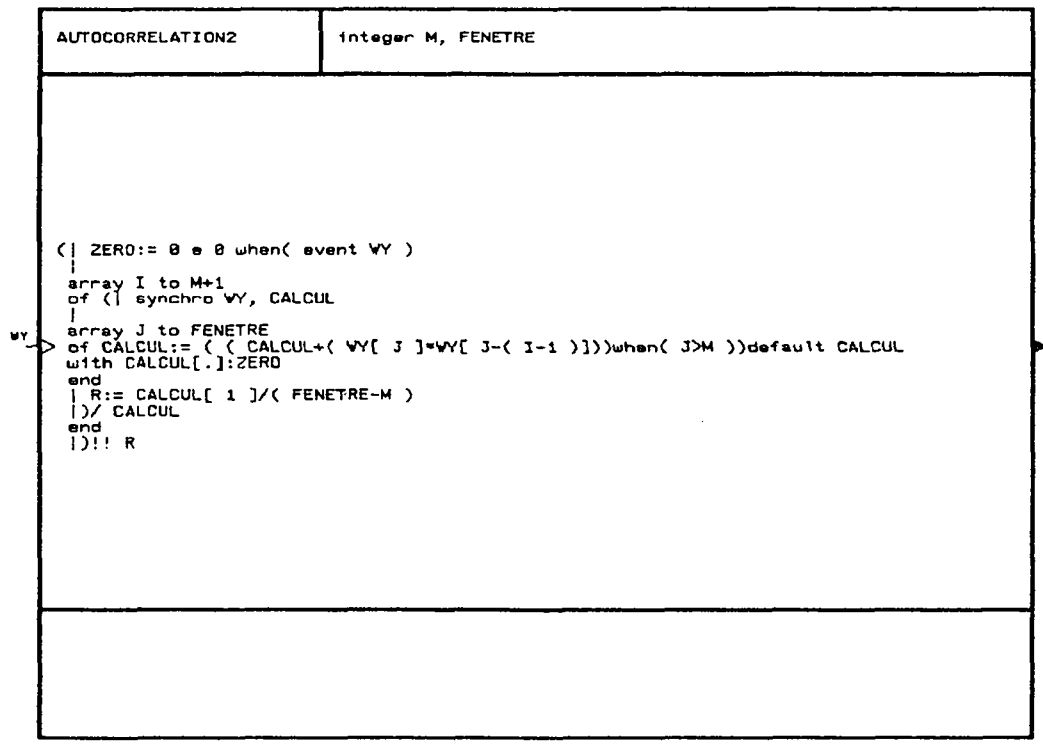

Figure A.3 : Le processus *AUTOCORRELATION2*

Ce processus calcule également le vecteur des covariances (ordres 0 à M), mais directement sur la fenêtre glissante **WY** fournie en entrée. Tous les signaux sont synchrones à **WY**.
Le processus effectue deux boucles imbriquées. Pour chaque élément du vecteur, l'ensemble de la fenêtre est parcourue. Pour avoir des résultats corrects, il faut donc contrôler à l'extérieur la période d'initialisation.

Dans notre cas, les M+1 coéfficients d'autocorrélation sont tous évalués sur une même fenêtre de longueur (**FENETRE**-M) par la formule suivante:

$$R[i] = \frac{\sum_{j=M+1}^{FENETRE}(WY[j] * WY[j-i+1])}{FENETRE - M}.$$

A.4 processus *AUTO*

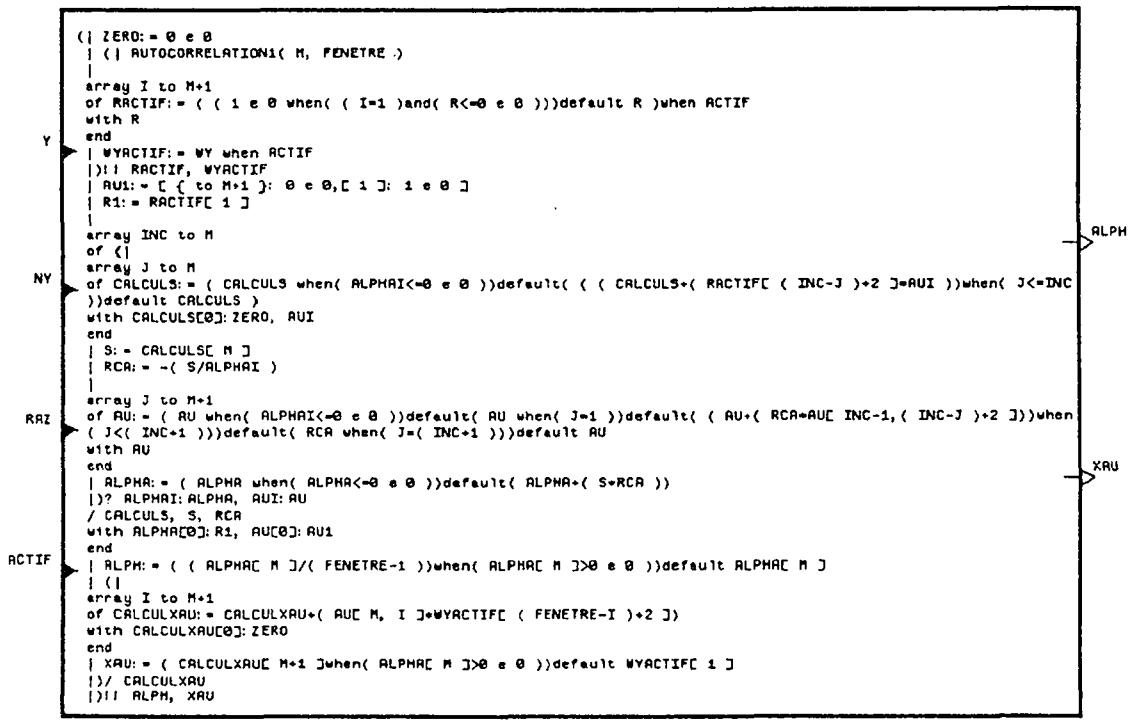

Figure A.4 : Le processus *AUTO*

Ce processus correspond à la programmation de la méthode d'autocorrélation, qui permet d'identifier le modèle court-terme autorégressif. Le signal est supposé autorégressif gaussien: $y_t = \sum_{i=1}^{p} a_i y_{t-i} + e_t$, *p ordre du modele.*
La méthode a pour but d'estimer les coefficients $(a_i)_{i=1,p}$ à partir de l'autocorrélation du signal, calculée sur une fenêtre de longueur fixe; les formules sont celles de Levinson [19] .
A chaque **RAZ**, une phase d'initialisation est lancée sur une fenêtre de longueur **FENETRE**. En sortie, **ALPH** et **XAU** représentent l'énergie résiduelle et l'erreur calculées sur cette même fenêtre. Les tableaux réguliers de processus sont utilisés pour programmer les structures de répétition.
Ce processus utilise *AUTOCORRELATION1* (annexe A.2). Le vecteur des covariances **R** est sous-échantillonné à l'horloge de **ACTIF**=vrai, les calculs étant effectués de façon synchrone à **RACTIF**.
N.B. les signaux d'entrée **Y**, **NY**, **ACTIF** sont synchrones; les signaux de sortie **ALPH**, **XAU**, (**ACTIF**=vrai) sont synchrones.

A.5 processus *BURG*

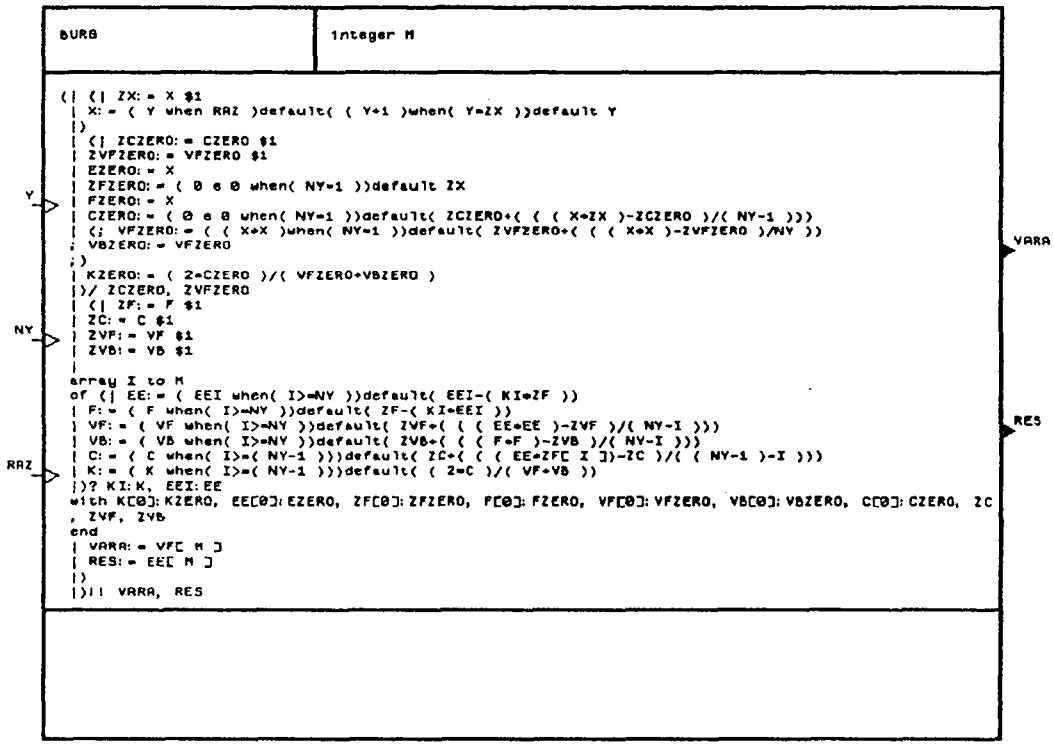

Figure A.5 : Le processus *BURG*

Ce processus correspond à la programmation de la méthode de Burg [4] , qui permet d'identifier le modèle long-terme autorégressif.
Le signal est supposé également autorégressif gaussien (annexe A.4). L'estimation des coefficients se fait de manière récursive en fonction du temps et de l'ordre des modèles (formules en treillis); la fenêtre d'analyse est croissante.
En sortie, **VARA** et **RES** représentent respectivement l'énergie résiduelle et l'erreur, qui servent pour le calcul de la distance.

Les formules en treillis se prêtent parfaitement à la programmation SIGNAL. Pour le démontrer plaçons nous dans un cadre plus général.
Soit $A_N(z) = 1 + \sum_1^N a_i z^i$, un filtre polynomial.
Si nous posons $e_t(n) = A_n(z)y_t$ et $f_t(n) = A_n^*(z)y_t$,
alors les formules récursives définissant les suites du filtre polynomial sont:
$e_t(n) = e_t(n-1) - k_n * f_{t-1}(n-1)$ et
$f_t(n) = -k_n * e_t(n-1) + f_{t-1}(n-1)$
Ces équations peuvent être représentées par un structure en treillis (figure A.6).

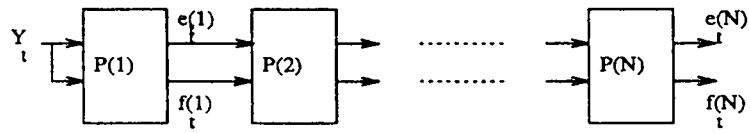

Figure A.6 : Une structure en treillis.

La traduction en SIGNAL est aisée avec l'aide des tableaux réguliers de processus:

$$(|\ zf := f\$1 \\ |\ array\ n\ to\ N \\ \quad of\ P(n) \\ \quad end \\ |) \qquad avec\ P(n) = \qquad (|\ e := e[n-1] - k * zf[n-1] \\ |\ f := -k * e[n-1] + zf[n-1] \\ |\ k := ... \\ |)$$

Dans le cas de Burg, k est égal à $2 * c/(vf + vb)$, qui dépendent de e et f (voir le texte SIGNAL), la programmation étant immédiate.

N.B. les signaux suffixés par **ZERO** concernent les initialisations.
N.B. le signal e est appelé **EE**.
N.B. les signaux **Y**, **NY**, **VARA**, **RES** sont synchrones.

A.6 processus *MODELE*

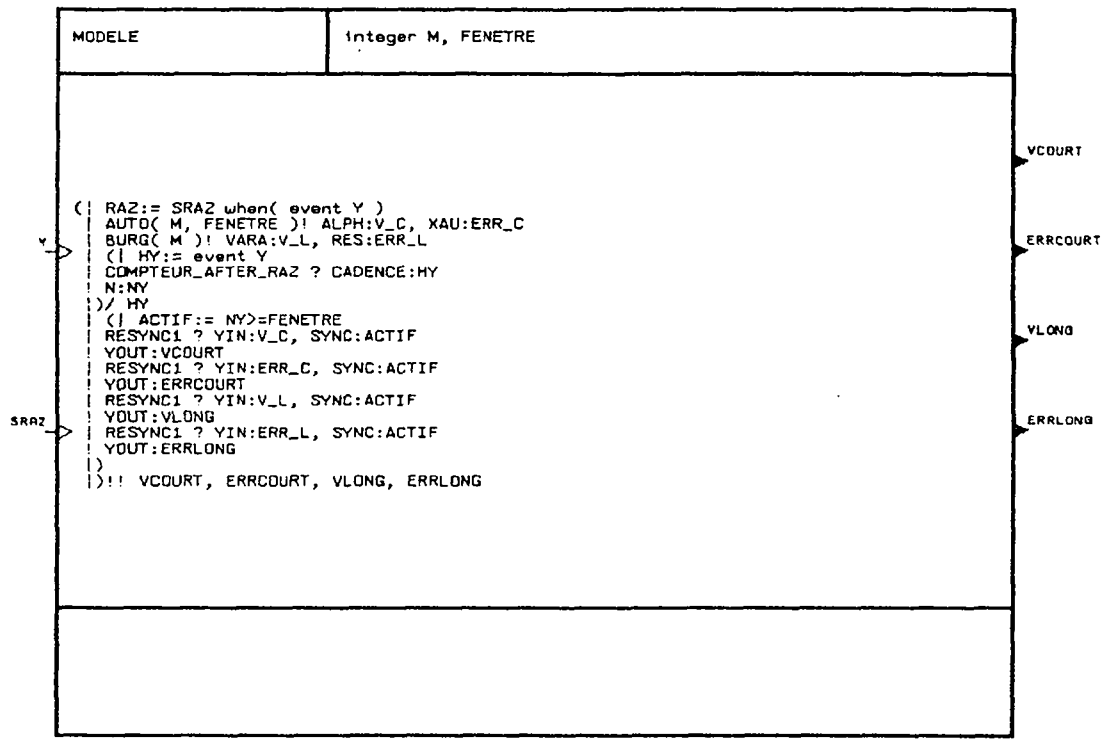

Figure A.7 : Le processus *MODELE*

Ce processus effectue les appels aux processus *AUTO* et *BURG*. **M** est l'ordre des modèles, **Y** le signal de parole et **SRAZ** indique une réinitialisation.
Les signaux de sortie du modèle court, l'énergie résiduelle et l'erreur, sont renommés **V_C** et **ERR_C**. Les signaux de sortie du modèle long sont renommés **V_L** et **ERR_L**. Puis ces quatre signaux sont resynchronisés à l'horloge de **ACTIF**=vrai, correspondant à la fin de la période d'initialisation.

A.7 processus *STAT_HINKLEY*

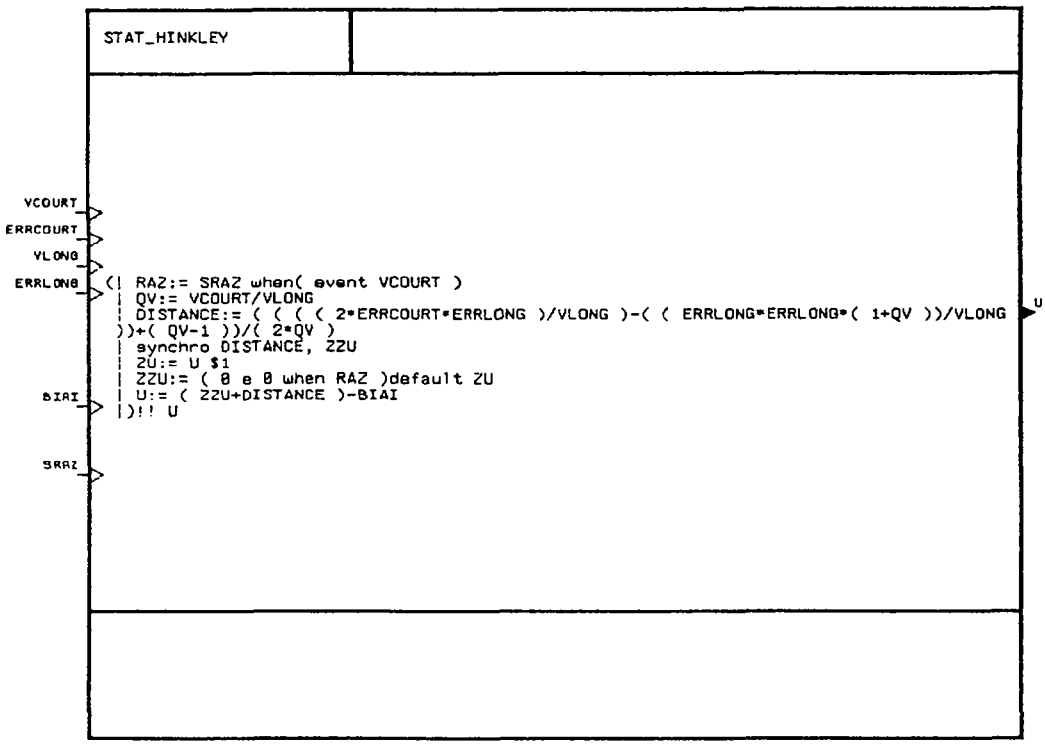

Figure A.8 : Le processus *STAT_HINKLEY*

Ce processus calcule la statistique de Hinkley à partir des signaux issus des modèles autoregressifs. La distance entre les deux modèles est tout d'abord calculée d'après la formule suivante:

$$DISTANCE = \frac{\frac{2*ERRCOURT*ERRLONG}{VLONG} - \frac{ERRLONG^2*(1+QV)}{VLONG} + QV - 1}{2 * QV},$$

$$avec\ QV = \frac{VCOURT}{VLONG}.$$

Puis la statistique **U** se calcule à partir de cette distance, du biais fourni en entrée et de l'ancienne valeur de **U** (**ZZU**):

$$U := ZZU + DISTANCE - BIAI.$$

N.B: Les signaux **VCOURT**, **ERRCOURT**, **VLONG**, **ERRLONG**, **DISTANCE**, **BIAI** et **U** sont synchrones. Le test est réinitialisé lorsque **SRAZ** est présent et égal à vrai.

