UTILISATION DE CÆSAR ET D'ALDEBERAN[1]

Hubert Garavel

31 août 1993

[1](annexes C et D de la thèse de l'auteur)

Table des matières

Annexe C

Application 1 : protocole du bit alterné

Je suis bi-alternatif
Alternativement positif
Je suis bi-bi-bi-bi-bi-bi-alternatif
Alternativement positif
Je suis alternativement bi-dégénéré, ça c'est super
Alternativement hyper-positif

Gogol I^{er}
poète, prophète, barbare

C.1 Description du service

Le protocole du bit alterné (*alternating bit protocol*) fait partie de la couche transport (4ème couche du modèle OSI). Il permet le transfert de données entre une paire d'entités pour lesquelles une connexion bi-directionnelle a été préalablement établie.

Pour simplifier le problème, on crée une disymétrie entre les deux entités : la première (T, comme *transmitter*) émet des messages à destination de la seconde (R, comme *receiver*).

Les messages sont modélisés par des numéros compris entre 1 et un entier maximal `N` ; ils sont spécifiés par le type abstrait suivant, qui ne précise pas leur nature exacte :

```
type MESSAGE is
   sorts MSG   (* type MSG = 1..N *)
endtype
```

Vu de la couche supérieure, le service fourni par le protocole du bit alterné est l'acheminement d'une série de messages de T vers R. La transmission est fiable : les messages ne peuvent pas être perdus ni dupliqués et ils sont reçus dans l'ordre où ils ont été émis. La spécification Lotos suivante décrit ce comportement :

```
specification ALTERNATING_BIT_SERVICE [PUT, GET] : noexit behaviour
   SERVICE [PUT, GET]
where
   process SERVICE [PUT, GET] : noexit :=
      PUT ?M:MSG;  (* acquisition d'un message *)
         GET !M;   (* livraison du message *)
            SERVICE [PUT, GET]
   endproc
endspec
```

C.2 Description du protocole

Le fonctionnement "idéal" du protocole du bit alterné est le suivant : T envoie un message à R ; à la réception de ce message, R renvoie un acquittement à T.

La liaison entre T et R n'est pas fiable : il est possible que des messages ou des acquittements soient perdus. En cas de perte, le medium peut, de manière facultative, signaler cette perte au destinataire (T ou R) en envoyant une indication de perte.

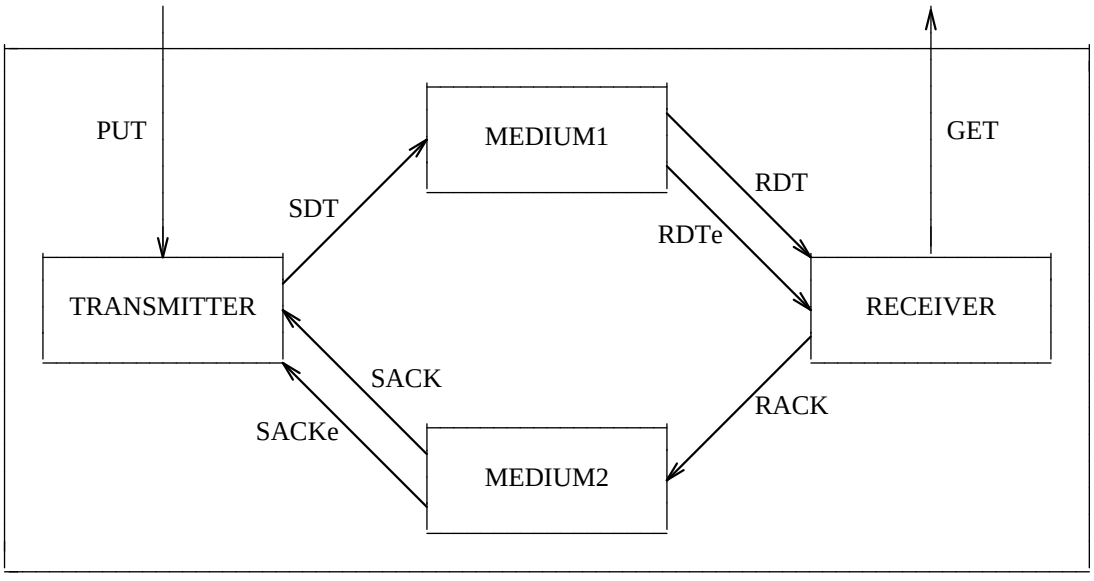
Pour détecter les pertes non signalées, les messages et les acquittements contiennent un bit de contrôle. Le bit de contrôle de chaque acquittement est égal au bit de contrôle du message qu'il acquitte. Les bits de contrôle de deux messages successivement émis ont des valeurs distinctes (la valeur du bit alterne à chaque émission).