A.8 processus *MAX*

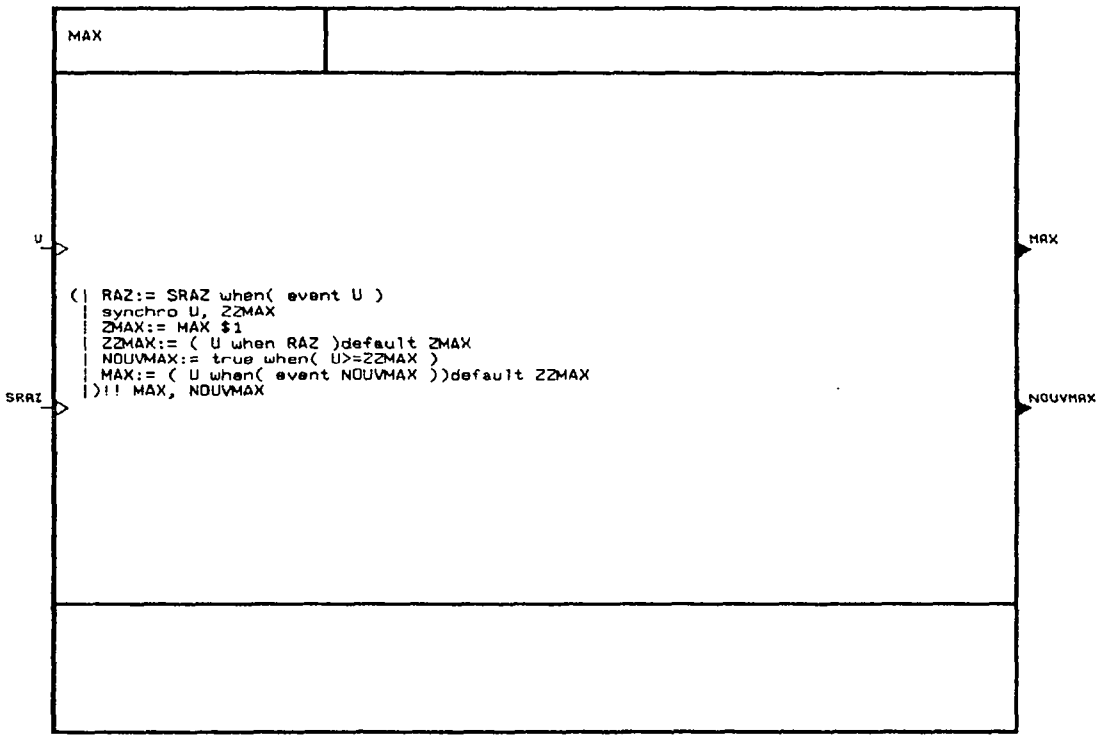

Figure A.9 : Le processus *MAX*

Ce processus calcule le maximum des valeurs d'un signal d'entrée **U**. Le maximum **MAX** est synchrone à **U**, et ne change pas de valeur lorsqu'il n'y a pas de nouveau maximum. **NOUVMAX** est également fourni en sortie; il prend la valeur vrai lors d'un nouveau maximum.

A.9 processus *CALCUL_VOISEMENT*

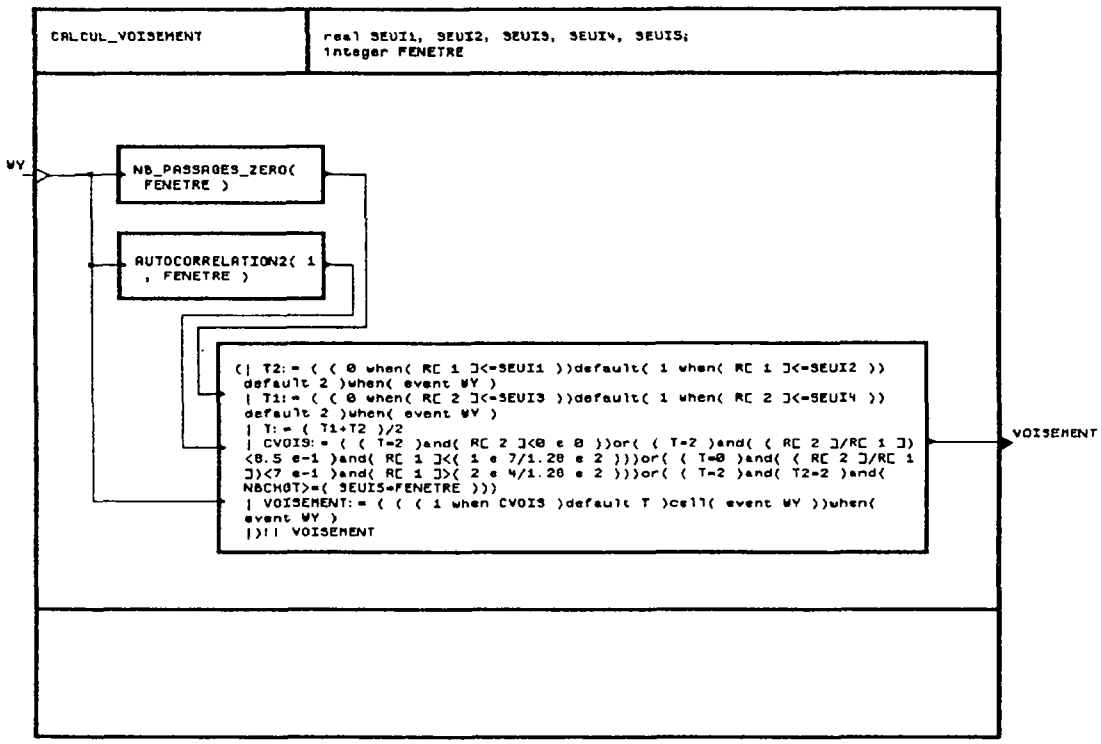

Figure A.10 : Le processus *CALCUL_VOISEMENT*

Ce processus associe une marque de voisement **VOISEMENT** à la fenêtre **WY** contenant le signal de parole. Le processus *AUTOCORRELATION2* calcule le vecteur des covariances **R** (ordres 0 et 1).
Une première marque de voisement est proposée après avoir comparé l'énergie et le coefficient d'autocorrélation d'ordre 1 avec des seuils donnés en paramètres du processus.
La marque de voisement peut ensuite être éventuellement corrigée après comparaison avec d'autres seuils ou avec le nombre de passages par zéro **NBCHGT** issu de *NB_PASSAGES_ZERO*.

A.10 processus *CONVOLUTION_LINEAIRE*

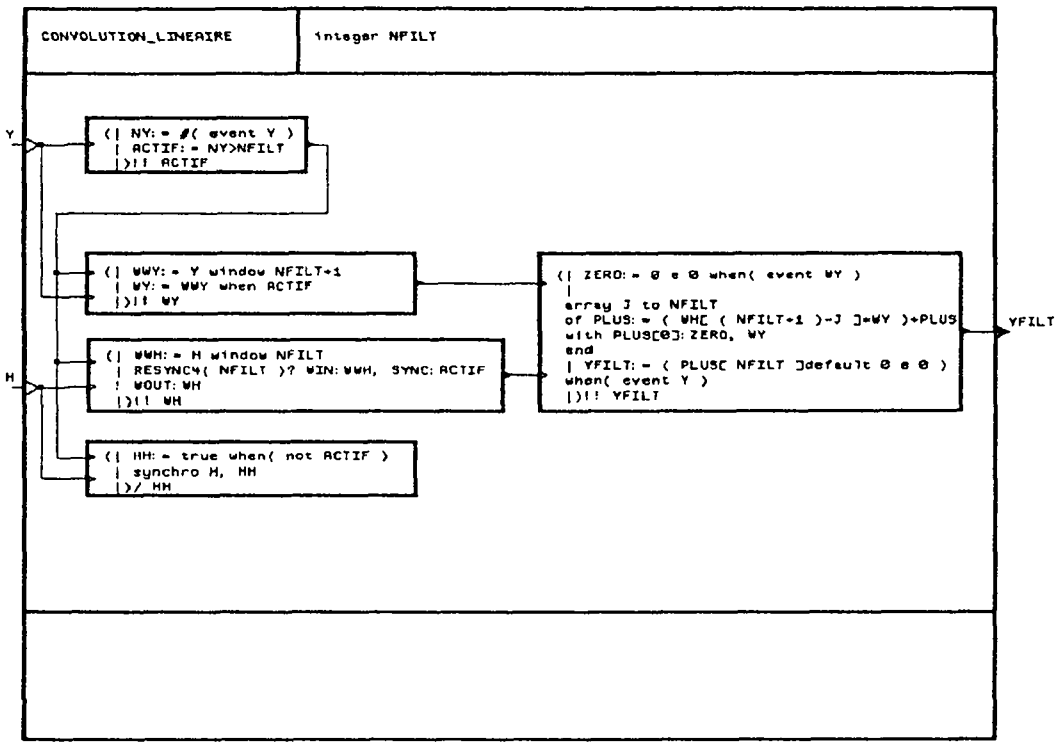

Figure A.11 : Le processus *CONVOLUTION_LINEAIRE*

Ce processus filtre le signal de parole **Y**. En sortie, le signal **YFILT** est synchrone à **Y** et vaut zéro pendant la phase d'initialisation de longueur **NFILT**. Les signaux **Y** et **H** (réponse impulsionnelle) sont stockés dans des fenêtres glissantes, elles-mêmes resynchronisées à l'horloge de **ACTIF**=vrai (signaux **WWY** et **WWH**).
Chaque échantillon filtré dépend des Nfilt échantillons précédents, de la façon suivante: $YFILT_t = \sum_{j=1}^{NFILT} H_j * Y_{t-j}$.

Le signal **H** n'est présent que pendant les **NFILT** premiers échantillons, c'est à dire pendant la phase d'initialisation, car il ne varie pas par la suite. Pour disposer de **H**, il suffit de synchroniser son horloge avec l'horloge de la phase d'initialisation. Soit en SIGNAL:

```
(| hh:= true when (not actif)
 | synchro h,hh
 |)/ hh
```

Annexe B

Bibliothèque de mécanismes SIGNAL

B.1 processus *RESYNC*

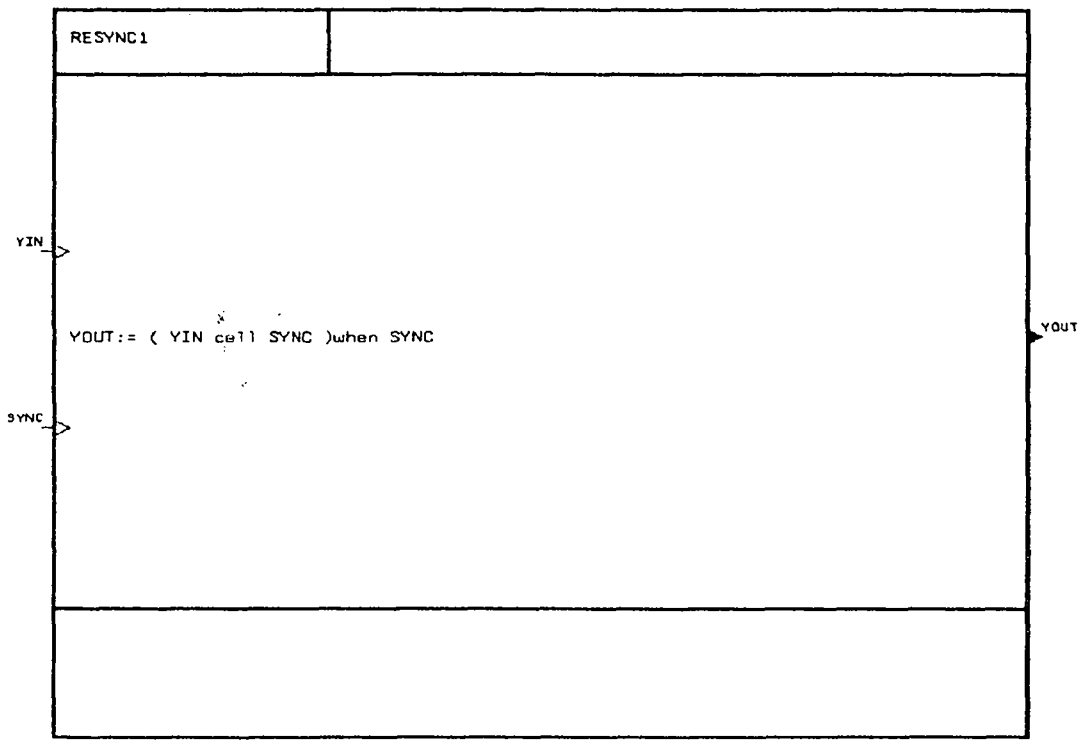

Figure B.1 : Le processus *RESYNC*

Ce processus resynchronise le signal **YIN** en un signal **YOUT** à l'horloge de **SYNC**=vrai (mémorisation puis souséchantillonnage).

N.B: Il existe "en cinq exemplaires", un pour chaque type des signaux **YIN** et **YOUT** (réels, entiers, booléens, tableaux d'entiers et tableaux de réels). Dans le cas de tableaux les signaux d'entrée et de sortie sont appelés **WIN** et **WOUT**.

B.2 processus *RESYNCEVENT*

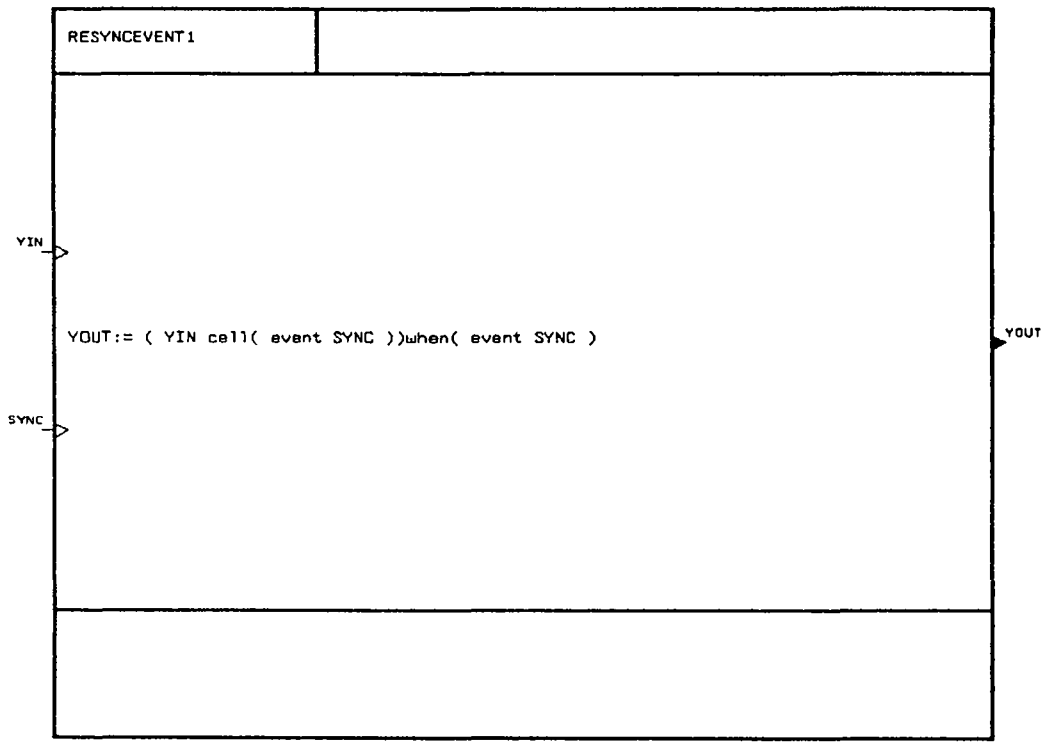

Figure B.2 : Le processus *RESYNCEVENT*

Ce processus resynchronise le signal **YIN** en un signal **YOUT** à l'horloge de **SYNC**.

N.B: De la même façon que *RESYNC*, il existe pour 5 types différents.

B.3 processus *COMPTEUR_AFTER_RAZ*

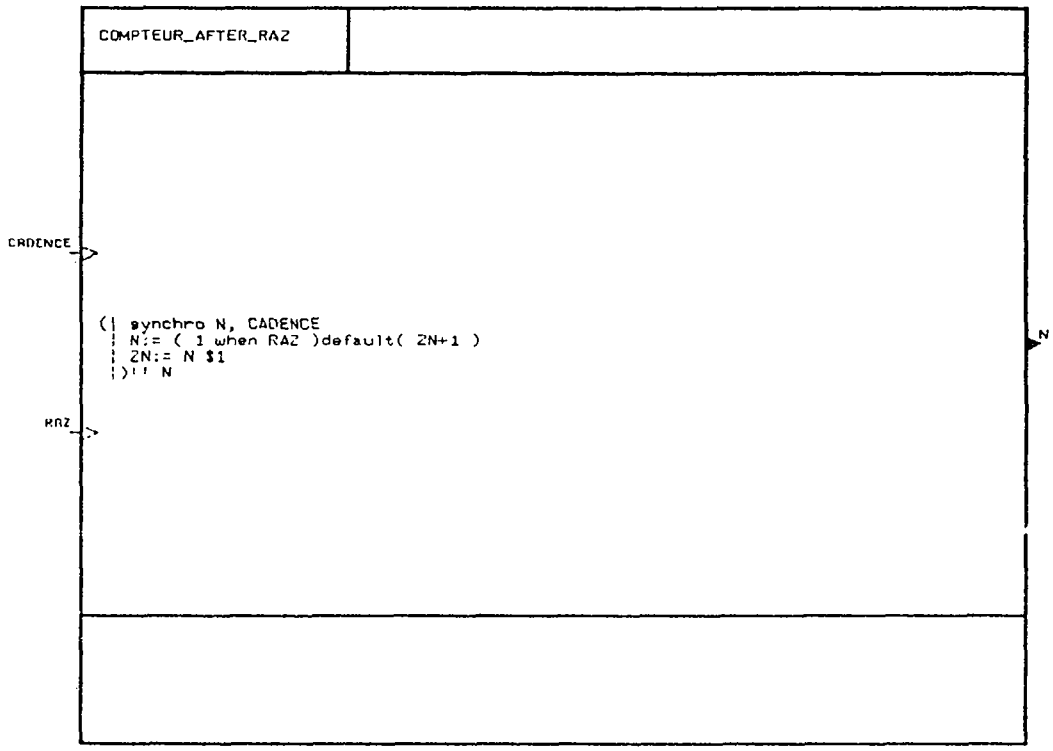

Figure B.3 : Le processus *COMPTEUR_AFTER_RAZ*

Ce processus est un compteur d'événements **CADENCE**. Le compteur **N** peut être réinitialisé de l'extérieur lorsque **RAZ** est présent et égal à vrai.

N.B: **CADENCE** et **N** sont synchrones. L'horloge de **RAZ** est celle de **CADENCE** (c.a.d. moins fréquente).

B.4 processus *COMPTEUR_MODULO*

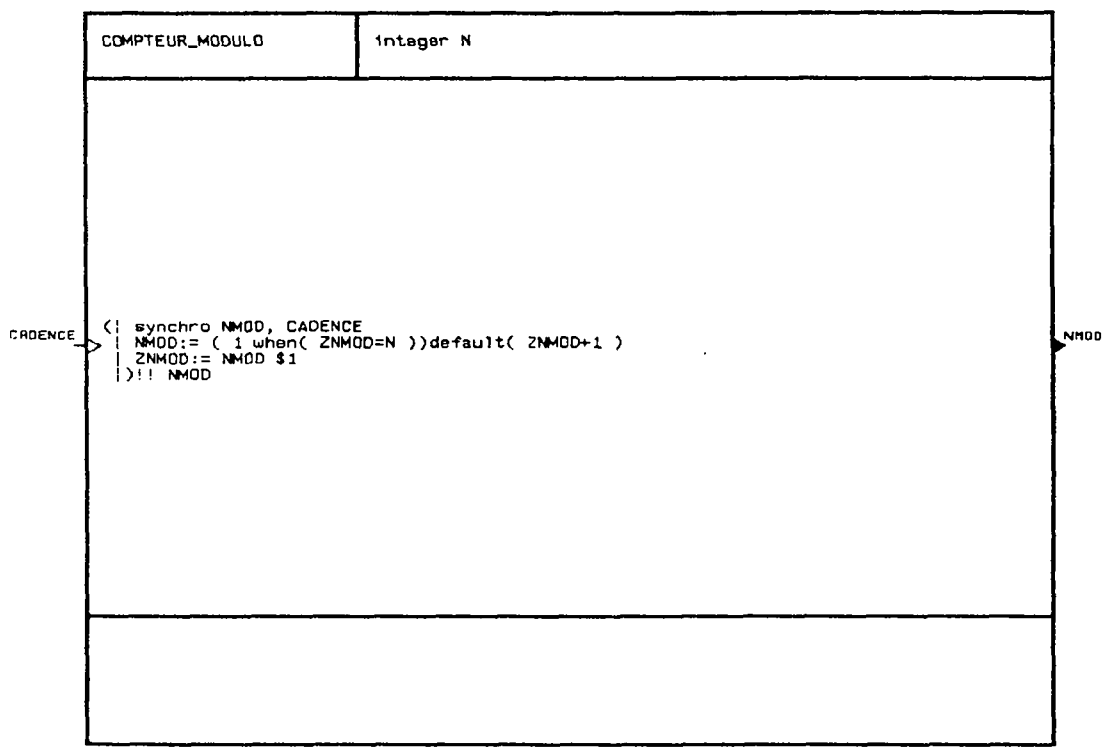

Figure B.4 : Le processus *COMPTEUR_MODULO*

Ce processus est un compteur d'événements **CADENCE** modulo le paramètre N. Le compteur est donc réinitialisé lorsque la dernière valeur du compteur (**ZNMOD**) est égale au modulo.

N.B: Le signal **NMOD** est synchrone à **CADENCE**.
N.B: Deux variantes de ce compteur sont données ci-après. Ces trois processus pourraient être regroupés en un, mais ils sont séparés ici pour se familiariser avec le langage SIGNAL.

B.5 processus *ACCU_MOD*

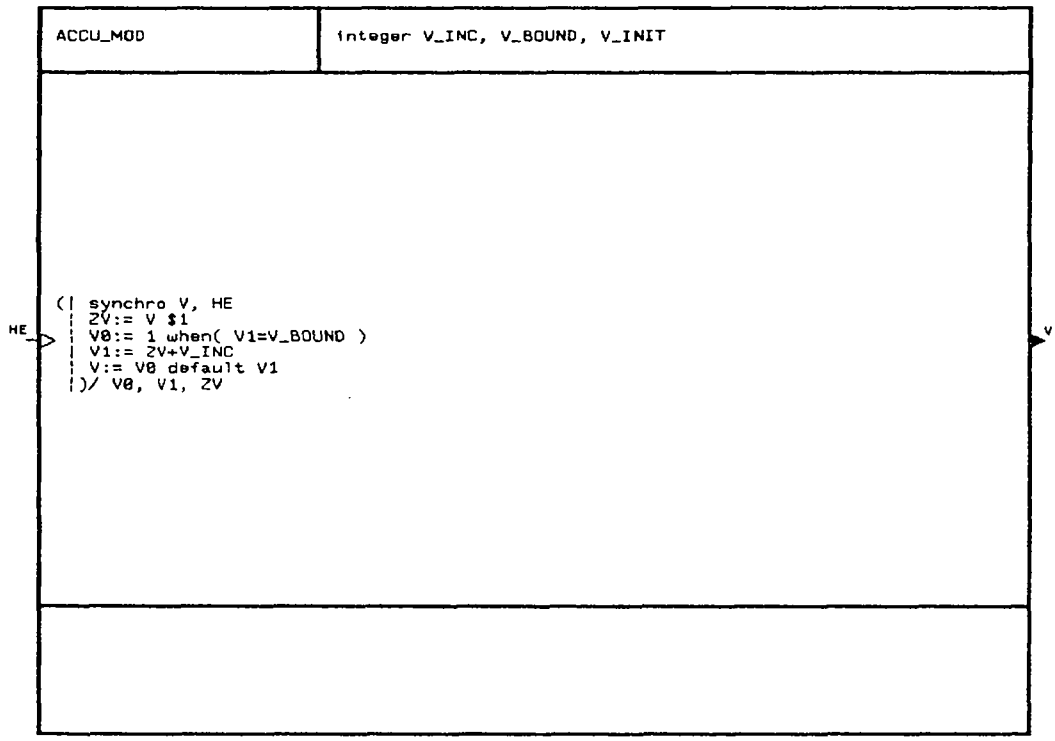

Figure B.5 : Le processus *ACCU_MOD*

Ce processus est un compteur modulo. La différence avec l'autre compteur modulo est que les valeurs d'initialisation lors du démarrage (**V_INIT**) et d'incrémentation (**V_INC**) sont passées en paramètre.
La valeur **V** du compteur est réinitialisée à un lorsque le modulo est atteint.

N.B: **V** est synchrone au signal d'entrée **HE**.

B.6 processus *ACCU_BOUND*

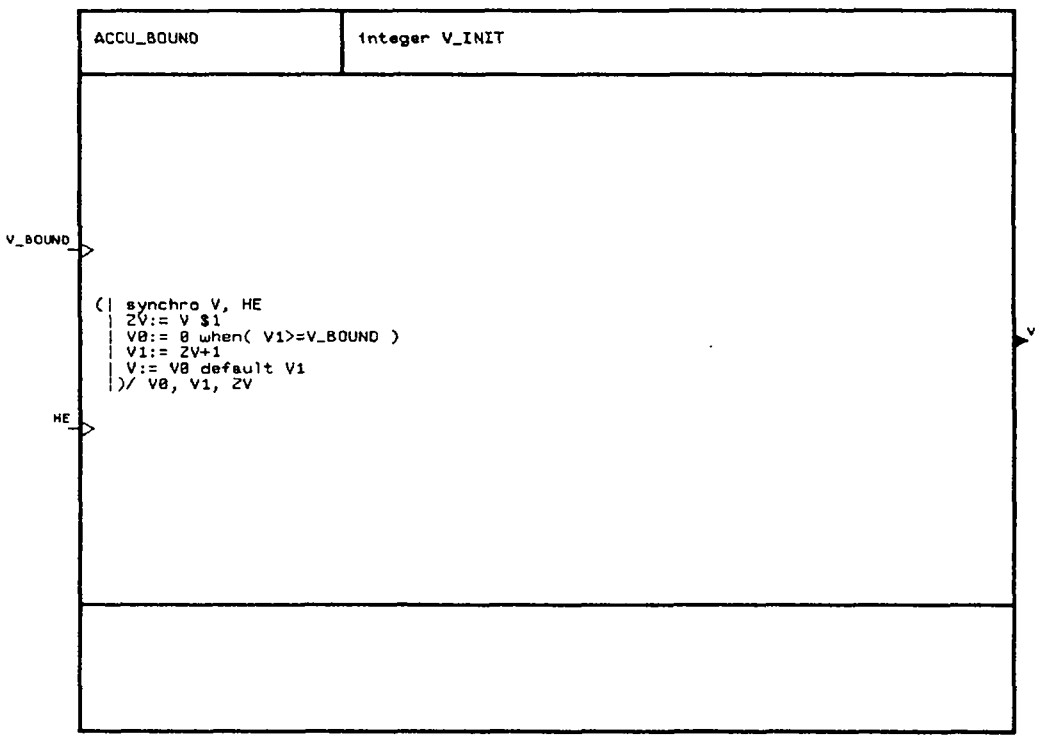

Figure B.6 : Le processus *ACCU_BOUND*

Ce processus est un compteur modulo. La différence avec l'autre compteur modulo est que la valeur du modulo **V_BOUND** est un signal et peut donc varier. La valeur **V** du compteur est réinitialisée à zéro lorsque le modulo est atteint.

N.B: **V** est synchrone au signal d'entrée **HE**.

B.7 processus *COMPTEUR_MODULO_AFTER_RAZ*

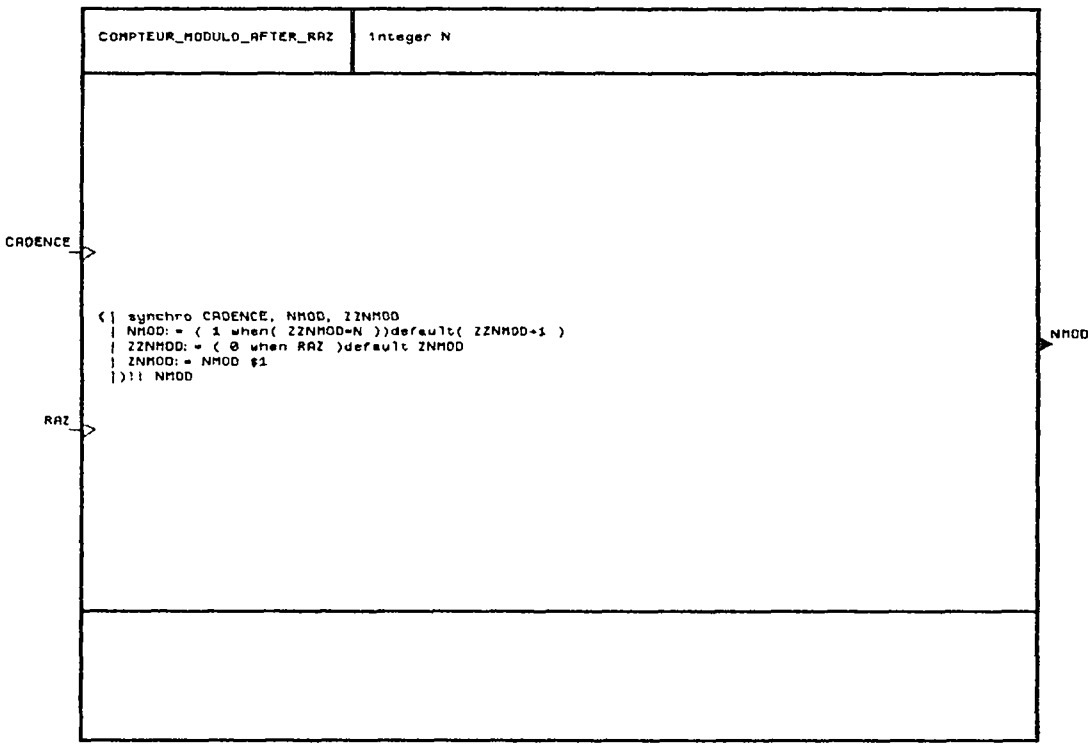

Figure B.7 : Le processus *COMPTEUR_MODULO_AFTER_RAZ*

Ce processus est un compteur d'événements **CADENCE** modulo le paramètre **N**. Le compteur est donc réinitialisé lorsque la dernière valeur du compteur (**ZZNMOD**) est égale au modulo.
Il peut également être réinitialisé de l'extérieur lorsque **RAZ** est présent et égal à vrai.

N.B: Le signal **NMOD** est synchrone à **CADENCE**. L'horloge de **RAZ** est inférieure à celle de **CADENCE**.

B.8 processus *COMPTEUR_DIFF*

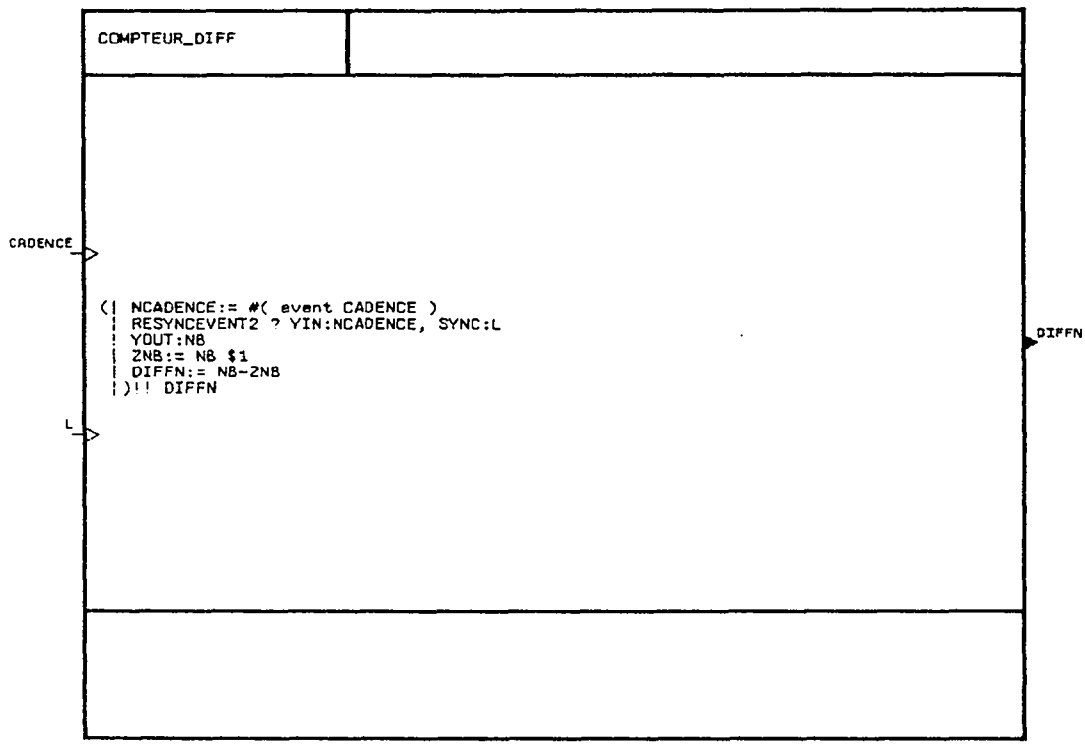

Figure B.8 : Le processus *COMPTEUR_DIFF*

Ce processus compte le nombre d'événements **CADENCE**. Le compteur **NCADENCE** est resynchronisé à l'horloge du signal **L**. En sortie, **DIFFN** exprime le nombre d'événements **CADENCE** arrivés depuis le dernier événement **L**.

N.B: Les signaux **L** et **DIFFN** sont synchrones.

B.9 processus *FLIPFLOP*

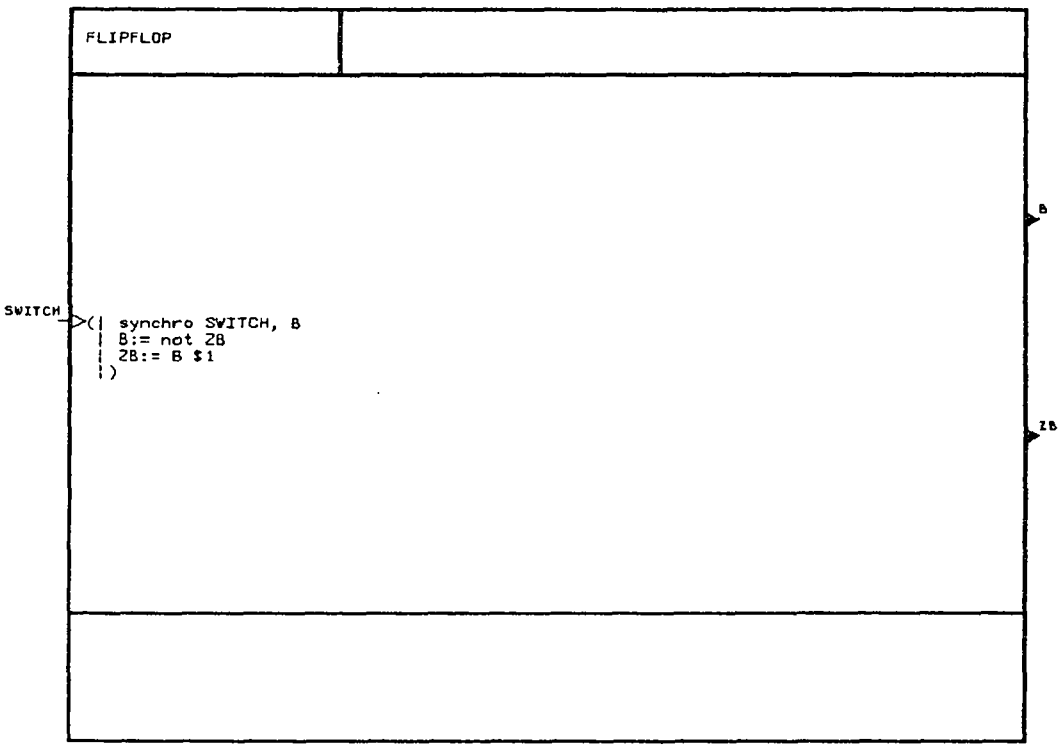

Figure B.9 : Le processus *FLIPFLOP*

Ce processus représente une bascule. Le signal booléen **B** est inversé à chaque réception du signal **SWITCH**.

N.B: Les signaux bf SWITCH, **B** et **ZB** sont synchrones.