Si l'entité T reçoit une indication de perte d'acquittement ou un acquittement avec un bit de contrôle erroné, elle réémet le dernier message envoyé.

Si l'entité R reçoit une indication de perte de message ou un message avec un bit de contrôle erroné, elle réémet le dernier acquittement envoyé.

C.3 Architecture du protocole

On choisit de décrire le protocole par quatre processus parallèles communicants :



processus	*signification*
TRANSMITTER	entité émettrice T
RECEIVER	entité réceptrice R
MEDIUM1	transmission des messages de T vers R
MEDIUM2	transmission des acquittements de R vers T

Le tableau suivant donne la liste des signaux utilisés. Les seuls signaux fournis par le service sont PUT et GET ; tous les autres désignent des signaux internes. Dans la suite M désigne un message (essentiellement un bloc de données) et B le bit de contrôle d'un message.

signal	*origine*	*destination*	*signification*
PUT !M	service	TRANSMITTER	émission d'un message
SDT !M !B	TRANSMITTER	MEDIUM1	envoi du message
RDT !M !B	MEDIUM1	RECEIVER	transmission du message
RDTe	MEDIUM1	RECEIVER	perte du message
GET !M	RECEIVER	service	réception du message
RACK !B	RECEIVER	MEDIUM2	renvoi d'un acquittement
SACK !B	MEDIUM2	TRANSMITTER	transmission de l'acquittement
SACKe	MEDIUM2	TRANSMITTER	perte de l'acquittement

L'émetteur et le récepteur fonctionnent de manière complètement asynchrone. Il en est de même pour les deux media. On peut donc spécifier l'architecture du protocole par le programme LOTOS ci-dessous, dans lequel les définitions des processus TRANSMITTER, RECEIVER, MEDIUM1 et MEDIUM2 ne sont pas explicitées :

```
specification ALTERNATING_BIT_PROTOCOL [PUT, GET] : noexit behaviour
   hide SDT, RDT, RDTe, RACK, SACK, SACKe in
      (
         (
         TRANSMITTER [PUT, SDT, SACK, SACKe] (0)
         |||
         RECEIVER [GET, RDT, RDTe, RACK] (0)
         )
      |[SDT, RDT, RDTe, RACK, SACK, SACKe]|
         (
         MEDIUM1 [SDT, RDT, RDTe]
         |||
         MEDIUM2 [RACK, SACK, SACKe]
         )
      )
where
   type BIT is
      sorts BIT
      opns 0 : -> BIT
           1 : -> BIT
           not : BIT -> BIT
   endtype
endspec
```

Remarque C-1

Le paramètre effectif 0 des processus TRANSMITTER et RECEIVER sert à initialiser la valeur du bit de contrôle ; par convention le bit de contrôle du premier message est égal à 0. ■

C.4 Spécification du medium des messages

En recevant un message M avec un bit de contrôle égal à B, le medium n^{o} 1 peut réagir de trois façons différentes :

- transmettre correctement le message et son bit de contrôle. En aucun cas le medium ne peut changer la valeur du bit de contrôle
- perdre le message et envoyer une indication de perte à l'entité réceptrice
- perdre silencieusement le message

```
process MEDIUM1 [SDT, RDT, RDTe] : noexit :=
   SDT ?M:MSG ?B:BIT;   (* reception d'un message *)
      (
      RDT !M !B;        (* transmission correcte *)
         MEDIUM1 [SDT, RDT, RDTe]
      []
      RDTe;             (* perte avec indication *)
         MEDIUM1 [SDT, RDT, RDTe]
      []
      i;                (* perte silencieuse *)
         MEDIUM1 [SDT, RDT, RDTe]
      )
endproc
```

C.5 Spécification du medium des acquittements

Le fonctionnement du medium n^{o} 2 est analogue à celui du medium n^{o} 1. La seule différence réside dans les noms de signaux et dans le fait que les acquittements, contrairement aux messages, ne portent pas d'information autre que le bit de contrôle.

```
process MEDIUM2 [RACK, SACK, SACKe] : noexit :=
   RACK ?B:BIT;   (* reception d'un acquittement *)
      (
      SACK !B;    (* transmission correcte *)
         MEDIUM2 [RACK, SACK, SACKe]
      []
      SACKe;      (* perte avec indication *)
         MEDIUM2 [RACK, SACK, SACKe]
      []
      i;          (* perte silencieuse *)
         MEDIUM2 [RACK, SACK, SACKe]
      )
endproc
```

C.6 Spécification de l'émetteur

L'entité émettrice acquiert un message via `PUT` et le transmet au medium n^{o} 1 après lui avoir ajouté la valeur courante B du bit de contrôle. Si elle reçoit en réponse un acquittement avec un bit de contrôle B la transmission a réussi, sinon il faut réémettre le message. Il y a 3 causes possibles de réémission :