B.10 processus *FBY*

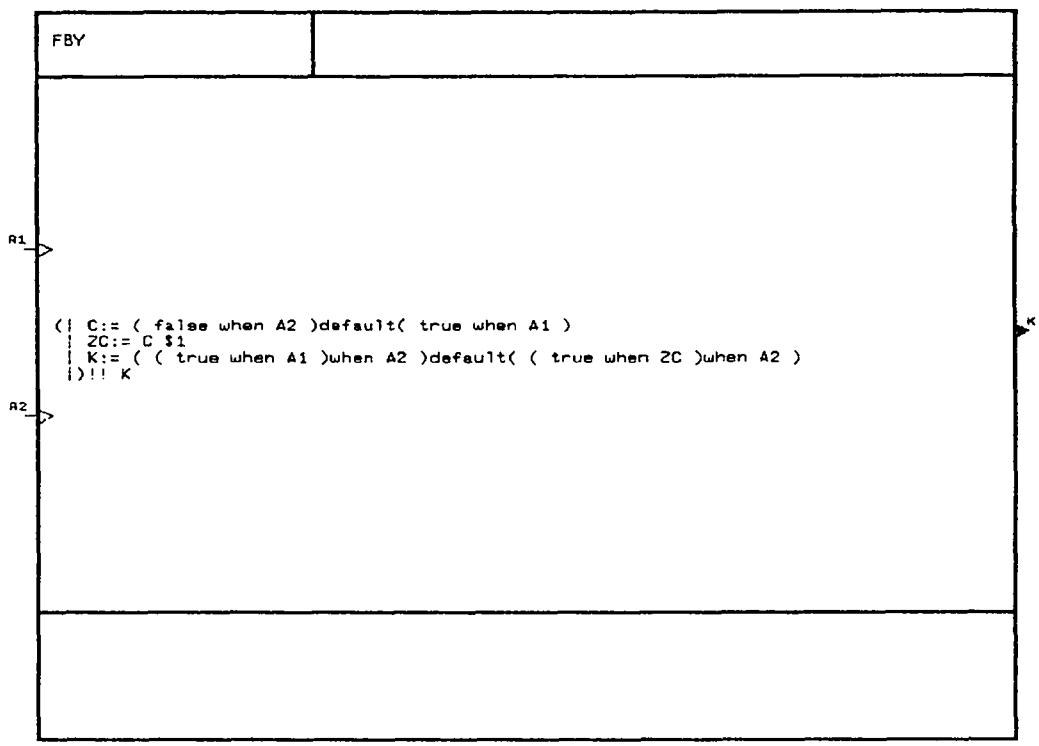

Figure B.10 : Le processus *FBY*

Ce processus est inspiré de celui décrit dans [5] ; le texte est ici réécrit différemment. Ils sont équivalents lorsque le signal d'entrée **A1** est un événement. Les différentes utilisations possibles de ce processus y sont décrites.
Ce processus retarde l'événement **A1** en un événement **K** jusqu'au prochain événement **A2**. **K** est émis si les deux signaux sont présents simultanément (figure B.11).

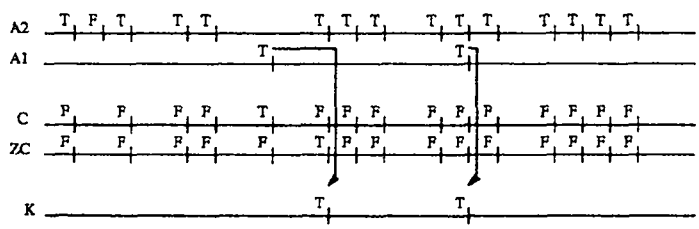

Figure B.11 : Exemple d'entrelacements des signaux du processus *FBY*

B.11 processus *INTERRUPT*

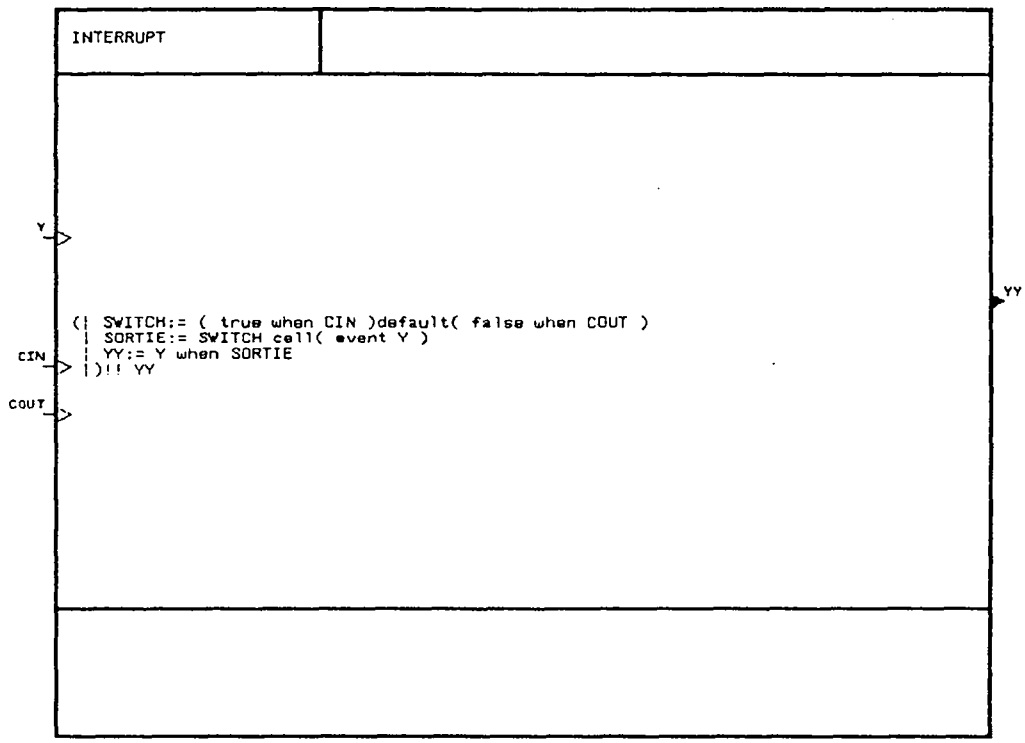

Figure B.12 : Le processus *INTERRUPT*

Ce processus sous-échantillonne le signal d'entrée **Y** en un signal **YY**. Le signal **YY** est interrompu lorsque **COUT** est présent et égal à vrai. Il est de nouveau émis lorsque **CIN** est présent et égal à vrai (figure B.13).
Lorsque **CIN** et **COUT** sont simultanément présents et égaux à vrai, il n'est pas tenu compte du signal **COUT**.
N.B: Les signaux **CIN**, **COUT** et **YY** ont des horloges inférieures à celle de **Y**.

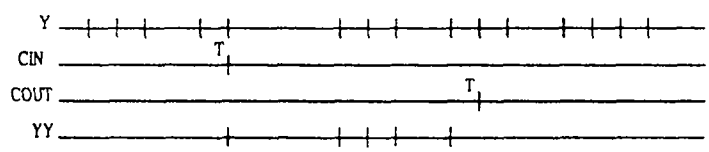

Figure B.13 : Exemple d'entrelacements des signaux du processus *INTERRUPT*

B.12 processus *DIFFERER*

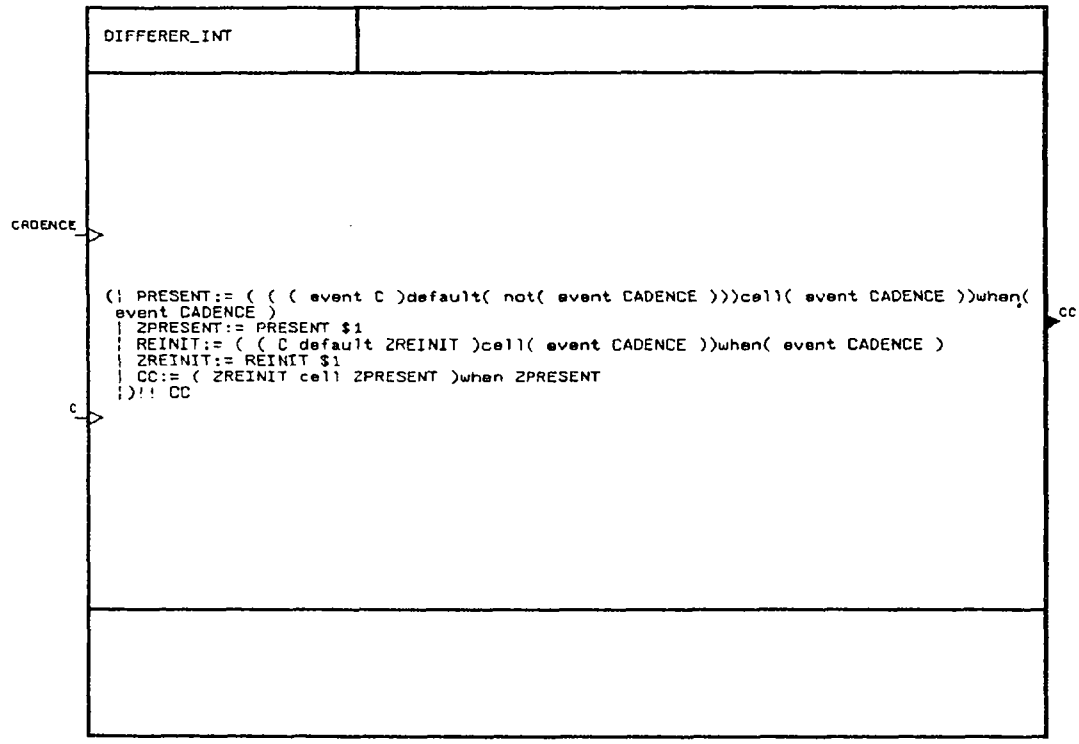

Figure B.14 : Le processus *DIFFERER*

Ce processus diffère d'un top d'horloge les valeurs d'un signal **C** en un signal **CC** par rapport à une horloge plus rapide appelée **CADENCE** (figure B.15).

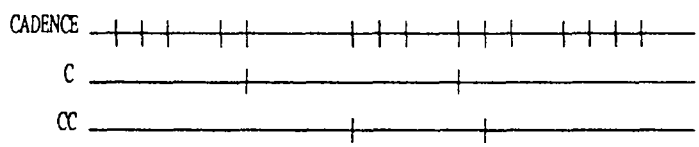

Figure B.15 : Exemple d'entrelacements des signaux du processus *DIFFERER*

Lorsque **C** et **CC** sont des entiers (resp. des booléens), le processus est appelé *DIFFERER_INT* (resp. *DIFFERER_LOG*).

B.13 processus *AUTOREPRODUCTION*

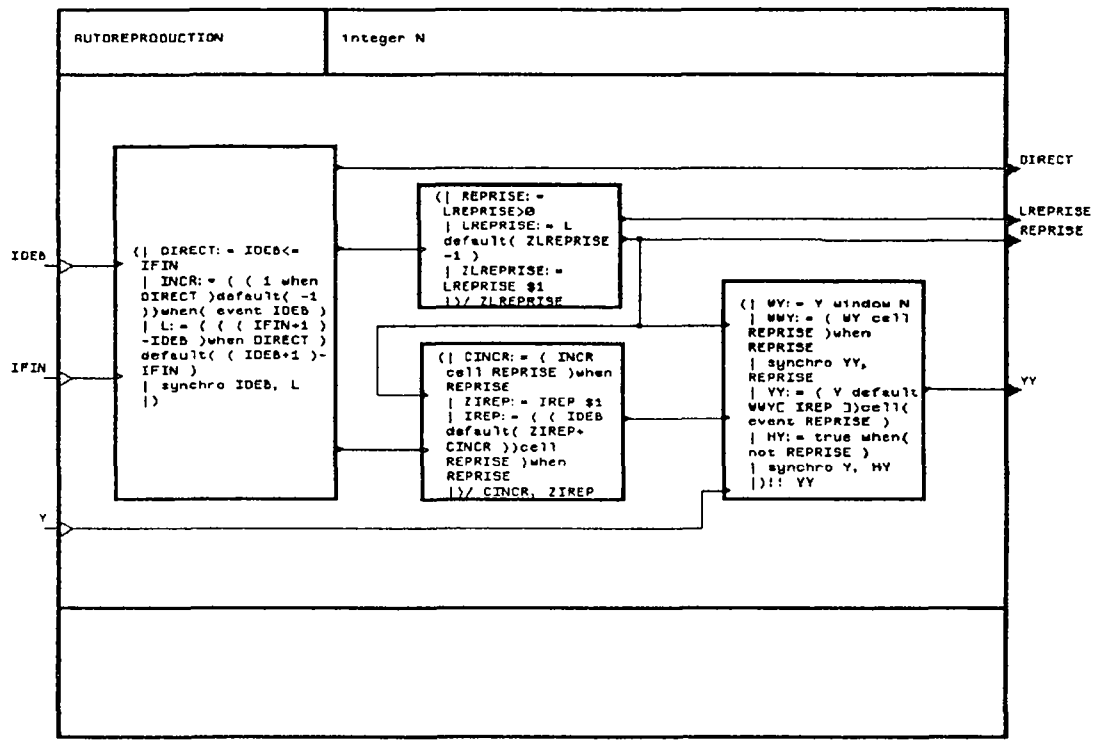

Figure B.16 : Le processus *AUTOREPRODUCTION*

Ce processus suréchantillonne le signal d'entrée **Y** en un signal **YY**. Le signal **Y** est stocké dans une fenêtre glissante. Lors de la présence des signaux **IDEB** et **IFIN**, le processus réinjecte les échantillons compris entre les indices **IDEB** et **IFIN** de la fenêtre.
Le paramètre **N** indique la taille de la fenêtre glissante.

Le signal **DIRECT** indique le sens de la réinjection (ou reprise); il vaut vrai lorsque **IDEB**≤**IFIN**, faux sinon. Le signal **L** indique la longueur de la reprise à effectuer. Le signal **INCR** indique la valeur de l'incrémentation pendant la reprise.

```
(| direct:= ideb ≤ ifin
 | incr:= ((1 when direct) default -1) when ( event ideb)
 | l:= (ifin+1-ideb when direct) default (ideb+1-ifin)
 | synchro ideb,l
 |)
```

L'horloge du signal **LREPRISE** est l'horloge du signal suréchantillonné. Il indique le nombre d'échantillons restant à réinjecter. **LREPRISE** est décrémenté de 1 lorsque les indices ne sont pas présents, et est réinitialisé à **L** lorsqu'ils sont présents. On est en période de reprise lorsque **LREPRISE** est positif (**REPRISE**=vrai).

```
(| reprise:= lreprise>0
 | lreprise:= l default (zlreprise-1)
 | zlreprise:=lreprise $ 1
 |)/zlreprise
```

Le calcul de l'indice courant dans la fenêtre glissante est donné par **IREP**. Il est réinitialisé à **IDEB** lorsque les indices sont présents, et est ensuite incrémenté ou décrémenté suivant la valeur de **INCR**. Ces calculs ne sont valides qu'en période de reprise. Les signaux sont donc resynchronisés avec le signal **REPRISE**=vrai.

```
(| cincr:= (incr cell reprise) when reprise
 | zirep:=irep $ 1
 | irep:= ((ideb default (zirep+cincr)) cell reprise) when reprise
 |)/cincr,zirep
```

Il suffit ensuite de synchroniser l'horloge de **Y** avec l'horloge du signal **REPRISE**=faux. L'horloge de **Y** n'est donc plus libre à l'extérieur du processus *AUTOCORRELATION*. Le signal suréchantillonné **YY** vaut **Y** lorsqu'on ne se trouve pas en période de reprise. Il prend la valeur de l'échantillon pointé par **IREP** en période de reprise.

```
(| wy:= y window n
 | wwy:= (wy cell reprise) when reprise
 | synchro yy,reprise
 | yy:= (y default wwy[irep]) cell ( event reprise)
 | hy:= true when (not reprise)
 | synchro y,hy
 |)!!yy
```

Annexe C

Bibliothèque de processus liés à l'exécution du programme

C.1 processus *TRACERTRAIT*

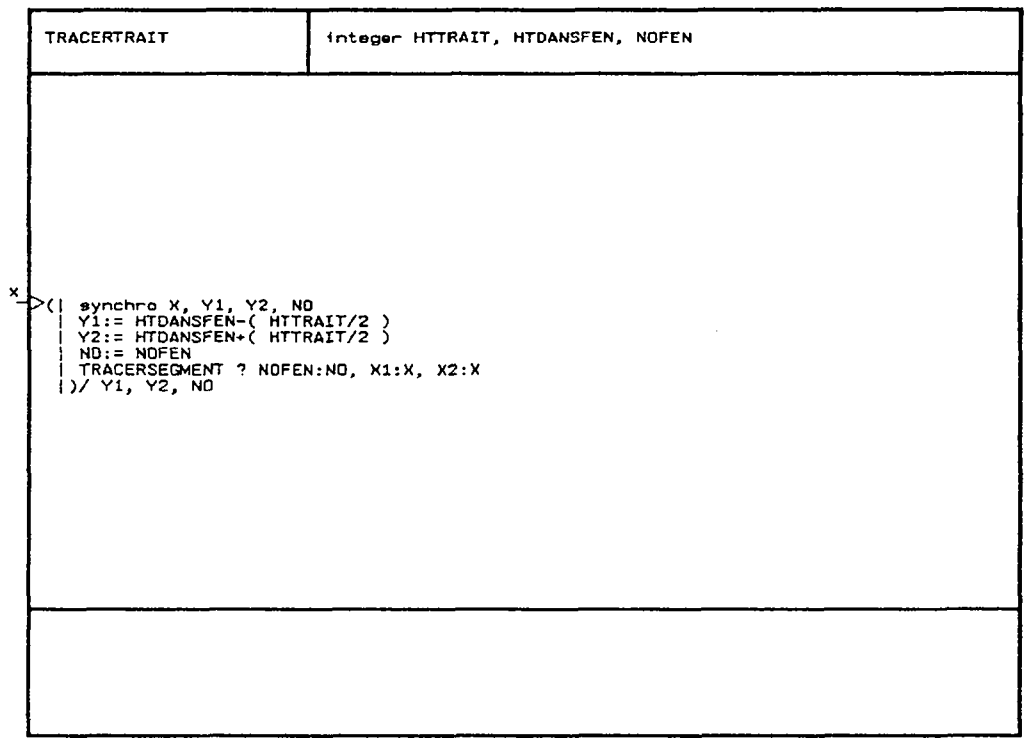

Figure C.1 : Le processus *TRACERTRAIT*

Ce processus trace un trait de hauteur **HTTRAIT** dans la fenêtre numéro **NOFEN** à une hauteur **HTDANSFEN** entre les points (**X**,**Y1**) et (**X**,**Y2**). A chaque événement **X**, le trait est tracé à l'aide de la procédure externe *TRACERSEGMENT*.

C.2 processus *SETWINDOW*

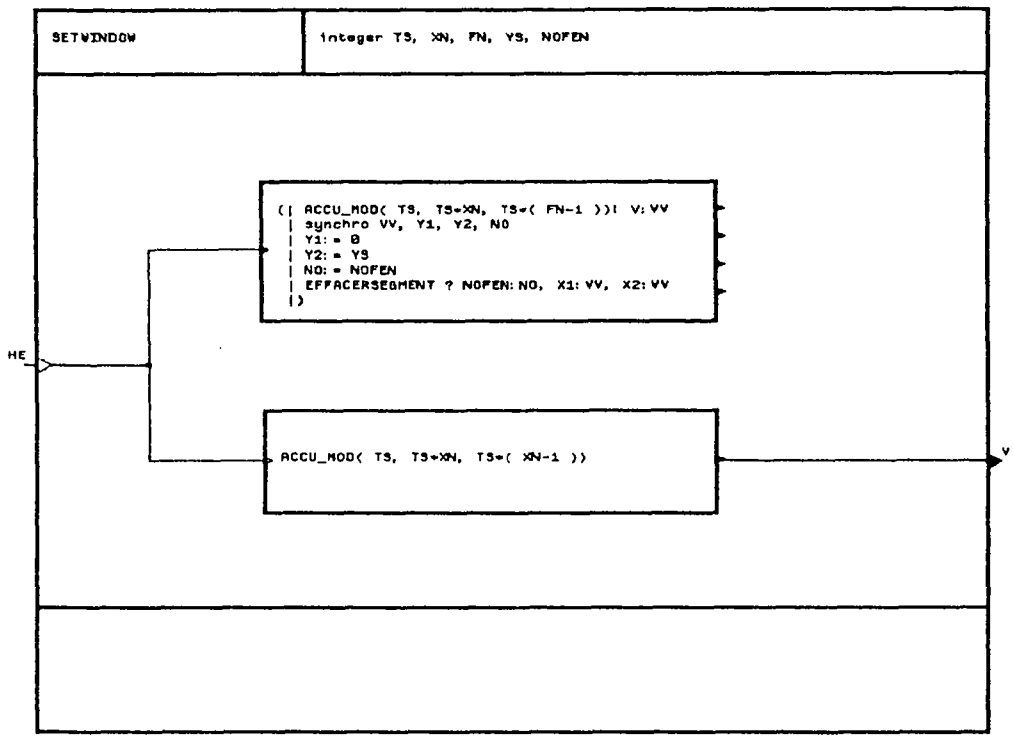

Figure C.2 : Le processus *SETWINDOW*

Ce processus gère l'écran de façon circulaire, c'est à dire qu'il gère une bande d'effacement de largeur **TS** (nombre de pixels) dans la fenêtre **NOFEN**. Deux boîtes noires composent ce processus:

- une première efface sur l'écran un rectangle de hauteur **YS** et de longueur **TS**. Au démarrage, l'effacement commence à l'abscisse (**TS***(**FN**-1)). La valeur du modulo est donnée en fonction de **TS** et de **XN**, longueur de la fenêtre
- une seconde compte le nombre d'événements **HE** modulo la valeur (**TS***XN).

Une utilisation de *SETWINDOW* est donnée dans le processus *TRACER-COURBE* (annexe C.3).

N.B: Le processus *ACCU_MOD* est situé en annexe B.6.
N.B: La procédure externe **EFFACERSEGMENT** efface dans la fenêtre de numéro **NOFEN** le rectangle entre les points (**X1**,**Y1**) et (**X2**,**Y2**)

C.3 processus *TRACERCOURBE*

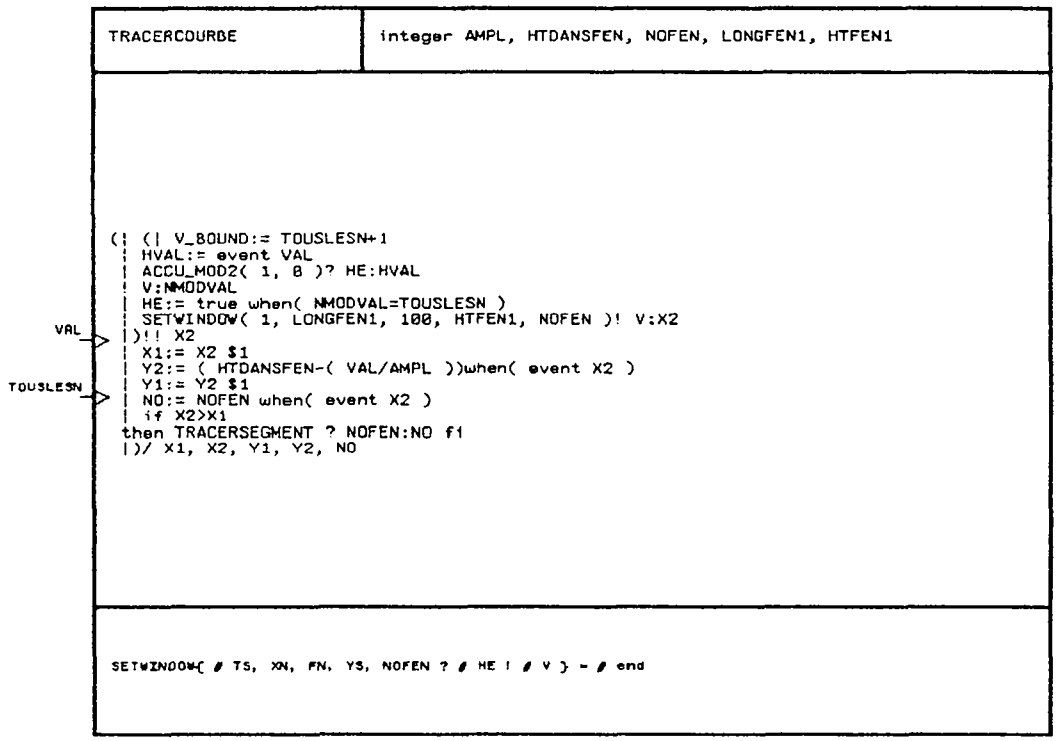

Figure C.3 : Le processus *TRACERCOURBE*

Ce processus trace la courbe du signal **VAL** en traçant tous les **TOUSLESN** points dans la fenêtre **NOFEN** à une hauteur **HTDANSFEN**.
Les paramètres **LONGFEN1** et **HTFEN1** représentent respectivement la longueur et la largeur de la fenêtre. **AMPL** est l'amplitude de la courbe.

Le processus *ACCU_BOUND2* (presque identique à *ACCU_BOUND*, annexe B.6, le modulo étant un signal d'entrée) est utilisé pour compter le nombre d'événements **VAL** modulo la longueur de l'écran. Puis *SETWINDOW* (annexe C.2) est utilisé pour gérer la bande d'effacement de l'écran de façon circulaire et positionner l'abscisse **X2** dans la fenêtre.
Le segment est tracé tous les **TOUSLESN** échantillons de **VAL** par *TRACER_SEGMENT*. Cette procédure externe est appelée à chaque événement **X2**

C.4 processus *INCRPANEL*

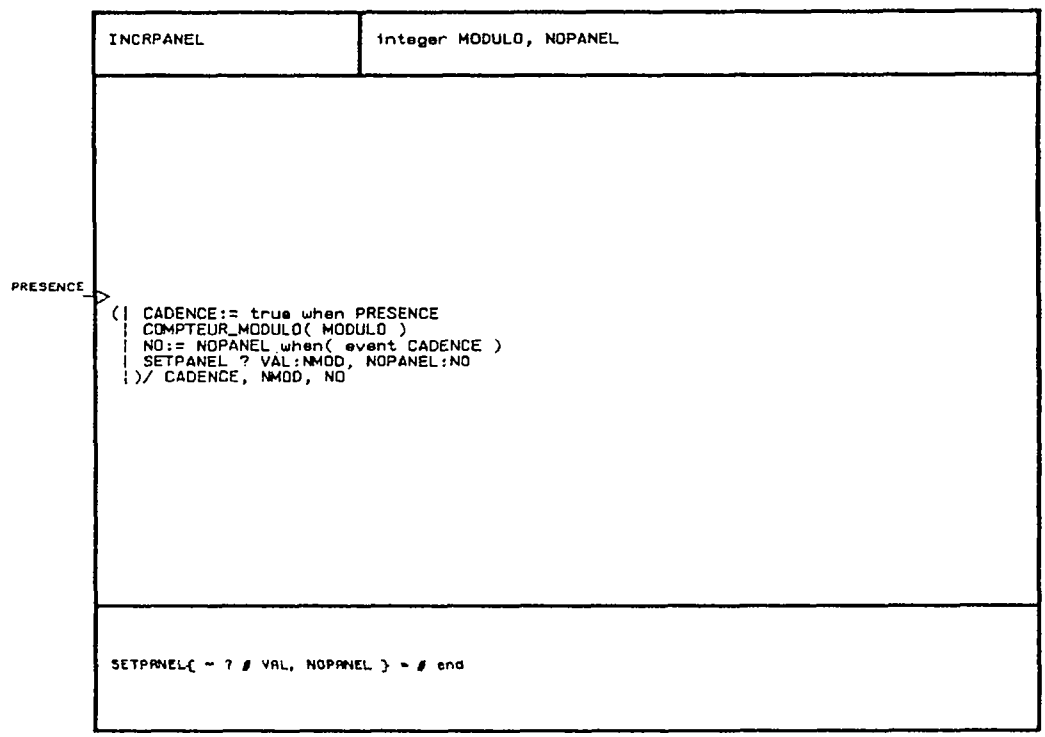

Figure C.4 : Le processus *INCRPANEL*

Ce processus utilise la procédure externe *SETPANEL*. Celle-ci est directement liée à un bouton de la fenêtre (le bouton numéro **NOPANEL**). Lorsque le signal **VAL** est présent, la valeur associée au bouton est augmentée de 1.
Le signal **VAL** est un compteur d'événements **PRESENCE**, modulo une valeur **MODULO**.

Ce processus est utilisé par exemple pour associer l'horloge d'un signal à un bouton (voir l'environnement autour du module de segmentation acoustique).

C.5 processus *DEBUG_PANEL*

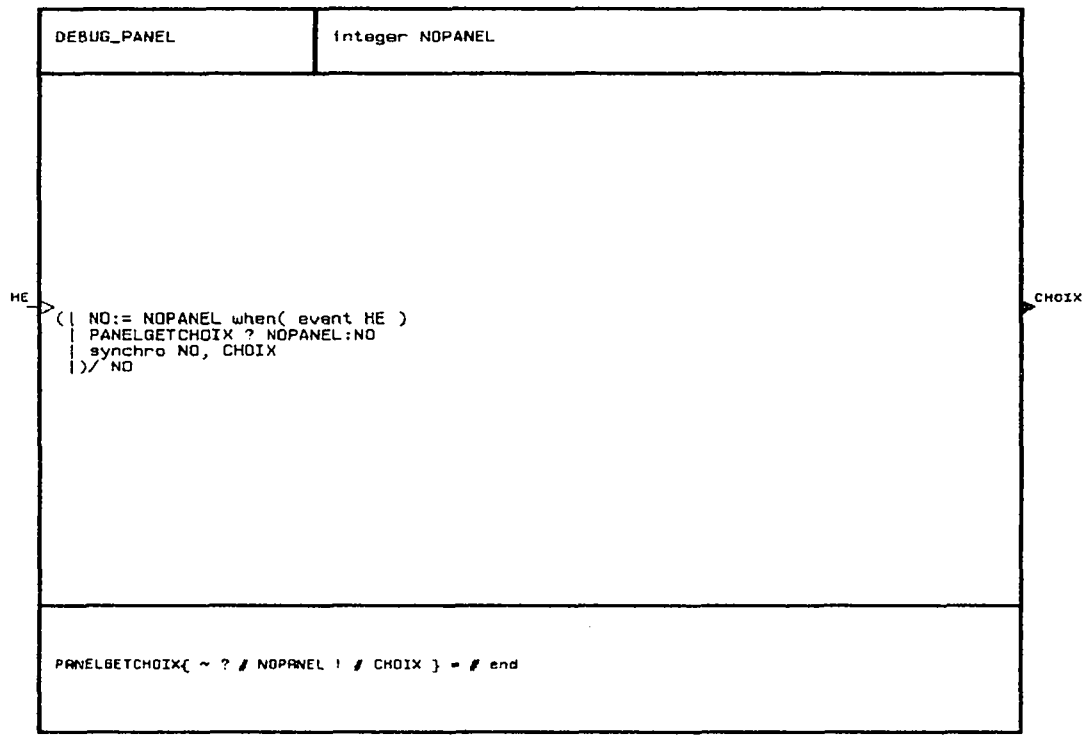

Figure C.5 : Le processus *DEBUG_PANEL*

Ce processus utilise la procédure externe *PANELGETCHOIX*. Celle-ci est directement liée à un bouton de la fenêtre (le bouton numéro **NOPANEL**). Lors de la présence de **HE**, le signal de sortie **CHOIX** de *PANELGETCHOIX* indique la valeur associée au bouton.

Ce processus est utilisé par exemple pour modifier les seuils, l'ordre des modèles... pendant l'exécution du programme.

Annexe D

Texte complet du programme

Le programme textuel du module de segmentation est présenté. Ce programme est exactement celui qui a été décrit au chapître 4.
Il a été compilé puis exécuté sur SUN 3/60, SUN 4/60 sous UNIX, ainsi que sur la VAXstation 3200 sous VMS, où les mesures ont été faites.

```
SEGMENTATION( integer KMIN, LIMITEBAK, MAXTAB
            ? real H, PAROLE
            ! integer VOIS, NUM, NUMF )
=    (| MODULE_SEGMENTATION( KMIN, LIMITEBAK, MAXTAB+LIMITEBAK )
     |)
where process

% ------------     BIBLIOTHEQUE DE MECANISMES SIGNAL  -----------------

     RESYNC1( ? real YIN;
              logical SYNC
            ! real YOUT )
     =    YOUT:= ( YIN cell SYNC )when SYNC
     end;
     RESYNCEVENT1( ? real YIN;
                   logical SYNC
                 ! real YOUT )
     =    YOUT:= ( YIN cell( event SYNC ))when( event SYNC )
     end;
     RESYNC2( ? integer YIN;
              logical SYNC
            ! integer YOUT )
     =    YOUT:= ( YIN cell SYNC )when SYNC
     end;
     RESYNCEVENT2( ? integer YIN;
                   logical SYNC
                 ! integer YOUT )
     =    YOUT:= ( YIN cell( event SYNC ))when( event SYNC )
     end;
     RESYNC3( ? logical YIN, SYNC
            ! logical YOUT )
     =    YOUT:= ( YIN cell SYNC )when SYNC
     end;
     RESYNCEVENT3( ? logical YIN, SYNC
                 ! logical YOUT )
     =    YOUT:= ( YIN cell( event SYNC ))when( event SYNC )
     end;
     RESYNC4( integer LTAB
            ? [ LTAB ]real WIN;
              logical SYNC
            ! [ LTAB ]real WOUT )
     =    WOUT:= ( WIN cell SYNC )when SYNC
     end;
     RESYNCEVENT4( integer LTAB
                 ? [ LTAB ]real WIN;
                   logical SYNC
                 ! [ LTAB ]real WOUT )
     =    WOUT:= ( WIN cell( event SYNC ))when( event SYNC )
     end;
     RESYNC5( integer LTAB
            ? [ LTAB ]integer WIN;
              logical SYNC
            ! [ LTAB ]integer WOUT )
     =    WOUT:= ( WIN cell SYNC )when SYNC
     end;
     RESYNCEVENT5( integer LTAB
                 ? [ LTAB ]integer WIN;
                   logical SYNC
                 ! [ LTAB ]integer WOUT )
     =    WOUT:= ( WIN cell( event SYNC ))when( event SYNC )
     end;
     COMPTEUR_AFTER_RAZ( ? logical CADENCE, RAZ
                         ! integer N )
     =    (| synchro N, CADENCE
          | N:= ( 1 when RAZ )default( ZN+1 )
          | ZN:= N $1
          |)!! N
     where
          integer ZN init 0
     end;
     COMPTEUR_MODULO( integer N
                    ? logical CADENCE
                    ! integer NMOD )
     =    (| synchro NMOD, CADENCE
          | NMOD:= ( 1 when( ZNMOD=N ))default( ZNMOD+1 )
          | ZNMOD:= NMOD $1
          |)!! NMOD
     where
          integer ZNMOD init 0
     end;
     COMPTEUR_MODULO_AFTER_RAZ( integer N
                              ? logical CADENCE, RAZ
                              ! integer NMOD )
     =    (| synchro CADENCE, NMOD, ZZNMOD
          | NMOD:= ( 1 when( ZZNMOD=N ))default( ZZNMOD+1 )
          | ZZNMOD:= ( 0 when RAZ )default ZNMOD
          | ZNMOD:= NMOD $1
          |)!! NMOD
     where
          integer ZNMOD init 0, ZZNMOD
     end;
     COMPTEUR_DIFF( ? logical CADENCE, L
                  ! integer DIFFN )
     =    (| NCADENCE:= #( event CADENCE )
          | RESYNCEVENT2 ? YIN:NCADENCE, SYNC:L
                         ! YOUT:NB
          | ZNB:= NB $1
          | DIFFN:= NB-ZNB
          |)!! DIFFN
     where
          integer NCADENCE, NB, ZNB init 0
     end;
     FLIPFLOP( ? logical SWITCH
             ! logical B, ZB )
     =    (| synchro SWITCH, B
          | B:= not ZB
          | ZB:= B $1
          |)
     end;
     FBY( ? logical A1, A2
        ! logical K )
     =    (| C:= ( false when A2 )default( true when A1 )
          | ZC:= C $1
          | K:= ( ( true when A1 )when A2 )default( ( true when ZC )when A2 )
          |)!! K
     where
          logical C, ZC init false
     end;
     INTERRUPT( ? real Y;
                logical CIN, COUT
              ! real YY )
     =    (| SWITCH:= ( true when CIN )default( false when COUT )
          | SORTIE:= SWITCH cell( event Y )
          | YY:= Y when SORTIE
          |)!! YY
     where
          logical SWITCH, SORTIE init true
     end;
```

```
AUTOREPRODUCTION[ integer N
                ? integer IDEB, IFIN;
                  real Y
                ! logical DIRECT;
                  integer LREPRISE;
                  logical REPRISE;
                  real YY ]
=    (| (| DIRECT:= IDEB<=IFIN
       | INCR:= ( ( 1 when DIRECT )default( -1 ))when( event IDEB )
       | L:= ( ( ( IFIN+1 )-IDEB )when DIRECT )default( ( IDEB+1 )-IFIN )
       | synchro IDEB, L
       |)! INCR:INCR_2, L:L_1
     | (| REPRISE:= LREPRISE>0
       | LREPRISE:= L default( ZLREPRISE-1 )
       | ZLREPRISE:= LREPRISE $1
       |)/ ZLREPRISE
         ? L:L_1
     | (| CINCR:= ( INCR cell REPRISE )when REPRISE
       | ZIREP:= IREP $1
       | IREP:= ( ( IDEB default( ZIREP+CINCR ))cell REPRISE )when
                REPRISE
       |)/ CINCR, ZIREP
         ? INCR:INCR_2
         ! IREP:IREP_3
     | (| WY:= Y window N
       | WWY:= ( WY cell REPRISE )when REPRISE
       | synchro YY, REPRISE
       | YY:= ( Y default WWY[ IREP ])cell( event REPRISE )
       | HY:= true when( not REPRISE )
       | synchro Y, HY
       |)!! YY
         ? IREP:IREP_3
    |)/ IREP_3, INCR_2, L_1
where
     [ N ]real WY, WWY;
     integer IREP_3, INCR_2, L_1, INCR, L, CINCR, IREP, ZIREP, ZLREPRISE
           init 0;
     logical REPRISE, HY
end;
DIFFERER_INT[ ? logical CADENCE;
              integer C
            ! integer CC ]
=    (| PRESENT:= ( ( ( event C )default( not( event CADENCE )))cell( event
                CADENCE ))when( event CADENCE )
     | ZPRESENT:= PRESENT $1
     | REINIT:= ( ( C default ZREINIT )cell( event CADENCE ))when( event
                CADENCE )
     | ZREINIT:= REINIT $1
     | CC:= ( ZREINIT cell ZPRESENT )when ZPRESENT
     |)!! CC
where
     logical PRESENT, ZPRESENT init false;
     integer REINIT, ZREINIT
end;
DIFFERER_LOG[ ? logical CADENCE, C
              ! logical CC ]
=    (| PRESENT:= ( ( ( event C )default( not( event CADENCE )))cell( event
                CADENCE ))when( event CADENCE )
     | ZPRESENT:= PRESENT $1
     | REINIT:= ( ( C default ZREINIT )cell( event CADENCE ))when( event
                CADENCE )
     | ZREINIT:= REINIT $1
     | CC:= ( ZREINIT cell ZPRESENT )when ZPRESENT
     |)!! CC
where
     logical PRESENT, ZPRESENT init false, REINIT, ZREINIT
end;

% ------------     BIBLIOTHEQUE DE MODULES STANDARDS  -----------------

NB_PASSAGES_ZERO[ integer FENETRE
                ? [ FENETRE ]real WY
                ! integer NBCHGT ]
=    (| synchro WY, SIGNE, CHGTSIGNE
     | ZERO:= 0 when( event WY )
     |
       array I to FENETRE
            of (& SIGNE:= ( ( -1 )when( WY<0 e 0 ))default( 1 when( WY>0
                          e 0 ))default SIGNE
                & CHGTSIGNE:= ( ( CHGTSIGNE+1 )when( ( ( SIGNE=1 )and( WY
                              <0 e 0 ))or( ( SIGNE=( -1 ))and( WY>0 e 0 )
                              )))default CHGTSIGNE
                &)
            with SIGNE[0]:ZERO, CHGTSIGNE[0]:ZERO, WY
            end
     | NBCHGT:= CHGTSIGNE[ FENETRE ]
     |)!! NBCHGT
where
     [ FENETRE ]integer SIGNE, CHGTSIGNE;
     integer ZERO
end;
AUTOCORRELATION1[ integer M, FENETRE
                ? real Y;
                  integer NY;
                  logical RAZ
                ! [ FENETRE+1 ]real WY;
                  [ M+1 ]real R ]
=    (| WY:= Y window FENETRE+1
     | ZR:= R $1
     |
       array I to M+1
            of (| RO:= ( ( ( Y*Y )when( I=1 ))default 0 e 0 )when RAZ
                | R1:= ZR when( NY<=( I-1 ))
                | R2:= ( ZR+( WY[ ( FENETRE-I )+2 ]*WY[ FENETRE+1 ]))when
                       ( NY<=FENETRE )
                | R3:= ( ZR+( WY[ ( FENETRE-I )+2 ]*WY[ FENETRE+1 ]))-(
                      WY[ 1 ]*WY[ I ])
                | R:= R0 default R1 default R2 default R3
                |)/ R0, R1, R2, R3
            with ZR
            end
     |)!! WY, R
where
     [ M+1 ]real ZR;
     real R0, R1, R2, R3
end;
AUTOCORRELATION2[ integer M, FENETRE
                ? [ FENETRE ]real WY
                ! [ M+1 ]real R ]
=    (| ZERO:= 0 e 0 when( event WY )
     |
       array I to M+1
            of (| synchro WY, CALCUL
                |
                  array J to FENETRE
                       of CALCUL:= ( ( CALCUL+( WY[ J ]*WY[ J-( I-1 )]))
                                   when( J>M ))default CALCUL
                       with CALCUL[.]:ZERO
```

```
                    end
                | R:= CALCUL[ 1 ]/( FENETRE-M )
                |)/ CALCUL
            end
      |)!! R
where
     [ FENETRE ]real CALCUL;
     real ZERO
end;
AUTO[ integer M, FENETRE
      ? real Y;
        integer NY;
        logical RAZ, ACTIF
      ! real ALPH, XAU ]
=     (| ZERO:= 0 e 0
      | (| AUTOCORRELATION1( M, FENETRE )
        |
          array I to M+1
                of RACTIF:= ( ( 1 e 0 when( ( I=1 )and( R<=0 e 0 )))
                              default R )when ACTIF
                with R
                end
        | WYACTIF:= WY when ACTIF
        |)!! RACTIF, WYACTIF
      | AU1:= [ [ to M+1 ]: 0 e 0,[ 1 ]: 1 e 0 ]
      | R1:= RACTIF[ 1 ]
      |
        array INC to M
              of (|
                    array J to M
                          of CALCULS:= ( CALCULS when( ALPHAI<=0 e 0 ))
                                        default( ( ( CALCULS+( RACTIF[ ( INC
                                              -J )+2 ]*AUI ))when( J<=INC ))
                                        default CALCULS )
                          with CALCULS[0]:ZERO, AUI
                          end
                  | S:= CALCULS[ M ]
                  | RCA:= -( S/ALPHAI )
                  |
                    array J to M+1
                          of AU:= ( AU when( ALPHAI<=0 e 0 ))default( AU
                                  when( J=1 ))default( ( AU+( RCA*AU[ INC-1
                                                ,( INC-J )+2 ]))when( J<( INC+
                                  1 )))default( RCA when( J=( INC+1 )))
                                  default AU
                          with AU
                          end
                  | ALPHA:= ( ALPHA when( ALPHA<=0 e 0 ))default( ALPHA+( S
                            *RCA ))
                  |)? ALPHAI:ALPHA, AUI:AU
                    / CALCULS, S, RCA
              with ALPHA[0]:R1, AU[0]:AU1
              end
      | ALPH:= ( ( ALPHA[ M ]/( FENETRE-1 ))when( ALPHA[ M ]>0 e 0 ))
               default ALPHA[ M ]
      | (|
           array I to M+1
                 of CALCULXAU:= CALCULXAU+( AU[ M, I ]*WYACTIF[ ( FENETRE-I
                                                )+2 ])
                 with CALCULXAU[0]:ZERO
                 end
        | XAU:= ( CALCULXAU[ M+1 ]when( ALPHA[ M ]>0 e 0 ))default WYACTIF
                [ 1 ]
        |)/ CALCULXAU
      |)!! ALPH, XAU
where
     [ M, M+1 ]real AU;
     [ FENETRE+1 ]real WY, WYACTIF;
     [ M+1 ]real R, RACTIF, AU1, AUI, CALCULXAU;
     [ M ]real CALCULS, ALPHA;
     real ZERO, R1, S, ALPHAI, RCA
end;
BURG[ integer M
      ? real Y;
        integer NY;
        logical RAZ
      ! real VARA, RES ]
=     (| (| ZX:= X $1
         | X:= ( Y when RAZ )default( ( Y+1 )when( Y=ZX ))default Y
         |)
       | (| ZCZERO:= CZERO $1
         | ZVFZERO:= VFZERO $1
         | EZERO:= X
         | ZFZERO:= ( 0 e 0 when( NY=1 ))default ZX
         | FZERO:= X
         | CZERO:= ( 0 e 0 when( NY=1 ))default( ZCZERO+( ( ( X*ZX )-ZCZERO
                   )/( NY-1 )))
         | (; VFZERO:= ( ( X*X )when( NY-1 ))default( ZVFZERO+( ( ( X*X )-
                       ZVFZERO )/NY ))
           ; VBZERO:= VFZERO
           ;)
         | KZERO:= ( 2*CZERO )/( VFZERO+VBZERO )
         |)/ ZCZERO, ZVFZERO
       | (| ZF:= F $1
         | ZC:= C $1
         | ZVF:= VF $1
         | ZVB:= VB $1
         |
           array I to M
                 of (| EE:= ( EEI when( I>=NY ))default( EEI-( KI*ZF ))
                     | F:= ( F when( I>=NY ))default( ZF-( KI*EEI ))
                     | VF:= ( VF when( I>=NY ))default( ZVF+( ( ( EE*EE )-
                            ZVF )/( NY-I )))
                     | VB:= ( VB when( I>=NY ))default( ZVB+( ( ( F*F )-ZVB
                            )/( NY-I )))
                     | C:= ( C when( I>=( NY-1 )))default( ZC+( ( ( EE*ZF[
                                                  I ])-ZC )/( ( NY-1 )-I )))
                     | K:= ( K when( I>=( NY-1 )))default( ( 2*C )/( VF+VB
                           ))
                     |)? KI:K, EEI:EE
                 with K[0]:KZERO, EE[0]:EZERO, ZF[0]:ZFZERO, F[0]:FZERO, VF
                      [0]:VFZERO, VB[0]:VBZERO, C[0]:CZERO, ZC, ZVF, ZVB
                 end
         | VARA:= VF[ M ]
         | RES:= EE[ M ]
         |)
       |)!! VARA, RES
where
     [ M ]real EE, F, ZF, K, C, ZC, VF, ZVF, VB, ZVB;
     real X, ZX init 0 e 0, KZERO, EZERO, FZERO, ZFZERO, VFZERO, ZVFZERO,
          VBZERO, CZERO, ZCZERO, KI, EEI
end;
MODELE[ integer M, FENETRE
        ? real Y;
          logical SRAZ
        ! real VCOURT, ERRCOURT, VLONG, ERRLONG ]
=     (| RAZ:= SRAZ when( event Y )
       | AUTO( M, FENETRE )! ALPH:V_C, XAU:ERR_C
```

```
        | BURG( M )! VARA:V_L, RES:ERR_L
        | (| HY:= event Y
          | COMPTEUR_AFTER_RAZ ? CADENCE:HY
                              ! N:NY
          |)/ HY
        | (| ACTIF:= NY>=FENETRE
          | RESYNC1 ? YIN:V_C, SYNC:ACTIF
                    ! YOUT:VCOURT
          | RESYNC1 ? YIN:ERR_C, SYNC:ACTIF
                    ! YOUT:ERRCOURT
          | RESYNC1 ? YIN:V_L, SYNC:ACTIF
                    ! YOUT:VLONG
          | RESYNC1 ? YIN:ERR_L, SYNC:ACTIF
                    ! YOUT:ERRLONG
          |)
        |)!! VCOURT, ERRCOURT, VLONG, ERRLONG
    where
        real V_C, ERR_C, V_L, ERR_L;
        logical HY, ACTIF, RAZ;
        integer NY
    end;
    STAT_HINKLEY[ ? real VCOURT, ERRCOURT, VLONG, ERRLONG, BIAI;
                    logical SRAZ
                  ! real U ]
    =   (| RAZ:= SRAZ when( event VCOURT )
         | QV:= VCOURT/VLONG
         | DISTANCE:= ( ( ( ( 2*ERRCOURT*ERRLONG )/VLONG )-( ( ERRLONG*ERRLONG
                      *( 1+QV ))/VLONG ))+( QV-1 ))/( 2*QV )
         | synchro DISTANCE, ZZU
         | ZU:= U $1
         | ZZU:= ( 0 e 0 when RAZ )default ZU
         | U:= ( ZZU+DISTANCE )-BIAI
         |)!! U
    where
        real QV, DISTANCE, ZU init 0 e 0, ZZU;
        logical RAZ
    end;
    MAX[ ? real U;
           logical SRAZ
         ! real MAX;
           logical NOUVMAX ]
    =   (| RAZ:= SRAZ when( event U )
         | synchro U, ZZMAX
         | ZMAX:= MAX $1
         | ZZMAX:= ( U when RAZ )default ZMAX
         | NOUVMAX:= true when( U>=ZZMAX )
         | MAX:= ( U when( event NOUVMAX ))default ZZMAX
         |)!! MAX, NOUVMAX
    where
        real ZMAX init 0 e 0, ZZMAX;
        logical RAZ
    end;
    CONVOLUTION_LINEAIRE[ integer NFILT
                          ? real Y, H
                          ! real YFILT ]
    =   (| (| WWY:= Y window NFILT+1
           | WY:= WWY when ACTIF
           |)!! WY
             ? ACTIF:ACTIF_1
             ! WY:WY_2
         | (| HH:= true when( not ACTIF )
           | synchro H, HH
           |)/ HH
             ? ACTIF:ACTIF_1
         | (| WWH:= H window NFILT
           | RESYNC4( NFILT )? WIN:WWH, SYNC:ACTIF
                             ! WOUT:WH
           |)!! WH
             ? ACTIF:ACTIF_1
             ! WH:WH_3
         | (| NY:= #( event Y )
           | ACTIF:= NY>NFILT
           |)!! ACTIF
             ! ACTIF:ACTIF_1
         | (| ZERO:= 0 e 0 when( event WY )
           |
             array J to NFILT
               .     of PLUS:= ( WH[ ( NFILT+1 )-J ]*WY )+PLUS
                     with PLUS[0]:ZERO, WY
                     end
           | YFILT:= ( PLUS[ NFILT ]default 0 e 0 )when( event Y )
           |)!! YFILT
             ? WH:WH_3, WY:WY_2
        |)/ WH_3, WY_2, ACTIF_1
    where
        [ NFILT+1 ]real WY, WY_2, WWY;
        [ NFILT ]real WH, WH_3, WWH, PLUS;
        real ZERO;
        integer NY;
        logical ACTIF, ACTIF_1, HH
    end;
    CALCUL_VOISEMENT[ real SEUI1, SEUI2, SEUI3, SEUI4, SEUI5;
                      integer FENETRE
                      ? [ FENETRE ]real WY
                      ! integer VOISEMENT ]
    =   (| NB_PASSAGES_ZERO( FENETRE )! NBCHGT:NBCHGT_1
         | AUTOCORRELATION2( 1, FENETRE )! R:R_2
         | (| T2:= ( ( 0 when( R[ 1 ]<=SEUI1 ))default( 1 when( R[ 1 ]<=SEUI2
                  ))default 2 )when( event WY )
           | T1:= ( ( 0 when( R[ 2 ]<=SEUI3 ))default( 1 when( R[ 2 ]<=SEUI4
                ))default 2 )when( event WY )
           | T:= ( T1+T2 )/2
           | CVOIS:= ( ( T=2 )and( R[ 2 ]<0 e 0 ))or( ( T=2 )and( ( R[ 2 ]/R[
                     1 ])<8.5 e-1 )and( R[ 1 ]<( 1 e 7/1.28 e 2 )))or( ( T=0
                     )and( ( R[ 2 ]/R[ 1 ])<7 e-1 )and( R[ 1 ]>( 2 e 4/1.28 e
                     2 )))or( ( T=2 )and( T2=2 )and( NBCHGT>=( SEUI5*FENETRE
                     )))
           | VOISEMENT:= ( ( ( 1 when CVOIS )default T )cell( event WY ))when
                         ( event WY )
           |)!! VOISEMENT
             ? R:R_2, NBCHGT:NBCHGT_1
        |)/ R_2, NBCHGT_1
    where
        [ 2 ]real R, R_2;
        integer NBCHGT, NBCHGT_1, T, T1, T2;
        logical CVOIS
    end;

% ------------------------------------------------------------
% ---------------     PROGRAMME PRINCIPAL  -------------------
% ------------------------------------------------------------

    MODULE_SEGMENTATION[ integer KMIN, LIMITEBAK, MAXTAB
                         ? real H, PAROLE
                         ! integer NUMF, NUM, VOIS ]
    =   (| MODULE_FILTRE( KMIN )! F_EVENTUELLE:F_EVENTUELLE_1, FRONTIERE:NUMF
         | MODULE_FORWARD_BACKWARD( KMIN, LIMITEBAK, MAXTAB )?
                                                F_HKF:F_EVENTUELLE_1
```

```
        |)/ F_EVENTUELLE_1
    where
        integer F_EVENTUELLE_1
    process

% ---------------      MODULE_FILTRE  ---------------------

        MODULE_FILTRE( integer KMIN
                       ? real H, PAROLE
                       ! integer FRONTIERE, F_EVENTUELLE )
        =    (| FILTRAGE( 128 )? Y:PAROLE
                             ! YFILT:YFILT_1
             | HINKLEY_OPT( -0.8 e 0, 80.e 0, KMIN )? Y:YFILT_1
             |)/ YFILT_1
        where
            YFILT_1
        process
            FILTRAGE( integer NFILT
                      ? real H, Y
                      ! real YFILT )
            =    (| CONVOLUTION_LINEAIRE( NFILT )
                 |)
            end;
            HINKLEY_OPT( real BIAIS, SEUIL;
                         integer KMIN
                         ? real Y
                         ! integer FRONTIERE, F_EVENTUELLE )
            =    (| (NY:= #HY)? HY:HY_1
                            ! NY:NY_2
                 | MODELE( 4, KMIN )? SRAZ:SRAZ_7
                                    ! ERRLONG:ERRLONG_6, VLONG:VLONG_5,
                                      ERRCOURT:ERRCOURT_4, VCOURT:VCOURT_3
                 | COMMUT ? SECONDRAZ:SECONDRAZ_18, COMMUT:COMMUT_17,
                            ERRLONG_B:ERRLONG_B_11, VLONG_B:VLONG_B_10,
                            ERRCOURT_B:ERRCOURT_B_9, VCOURT_B:VCOURT_B_8,
                            ERRLONG_A:ERRLONG_6, VLONG_A:VLONG_5,
                            ERRCOURT_A:ERRCOURT_4, VCOURT_A:VCOURT_3,
                            HY:HY_1
                          ! ERRLONG:ERRLONG_16, VLONG:VLONG_15,
                            ERRCOURT:ERRCOURT_14, VCOURT:VCOURT_13,
                            RAZ_B:RAZ_B_12, RAZ_A:SRAZ_7
                 | (| STAT_HINKLEY ? SRAZ:SRAZ_20, BIAI:BIAI_19,
                                     ERRLONG:ERRLONG_16, VLONG:VLONG_15,
                                     ERRCOURT:ERRCOURT_14, VCOURT:VCOURT_13
                                   ! U:U_21
                    | MAX ? U:U_21, SRAZ:SRAZ_20
                          ! NOUVMAX:NOUVMAX_23, MAX:MAX_22
                    | (| (| SEUI:= SEUIL when( event U )
                         | DETECTION:= true when( ( MAX-U )>=SEUI )
                         |)!! DETECTION
                       | (| DIFFERER_LOG ? CADENCE:HY, C:DETECTION
                                          ! CC:SWITCH
                         | HSECONDRAZ:= ( event NOUVMAX )default( event
                                          DETECTION )
                         | DIFFERER_LOG ? CADENCE:HY, C:HSECONDRAZ
                                          ! CC:SECONDRAZ
                         |)!! SWITCH, SECONDRAZ
                       | (| HU:= event U
                         | FBY ? A1:SWITCH, A2:HU
                               ! K:RAZTEST
                         |)!! RAZTEST
                       | (| POS_MAX:= ( 0 when( NY=1 ))default( NY when(
                                       event NOUVMAX ))
                         | F_EVENTUELLE:= POS_MAX cell( event HY )
                         | FRONTIERE:= F_EVENTUELLE when( event DETECTION )
                         |)!! F_EVENTUELLE, FRONTIERE
                       |)/ DETECTION
                         ? NOUVMAX:NOUVMAX_23, MAX:MAX_22, U:U_21, NY:NY_2,
                           HY:HY_1
                         ! RAZTEST:SRAZ_20, SECONDRAZ:SECONDRAZ_18,
                           SWITCH:COMMUT_17
                    | (BIAI:= BIAIS when( event VCOURT )? VCOURT:VCOURT_13
                                                       ! BIAI:BIAI_19
                    |)/ NOUVMAX_23, MAX_22, U_21, SRAZ_20, BIAI_19
                 | (HY:= event Y)! HY:HY_1
                 | MODELE( 4, KMIN )? SRAZ:RAZ_B_12
                                    ! ERRLONG:ERRLONG_B_11,
                                      VLONG:VLONG_B_10,
                                      ERRCOURT:ERRCOURT_B_9,
                                      VCOURT:VCOURT_B_8
                 |)/ SECONDRAZ_18, COMMUT_17, ERRLONG_16, VLONG_15,
                     ERRCOURT_14, VCOURT_13, RAZ_B_12, ERRLONG_B_11,
                     VLONG_B_10, ERRCOURT_B_9, VCOURT_B_8, SRAZ_7, ERRLONG_6,
                     VLONG_5, ERRCOURT_4, VCOURT_3, NY_2, HY_1
        where
            real MAX, MAX_22, U, U_21, BIAI_19, ERRLONG_16, VLONG_15,
                 ERRCOURT_14, VCOURT_13, ERRLONG_B_11, VLONG_B_10,
                 ERRCOURT_B_9, VCOURT_B_8, ERRLONG_6, VLONG_5, ERRCOURT_4
                 , VCOURT_3, BIAI, SEUI;
            integer NY, NY_2, POS_MAX;
            logical NOUVMAX, NOUVMAX_23, RAZTEST, SRAZ_20, SECONDRAZ_18,
                    SWITCH, COMMUT_17, RAZ_B_12, SRAZ_7, HY_1, DETECTION,
                    SECONDRAZ, HSECONDRAZ, HU
        process
            COMMUT( ? logical HY;
                      real VCOURT_A, ERRCOURT_A, VLONG_A, ERRLONG_A,
                           VCOURT_B, ERRCOURT_B, VLONG_B, ERRLONG_B;
                      logical SECONDRAZ, COMMUT
                    ! real VCOURT, ERRCOURT, VLONG, ERRLONG;
                      logical RAZ_A, RAZ_B )
            =    (| (| FLIPFLOP ? SWITCH:COMMUT
                                ! B:BASCULE, ZB:ZBASCULE
                                / ZBASCULE
                     | RESYNCEVENT3 ? YIN:BASCULE, SYNC:HY
                                    ! YOUT:MAIN_A
                     |)/ BASCULE
                   | if MAIN_A
                     then (| VCOURT:= VCOURT_A
                           | ERRCOURT:= ERRCOURT_A
                           | VLONG:= VLONG_A
                           | ERRLONG:= ERRLONG_A
                           | RAZ_B:= SECONDRAZ
                           |)
                     else (| VCOURT:= VCOURT_B
                           | ERRCOURT:= ERRCOURT_B
                           | VLONG:= VLONG_B
                           | ERRLONG:= ERRLONG_B
                           | RAZ_A:= SECONDRAZ
                           |)fi
                 |)/ MAIN_A
            where
                logical BASCULE, ZBASCULE init true, MAIN_A init true
            end
        end;

% ---------------      MODULE_FORWARD_BACKWARD  ---------------------
```

```
MODULE_FORWARD_BACKWARD( integer KMIN, LIMITEBAK, MAXTAB
                       ? integer F_HKF, NUMF;
                         real PAROLE
                       ! integer NUM, VOIS )
=    (| TEST_HINKLEY( KMIN )? SEUIL:SEUIL_4, BIAIS:BIAIS_3, RAZ:RAZ_2,
                              Y:Y_1
                            ! DETECTION:DETECTION_6,
                              F_EVENTUELLE:F_EVENTUELLE_5
     | TEST_VOISEMENT( KMIN )? RAZ:RAZ_8, Y:Y_7
                            ! VOIS:VOIS_11, DETECTION:DETECTION_10,
                              F_EVENTUELLE:F_EVENTUELLE_9
     | GESTION_INFO( KMIN )? SWITCH_FB:SWITCH_FB_17,
                             REDEMARRAGE:REDEMARRAGE_16,
                             HRAPIDE:HRAPIDE_15, DET_V:DETECTION_10,
                             LSEG_V:F_EVENTUELLE_9,
                             DET_HK:DETECTION_6,
                             LSEG_HK:F_EVENTUELLE_5
                           ! LSEGMENTB:LSEGMENTB_14,
                             LSEGMENTF:LSEGMENTF_13, QUI:QUI_12
     | GESTION_MEMOIRE( KMIN, LIMITEBAK, MAXTAB )?
                                        LSEGMENTB:LSEGMENTB_14,
                                        LSEGMENTF:LSEGMENTF_13,
                                        QUI:QUI_12, VOIS1:VOIS_11,
                                        STOPV:DETECTION_10,
                                        STOPHK:DETECTION_6
                                      ! SWITCH_FB:SWITCH_FB_17,
                                        REDEMARRAGE:REDEMARRAGE_16,
                                        HRAPIDE:HRAPIDE_15,
                                        RAZ_V:RAZ_8, Y_V:Y_7,
                                        SEUIL:SEUIL_4, BIAIS:BIAIS_3,
                                        RAZ_HK:RAZ_2, Y_HK:Y_1
     |)/ SWITCH_FB_17, REDEMARRAGE_16, HRAPIDE_15, LSEGMENTB_14,
        LSEGMENTF_13, QUI_12, VOIS_11, DETECTION_10, F_EVENTUELLE_9,
        RAZ_8, Y_7, DETECTION_6, F_EVENTUELLE_5, SEUIL_4, BIAIS_3,
        RAZ_2, Y_1
where
     integer LSEGMENTB_14, LSEGMENTF_13, QUI_12, VOIS_11,
             F_EVENTUELLE_9, F_EVENTUELLE_5;
     logical SWITCH_FB_17, REDEMARRAGE_16, HRAPIDE_15, DETECTION_10,
             RAZ_8, DETECTION_6, RAZ_2;
     real Y_7, SEUIL_4, BIAIS_3, Y_1
process
     GESTION_MEMOIRE( integer KMIN, LIMITEBAK, MAXTAB
                    ? logical STOPHK, STOPV;
                      integer VOIS1;
                      real PAROLE;
                      integer LSEGMENTF, LSEGMENTB, QUI
                    ! real Y_HK;
                      logical RAZ_HK;
                      real BIAIS, SEUIL, Y_V;
                      logical RAZ_V, REDEMARRAGE, HRAPIDE, SWITCH_FB
                      ;
                      integer NUM, VOIS )
     =    (| AUTOREPRODUCTION( MAXTAB )? IDEB:IDEB_2, IFIN:IFIN_1,
                                         Y:PAROLE
                                       ! DIRECT:DIRECT_6, YY:YY_5,
                                         REPRISE:REPRISE_4,
                                         LREPRISE:LREPRISE_3
          | ALIMENTATION_TESTS ? YCOUR:YCOUR_10, SENSCOUR:SENSCOUR_9,
                                 ONSEMETB:ONSEMETB_8,
                                 ONSEMETF:ONSEMETF_7
          | GESTION_INDICES( KMIN, LIMITEBAK, MAXTAB )?
                                         FINRETOURARRIERE:
                                         FINRETOURARRIERE_12,
                                         HPAROLE:HPAROLE_11,
                                         LSEGMENTBI:LSEGMENTB
                                       ! IDEB:IDEB_2, IFIN:IFIN_1
          | GESTION_RETARD ? DIRECT:DIRECT_6, YY:YY_5,
                             REPRISE:REPRISE_4, LREPRISE:LREPRISE_3
                           ! FINRETOURARRIERE:FINRETOURARRIERE_12,
                             HPAROLE:HPAROLE_11, YCOUR:YCOUR_10,
                             SENSCOUR:SENSCOUR_9, ONSEMETB:ONSEMETB_8,
                             ONSEMETF:ONSEMETF_7
          |)/ FINRETOURARRIERE_12, HPAROLE_11, YCOUR_10, SENSCOUR_9,
              ONSEMETB_8, ONSEMETF_7, DIRECT_6, YY_5, REPRISE_4,
              LREPRISE_3, IDEB_2, IFIN_1
where
     real YCOUR_10 init 0 e 0, YY_5;
     integer LREPRISE_3, IDEB_2, IFIN_1;
     logical FINRETOURARRIERE_12, HPAROLE_11, SENSCOUR_9 init
             true, ONSEMETB_8, ONSEMETF_7, DIRECT_6, REPRISE_4, C
process
     GESTION_RETARD( ? logical REPRISE;
                       integer LREPRISE;
                       real YY;
                       logical DIRECT
                     ! logical HPAROLE, ONSEMETF, ONSEMETB;
                       real YCOUR init 0 e 0;
                       logical REDEMARRAGE, HRAPIDE, SWITCH_FB,
                               FINRETOURARRIERE, SENSCOUR init
                               true )
     =    (| HRAPIDE:= event YY
          | (| SENS:= ( DIRECT cell( event HRAPIDE ))when( event
                        HRAPIDE )
               | SENSCOUR:= SENS $1
               |)!! SENSCOUR
          | YCOUR:= YY $1
          | (| ONVASEMETTREF:= true when DIRECT
               | DIFFERER_LOG ? CADENCE:HRAPIDE, C:ONVASEMETTREF
                              ! CC:ONSEMETF
               |)!! ONSEMETF
          | (| ONVASEMETTREB:= true when( not DIRECT )
               | DIFFERER_LOG ? CADENCE:HRAPIDE, C:ONVASEMETTREB
                              ! CC:ONSEMETB
               |)!! ONSEMETB
          | REDEMARRAGE:= ONSEMETF default ONSEMETB
          | (| SWITCH:= ( true when ONSEMETF )default( false
                          when ONSEMETB )
               | ZSWITCH:= SWITCH $1
               | SWITCH_FB:= true when( SWITCH/=ZSWITCH )
               |)!! SWITCH_FB
          | (| REPRISECOUR:= REPRISE $1
               | HPAROLE:= true when( not REPRISECOUR )
               |)!! HPAROLE
          | (| LREPRISECOUR:= LREPRISE $1
               | FINRETOURARRIERE:= true when( ( LREPRISECOUR=1 )
                                    and( not SENSCOUR ))
               |)!! FINRETOURARRIERE
          |)
     where
          integer LREPRISECOUR init-1;
          logical SENS init true, ONVASEMETTREF, ONVASEMETTREB,
                  SWITCH, ZSWITCH init true, REPRISECOUR init
                  false
     end;
     ALIMENTATION_TESTS( ? logical STOPHK, STOPV;
                           integer VOIS1;
                           logical HRAPIDE, SENSCOUR, ONSEMETF,
```

```
                        ONSEMETB, REDEMARRAGE;
                      real YCOUR
                    ! real Y_HK;
                      logical RAZ_HK;
                      real BIAIS, SEUIL, Y_V;
                      logical RAZ_V )
    (| (| DIFFERER_LOG ? CADENCE:HRAPIDE, C:STOPHK
                       ! CC:HKOUT
     | INTERRUPT ? Y:YCOUR, CIN:REDEMARRAGE, COUT:HKOUT
                 ! YY:Y_HK
     | RAZ_HK:= REDEMARRAGE when( event Y_HK )
     | VOISE:= VOIS1=2
     | DIFFERER_LOG ? CADENCE:HRAPIDE, C:VOISE
                    ! CC:SEGVOISE
     | BIAIS:= ( ( -2 e-1 )when ONSEMETB )default( ( -2
               e-1 )when SEGVOISE )default( ( -8 e-1 )
               when( not SEGVOISE ))
     | SEUIL:= ( 4 e 1 when ONSEMETB )default( 4 e 1
               when SEGVOISE )default( 8 e 1 when( not
               SEGVOISE ))
     |)!! Y_HK, RAZ_HK, BIAIS, SEUIL
    | (| DIFFERER_LOG ? CADENCE:HRAPIDE, C:STOPV
                     ! CC:VOUT
     | INTERRUPT ? Y:YCOUR, CIN:ONSEMETF, COUT:VOUT
                 ! YY:YCOURV
     | Y_V:= YCOURV when SENSCOUR
     | RAZ_V:= ONSEMETF when( event Y_V )
     |)!! Y_V, RAZ_V
    |)
where
    real YCOURV;
    logical HKOUT, VOUT, VOISE, SEGVOISE
end;
GESTION_INDICES| integer KMIN, LIMITEBAK, N
                 ? logical HPAROLE, FINRETOURARRIERE;
                   integer LSEGMENTBI, QUI, LSEGMENTF, VOIS1
                 ! integer IDEB, IFIN, NUM, VOIS )
    (| (| OKVOISE:= VOIS1=2
       | HSEGMENTF:= event LSEGMENTF
       | RESYNCEVENT3 ? YIN:OKVOISE, SYNC:HSEGMENTF
                      ! YOUT:VOISE
       | VOISEPREC:= VOISE $1
       | LMIN:= ( ( ( 4*KMIN )when( ( QUI=1 )or VOISEPREC
                ))default( 2*KMIN ))when( event LSEGMENTF
                )
       | OKB:= ( LSEGMENTF>=LMIN )and VOISE
       |)!! VOISEPREC, OKB
         ! OKB:OKB_2, VOISEPREC:VOISEPREC_1
     | (| LSEGMENTVALIDE:= ( LSEGMENTF when( not OKB ))
                           default LSEGMENTB
       | ZNUM:= NUM $1
       | NUM:= ZNUM+LSEGMENTVALIDE
       | HNUM:= event NUM
       | RESYNCEVENT2 ? YIN:VOIS1, SYNC:HNUM
                      ! YOUT:VOIS
       |)!! NUM, VOIS
         ? LSEGMENTB:LSEGMENTB_3, OKB:OKB_2
       | (| (| HLSEGMENTF:= event LSEGMENTF
            | COMPTEUR_DIFF ? CADENCE:HPAROLE, L:HLSEGMENTF
                            ! DIFFN:NAJOUTES
            | RESYNCEVENT2 ? YIN:ZIDEB, SYNC:HLSEGMENTF
                           ! YOUT:CIDEB
            | IREP:= ( ( CIDEB-NAJOUTES )+LSEGMENTF )when(
                     not OKB )
            | JREP:= N when( event IREP )
            | IB:= ( ( ( CIDEB-NAJOUTES )+LSEGMENTF )-1 )
                   when OKB
            | JB:= ( ( CIDEB-NAJOUTES )+LIMITEBAK )when OKB
            |)!! IREP, JREP, IB, JB, OKB
          | (| DMIN:= ( ( ( 2*KMIN )when( QUI=1 ))default(
                      KMIN when VOISEPREC )default( KMIN/2 ))
                      when( event LSEGMENTF )
            | HLSEGMENTBI:= event LSEGMENTBI
            | RESYNCEVENT2 ? YIN:DMIN, SYNC:HLSEGMENTBI
                           ! YOUT:DISTMIN
            | RESYNCEVENT2 ? YIN:ZIDEB, SYNC:HLSEGMENTBI
                           ! YOUT:CCIDEB
            | RESYNCEVENT2 ? YIN:ZIFIN, SYNC:HLSEGMENTBI
                           ! YOUT:CCIFIN
            | ACCEPTE:= ( CCIDEB-LSEGMENTBI )>=( ( CCIFIN-
                        LIMITEBAK )+DISTMIN )
            | IPDTB:= ( ( CCIDEB-LSEGMENTBI )+1 )when
                      ACCEPTE
            | JPDTB:= CCIFIN when( event IPDTB )
            |)!! IPDTB, JPDTB, ACCEPTE
          | (| FINRA:= ( event FINRETOURARRIERE )default(
                       true when( not ACCEPTE ))
            | HFINRA:= event FINRA
            | RESYNCEVENT2 ? YIN:ZIDEB, SYNC:HFINRA
                           ! YOUT:CCCIDEB
            | RESYNCEVENT2 ? YIN:ZIFIN, SYNC:HFINRA
                           ! YOUT:CCCIFIN
            | IREPAB:= CCCIDEB+1
            | JREPAB:= N when( event IREPAB )
            | LSEGMENTB:= ( CCCIDEB+1 )-( CCCIFIN-LIMITEBAK
                          )
            |)!! IREPAB, JREPAB, LSEGMENTB
          | IDEB:= IREP default IB default IREPAB default
                   IPDTB default( 0 when( event ACCEPTE ))
          | ZIDEB:= IDEB $1
          | IFIN:= JREP default JB default JREPAB default
                   JPDTB default( 0 when( event ACCEPTE ))
          | ZIFIN:= IFIN $1
          |)!! IDEB, IFIN, LSEGMENTB
            ? OKB:OKB_2, VOISEPREC:VOISEPREC_1
            ! LSEGMENTB:LSEGMENTB_3
       |)/ LSEGMENTB_3, OKB_2, VOISEPREC_1
    where
        integer LSEGMENTB_3, LMIN, NAJOUTES, IREP, JREP, IB, JB
                , IREPAB, JREPAB, IPDTB, JPDTB, ZIDEB init N+1,
                ZIFIN, CIDEB init N+1, CCIDEB, CCIFIN, CCCIDEB,
                CCCIFIN, DMIN, DISTMIN, LSEGMENTB,
                LSEGMENTVALIDE init-1, ZNUM init-1;
        logical OKB_2, VOISEPREC_1, VOISE, VOISEPREC init false
                , OKB, HLSEGMENTF, ACCEPTE, FINRA, OKVOISE,
                HSEGMENTF, HNUM, HLSEGMENTBI, HFINRA
    end
end;
TEST_HINKLEY| integer KMIN
              ? real Y;
                logical RAZ;
                real BIAIS, SEUIL
              ! integer F_EVENTUELLE;
                logical DETECTION )
    (| MODELE( 16, KMIN )? SRAZ:RAZ
                          ! ERRLONG:ERRLONG_4, VLONG:VLONG_3,
                            ERRCOURT:ERRCOURT_2, VCOURT:VCOURT_1
     | (| RESYNCEVENT1 ? YIN:BIAIS, SYNC:HERRLONG
```

```
                    : YOUT:BIAI
        | HERRLONG:= event ERRLONG
        |)/ HERRLONG
          ? ERRLONG:ERRLONG_4
          ! BIAI:BIAI_5
      | STAT_HINKLEY ? SRAZ:SRAZ_6, BIAI:BIAI_5,
                       ERRLONG:ERRLONG_4, VLONG:VLONG_3,
                       ERRCOURT:ERRCOURT_2, VCOURT:VCOURT_1
                     ! U:U_7
      | MAX ? U:U_7, SRAZ:SRAZ_6
          ! NOUVMAX:NOUVMAX_9, MAX:MAX_8
      | (| (| HU:= event U
           | RESYNCEVENT1 ? YIN:SEUIL, SYNC:HU
                          ! YOUT:SEUI
           | DETECTION:= true when( ( MAX-U )>=SEUI )
           |)!! DETECTION
        | (| POS_MAX:= ( 0 when RAZ )default( NY when( event
                        NOUVMAX ))
           | F_EVENTUELLE:= POS_MAX cell( event NY )
           |)!! F_EVENTUELLE
        |)? NY:NY_10, NOUVMAX:NOUVMAX_9, MAX:MAX_8, U:U_7
      | (| HY:= event Y
        | COMPTEUR_AFTER_RAZ ? CADENCE:HY
                             ! N:NY
        | RAZTEST:= NY=KMIN
        |)/ HY
          ! NY:NY_10, RAZTEST:SRAZ_6
      |)/ NY_10, NOUVMAX_9, MAX_8, U_7, SRAZ_6, BIAI_5, ERRLONG_4
        , VLONG_3, ERRCOURT_2, VCOURT_1
where
     real MAX, MAX_8, U, U_7, BIAI, BIAI_5, ERRLONG, ERRLONG_4,
        VLONG, VLONG_3, ERRCOURT, ERRCOURT_2, VCOURT, VCOURT_1,
        SEUI;
     integer NY, NY_10, POS_MAX;
     logical NOUVMAX, NOUVMAX_9, RAZTEST, SRAZ_6, HY, HU,
          HERRLONG
end;
TEST_VOISEMENT{ integer KMIN
                ? logical RAZ;
                  real Y
                ! integer VOIS;
                  logical DETECTION;
                  integer F_EVENTUELLE }
=    (| CALCUL_VOISEMENT( 2.1 e 5/1.28 e 2, 8.5 e 6/1.28 e 2, 1.9
                          e 5/1.28 e 2, 5.5 e 6/1.28 e 2, 8.e 1/2
                          .56 e 2, KMIN/2 )? WY:WY_1
                               ! VOISEMENT:VOISEMENT_2
     | CONTROLE( KMIN/2, KMIN/4 )! NY:NY_6,
                                   INFPREMIERTEST:
                                   INFPREMIERTEST_5,
                                   NBTESTS:NBTESTS_4,
                                   PREMIERTEST:PREMIERTEST_3,
                                   WWY:WY_1
     | (| (| (| ZNVOISC:= NVOISC $1
              | ZZNVOISC:= ( ( 0 when PREMIERTEST )default
                             ZNVOISC )when( event VOIS )
              |)!! ZZNVOISC
           | (| ZDERNIERV:= DERNIERV $1
              | ZZDERNIERV:= ( ( VOIS when PREMIERTEST )default
                               ZDERNIERV )when( event VOIS )
              |)!! ZZDERNIERV
           | (| ZAVDERNIERV:= AVDERNIERV $1
              | ZZAVDERNIERV:= ( ( VOIS when PREMIERTEST )
                                 default ZAVDERNIERV )when( event
                                 VOIS )
              |)!! ZZAVDERNIERV
           | C1:= ZZDERNIERV=VOIS
           | C2:= ZZAVDERNIERV=ZZDERNIERV
           | C3:= ZZAVDERNIERV=VOIS
           | C4:= ( NVOISC>4 )and( NVOISC/=NBTESTS )
           | NVOISC:= ( ( ( ZZNVOISC+1 )when C1 )default(
                      NBTESTS when C3 )default 1 )when( event
                      VOIS )
           | DERNIERV:= VOIS
           | AVDERNIERV:= ( ( ZZAVDERNIERV when( C1 or C2 or C3
                          ))default ZZDERNIERV )when( event VOIS
                          )
           | NRU:= ( ( 0 when( C1 and( ZZNVOISC=( NBTESTS-1 ))))
                  default( ZZNVOISC when( C1 and( ZZNVOISC=(
                  NBTESTS-2 ))))default( ( ZZNVOISC+1 )when( C1
                  and( ZZNVOISC/=( NBTESTS-1 ))))default( 1
                  when C2 )default( 0 when C3 )default(
                  ZZNVOISC+1 ))when( event VOIS )
           | DETECTION:= true when( ( C1 and C4 )or( ( not C1 )
                         and( not C2 )and( not C3 )))
           | LSEGACTIF:= ( ( NBTESTS+1 )-NRU )*( KMIN/4 )
           |)!! DETECTION, LSEGACTIF
             ? NBTESTS:NBTESTS_4, PREMIERTEST:PREMIERTEST_3,
               VOIS:VOISEMENT_2
             ! LSEGACTIF:LSEGACTIF_7
        | (VOIS1:= VOIS when PREMIERTEST)?
                                 PREMIERTEST:PREMIERTEST_3,
                                 VOIS:VOISEMENT_2
                                 ! VOIS1:VOIS
        | (| LSEGACTIFNY:= LSEGACTIF cell( event NY )
          | F_EVENTUELLE:= ( NY when INFPREMIERTEST )default
                           LSEGACTIFNY
          |)/ LSEGACTIFNY
            ? LSEGACTIF:LSEGACTIF_7, NY:NY_6,
              INFPREMIERTEST:INFPREMIERTEST_5
        |)/ LSEGACTIF_7
     |)/ NY_6, INFPREMIERTEST_5, NBTESTS_4, PREMIERTEST_3,
        VOISEMENT_2, WY_1
where
     [ KMIN/2 ]real WY_1;
     integer LSEGACTIF, LSEGACTIF_7, NY_6, NBTESTS_4, VOISEMENT,
          VOISEMENT_2, VOIS1, VOIS, LSEGACTIFNY, NVOISC,
          ZNVOISC, ZZNVOISC, NRU, DERNIERV, ZDERNIERV,
          ZZDERNIERV, AVDERNIERV, ZAVDERNIERV, ZZAVDERNIERV;
     logical INFPREMIERTEST_5, PREMIERTEST_3, C1, C2, C3, C4
process
     CONTROLE{ integer IECH, IIECH
               ? logical RAZ;
                 real Y
               ! logical PREMIERTEST;
                 [ IECH ]real WWY;
                 integer NBTESTS;
                 logical INFPREMIERTEST;
                 integer NY }
     =    (| (| HY:= event Y
             | COMPTEUR_MODULO_AFTER_RAZ( IIECH )? CADENCE:HY
                                  ! NMOD:NYMOD
             | COMPTEUR_AFTER_RAZ ? CADENCE:HY
                                  ! N:NY
             |)/ HY
          | (| ACTIF:= ( NY/=IIECH )and( NYMOD=IIECH )
             | WY:= Y window IECH
             | WWY:= WY when ACTIF
```

```
            |)/ ACTIF, WY
          | INFPREMIERTEST:= NY<IECH
          | (| HWWY:= event WWY
            | PREMIERTEST:= ( NY=IECH )when HWWY
            | COMPTEUR_AFTER_RAZ ? CADENCE:HWWY,
                                   RAZ:PREMIERTEST
                                 ! N:NBTESTS
            |)/ HWWY
          |)/ NYMOD
    where
         [ IECH ]real WY;
         integer NYMOD;
         logical HY, ACTIF, HWWY
    end
end;
GESTION_INFO[ integer KMIN
            ? integer NUM, F_HKF, NUMF;
              logical HRAPIDE;
              logical REDEMARRAGE;
              integer LSEG_V;
              logical DET_V;
              integer LSEG_HK;
              logical DET_HK, SWITCH_FB
            ! integer LSEGMENTF, QUI, LSEGMENTB ]
=   (| COMMUT_FB ! DET_HK_BAK:DET_HK_BAK_4,
                   LSEG_HK_BAK:LSEG_HK_BAK_3,
                   DET_HK_FOR:DET_HK_FOR_2,
                   LSEG_HK_FOR:LSEG_HK_FOR_1
     | GESTION_INFO_BACKWARD ? DET_HK_BAK:DET_HK_BAK_4,
                               LSEG_HK_BAK:LSEG_HK_BAK_3
     | GESTION_INFO_FORWARD ? DET_HKF:DET_HKF_6,
                              LSEG_HKF:LSEG_HKF_5,
                              DET_HK:DET_HK_FOR_2,
                              LSEG_HK:LSEG_HK_FOR_1
     | INTERFACE_FILTRE( 5,( 2*KMIN )+2 )? NUMF1:NUMF,
                                   F_HKF1:F_HKF
                                   ! DET_HKF:DET_HKF_6,
                                   LSEG_HKF:LSEG_HKF_5
     |)/ DET_HKF_6, LSEG_HKF_5, DET_HK_BAK_4, LSEG_HK_BAK_3,
        DET_HK_FOR_2, LSEG_HK_FOR_1
where
     logical DET_HKF_6, DET_HK_BAK_4, DET_HK_FOR_2;
     integer LSEG_HKF_5, LSEG_HK_BAK_3, LSEG_HK_FOR_1
process
     INTERFACE_FILTRE[ integer NF, BMAX
                     ? integer NUM, F_HKF1, NUMF1;
                       logical HRAPIDE
                     ! integer LSEG_HKF;
                       logical DET_HKF ]
     =    (| DIFFERER_INT ? CADENCE:HRAPIDE, C:NUM
                          ! CC:DNUM
          | HREDEMARRAGE:= event DNUM
          | DIFFERER_INT ? CADENCE:HRAPIDE, C:NUMF1
                          ! CC:NUMF
          | DIFFERER_INT ? CADENCE:HRAPIDE, C:F_HKF1
                          ! CC:F_HKF
          | (| WNUMF:= NUMF window NF
             | RESYNCEVENT5( NF )? WIN:WNUMF, SYNC:HREDEMARRAGE
                                 ! WOUT:WWNUMF
             | ZERO:= 0 when( event HREDEMARRAGE )
             | synchro HREDEMARRAGE, VAL
             |
               array I to NF
                     of VAL:= ( WWNUMF when( WWNUMF>( DNUM+BMAX+
                                                      1 ))default VAL
                   with VAL[.]:ZERO, WWNUMF
                   end
             | FRONTIEREF:= VAL[ 1 ]
             | PRESENTF:= VAL[ 1 ]/=0
             |)!! FRONTIEREF, PRESENTF
          | (| MMINNUMF:= DNUM+BMAX+1
             | HNUMF:= event NUMF
             | RESYNCEVENT2 ? YIN:MMINNUMF, SYNC:HNUMF
                            ! YOUT:MINNUMF
             | DET_HKF:= PRESENTF default( not( event
                         HREDEMARRAGE ))default( NUMF>MINNUMF )
             |)!! DET_HKF
          | (| HF_HKF:= event F_HKF
             | RESYNCEVENT3 ? YIN:DET_HKF, SYNC:HF_HKF
                            ! YOUT:DDET_HKF
             | RESYNCEVENT2 ? YIN:DNUM, SYNC:HF_HKF
                            ! YOUT:DDNUM
             | LSEG_HKF:= ( ( FRONTIEREF-DNUM )when PRESENTF )
                          default( 0 when( not PRESENTF ))
                          default( ( F_HKF-DDNUM )when( not
                          DDET_HKF ))
             |)!! LSEG_HKF
          |)!! DET_HKF, LSEG_HKF
     where
          [ NF ]integer WNUMF init[ ( to NF ): 0 ], WWNUMF init[
                                    ( to NF ]: 0 ], VAL;
          integer NUMF, F_HKF, DNUM, ZERO, FRONTIEREF, MINNUMF,
                  MMINNUMF, DDNUM init 0;
          logical HCALCUL, PRESENTF, DDET_HKF init false,
                  HREDEMARRAGE, HNUMF, HF_HKF
     end;
     COMMUT_FB[ ? logical SWITCH_FB;
                  integer LSEG_HK;
                  logical DET_HK
                ! integer LSEG_HK_FOR;
                  logical DET_HK_FOR;
                  integer LSEG_HK_BAK;
                  logical DET_HK_BAK ]
     =    (| (| FLIPFLOP ? SWITCH:SWITCH_FB
                         ! B:BASCULE, ZB:ZBASCULE
                         / ZBASCULE
             | HLSEG_HK:= event LSEG_HK
             | RESYNCEVENT3 ? YIN:BASCULE, SYNC:HLSEG_HK
                            ! YOUT:MAINFOR
             |)!! MAINFOR
          | LSEG_HK_FOR:= LSEG_HK when MAINFOR
          | DET_HK_FOR:= DET_HK when MAINFOR
          | LSEG_HK_BAK:= LSEG_HK when( not MAINFOR )
          | DET_HK_BAK:= DET_HK when( not MAINFOR )
          |)/ MAINFOR
     where
          logical BASCULE, ZBASCULE init true, MAINFOR init true,
                  HLSEG_HK
     end;
     GESTION_INFO_FORWARD[ ? integer LSEG_HKF;
                             logical DET_HKF;
                             integer LSEG_V;
                             logical DET_V, REDEMARRAGE;
                             integer LSEG_HK;
                             logical DET_HK
                           ! integer LSEGMENTF, QUI ]
     =    (| HFORWARD:= ( event LSEG_HK )default( event LSEG_V )
                        default( event LSEG_HKF )
```

```
                    | RAZ:= REDEMARRAGE when( event HFORWARD )
                    | LSEGHK:= LSEG_HK cell( event HFORWARD )
                    | LSEGV:= LSEG_V cell( event HFORWARD )
                    | LSEGHKF:= LSEG_HKF cell( event HFORWARD )
                    | PRESENTHK:= ( ( not( event RAZ ))default DET_HK )
                                  cell( event HFORWARD )
                    | PRESENTV:= ( ( not( event RAZ ))default DET_V )cell(
                                 event HFORWARD )
                    | PRESENTHKF:= ( DET_HKF cell( event HFORWARD ))when(
                                   event HFORWARD )
                    | HKINF:= LSEGHK<=LSEGHKF
                    | VINF:= LSEGV<=LSEGHKF
                    | HKFINF:= ( ( LSEGHKF<=LSEGHK )and PRESENTHK )or( (
                              LSEGHKF<=LSEGV )and PRESENTV )
                    | LSEGMENTF:= ( LSEGHK when( HKINF and PRESENTHK ))
                                  default( LSEGV when( VINF and PRESENTV )
                                  )default( LSEGHKF when( HKFINF and
                                  PRESENTHKF ))
                    | QUI:= ( 0 when( HKINF and PRESENTHK ))default( 1
                           when( VINF and PRESENTV ))default( 2 when(
                           HKFINF and PRESENTHKF ))
                    |)!! LSEGMENTF, QUI
               where
                    integer LSEGHK, LSEGV, LSEGHKF;
                    logical HFORWARD, PRESENTHK init false, PRESENTV init
                          false, PRESENTHKF init false, HKINF, VINF,
                          HKFINF, RAZ
               end;
               GESTION_INFO_BACKWARD{ ? integer LSEG_HK_BAK;
                                        logical DET_HK_BAK
                                      ! integer LSEGMENTB }
               =    LSEGMENTB:= LSEG_HK_BAK when DET_HK_BAK
               end
          end
     end
end
```

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 518 **MULTISCALE SYSTEM THEORY**
Albert BENVENISTE, Ramine NIKOUKHAH, Alan S. WILLSKY
Février 1990, 30 Pages.

PI 519 **PANDORE : A SYSTEM TO MANAGE DATA DISTRIBUTION**
Françoise ANDRE, Jean-Louis PAZAT, Henry THOMAS
Février 1990, 14 Pages.

PI 520 **SCHEDULING AFFINE PARAMETERIZED RECURRENCES BY MEANS OF VARIABLE DEPENDENT TIMING FUNCTIONS**
Christophe MAURAS, Patrice QUINTON,
Sanjay RAJOPADHYE, Yannick SAOUTER
Février 1990, 14 Pages.

PI 521 **COMPUTABILITY OF RECURRENCE EQUATIONS**
Yannick SAOUTER, Patrice QUINTON
Février 1990, 28 Pages.

PI 522 **PROGRAMMING BY MULTISET TRANSFORMATION**
Jean-Pierre BANATRE, Daniel LE METAYER
Mars 1990, 26 Pages.

PI 523 **GOTHIC MEMORY MANAGEMENT : A MULTIPROCESSOR SHARED SINGLE LEVEL STORE**
Béatrice MICHEL
Mars 1990, 20 Pages.

PI 524 **ORDER NOTIONS AND ATOMIC MULTICAST IN DISTRIBUTED SYSTEMS : A SHORT SURVEY**
Michel RAYNAL
Mars 1990, 18 Pages.

PI 525 **MULTI-SCALE AUTOREGRESSIVE PROCESSES**
Michèle BASSEVILLE, Albert BENVENISTE
Mars 1990, 136 Pages.

PI 526 **TRANSFORMATIONS PYRAMIDALES D'IMAGES NUMERIQUES**
Nadia BAAZIZ, Claude LABIT
Mars 1990, 44 Pages.

PI 527 **LE LANGAGE SIGNAL : UN EXEMPLE EN SEGMENTATION AUTOMATIQUE DE LA PAROLE CONTINUE**
Claude LE MAIRE
Mars 1990, 112 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399