- l'émetteur a reçu un acquittement ayant $(\neg B)$ comme bit de contrôle

- l'émetteur a reçu une indication de perte d'acquittement `SACKe`
- l'émetteur peut réémettre spontanément le message afin d'éviter le blocage dans le cas où le medium n° 1 (*resp.* n° 2) aurait perdu silencieusement un message (*resp.* un acquittement). En réalité, cette réémission n'a lieu que si une certaine contrainte de délai (*timeout*) est vérifiée (*cf.* [RSV86] et [RV87]) mais, LOTOS ne permettant pas d'exprimer le délai, on le modélise par un événement silencieux "**i**"

Remarque C-2
Pour décrire l'entité émettrice on a choisi une structure de contrôle paramétrée par des données (essentiellement la valeur B du bit de contrôle). D'autres solutions auraient été également viables, en particulier une modélisation sans données, avec une structure de contrôle complètement développée. Il s'agit de la dualité classique entre le contrôle et les données. ■

```
process TRANSMITTER [PUT, SDT, SACK, SACKe] (B:BIT) : noexit :=
   PUT ?M:MSG;  (* acquisition d'un message *)
      TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
where
   process TRANSMIT [PUT, SDT, SACK, SACKe] (B:BIT, M:MSG) : noexit :=
      SDT !M !B;           (* emission du message *)
         (
         SACK !B;          (* bit de controle correct *)
            TRANSMITTER [PUT, SDT, SACK, SACKe] (not (B))
         []
         SACK !(not (B));  (* bit de controle incorrect => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         []
         SACKe;            (* indication de perte => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         []
         i;                (* timeout => reemission *)
            TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
         )
   endproc
endproc
```

C.7 Spécification du récepteur

Remarque C-3
Le comportement de l'entité réceptrice est parfaitement symétrique de celui de l'entité paire, bien que la modélisation "contrôle+données" choisie puisse faire croire à la disymétrie. ■

Lorsqu'elle reçoit un message avec un bit de contrôle B correct, l'entité réceptrice délivre le message via `GET` et renvoie un acquittement avec un bit de contrôle égal à B. Dans les autres cas, elle renvoie un acquittement incorrect (ayant $(\neg B)$ comme bit de contrôle) ; ces cas sont au nombre de trois :

- le récepteur a reçu un message ayant $(\neg B)$ comme bit de contrôle
- le récepteur a reçu une indication de perte de message `RDTe`
- le récepteur peut émettre spontanément un acquittement invalide afin d'éviter le blocage dans le cas où le medium n° 1 (*resp.* n° 2) aurait perdu silencieusement un message (*resp.* un acquittement). Comme expliqué plus haut, il s'agit d'un *timeout* que l'on modélise par un événement silencieux "**i**"

```
process RECEIVER [GET, RDT, RDTe, RACK] (B:BIT) : noexit :=
   RDT ?M:MSG !B;          (* bit de controle correct *)
      GET !M;              (* livraison du message *)
         RACK !B;          (* envoi d'un acquittement correct *)
            RECEIVER [GET, RDT, RDTe, RACK] (not (B))
   []
   RDT ?M:MSG !(not (B));  (* bit de controle incorrect => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
   []
   RDTe;                   (* indication de perte => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
   []
   i;                      (* timeout => *)
      RACK !(not (B));     (* envoi d'un acquittement incorrect *)
         RECEIVER [GET, RDT, RDTe, RACK] (B)
endproc
```

C.8 Validation

A l'aide de la version 3.1 de CÆSAR on a produit, pour diverses valeurs de N, les réseaux et les graphes qui correspondent aux descriptions en LOTOS du service et du protocole.

Le réseau du service comporte 2 places, 2 transitions et 1 variable ; celui du protocole comporte 17 places, 21 transitions et 7 variables. Ces valeurs sont indépendantes de N.

Les deux tableaux suivants indiquent les résultats obtenus respectivement pour le service et le protocole. Dans chaque cas on indique la taille du graphe (nombre d'états et nombre d'arcs) ainsi que le temps total mis par CÆSAR pour produire ce graphe ; cette durée est exprimée sous la forme d'un couple *max–min* où :

- *max* est mesuré sur une station de travail SUN 3/50 avec 4Mo de mémoire principale, pour un seul utilisateur qui a ouvert 3 fenêtres sous X-windows version 10. On relève ainsi les performances de CÆSAR dans un environnement "normal"
- *min* est mesuré sur une station de travail SUN 3/60 avec 8Mo de mémoire principale, pour un seul utilisateur qui exécute seulement CÆSAR

Les durées *max* et *min* sont données sous la forme *minutes*:*secondes* ; il s'agit de temps utilisateur et non de temps virtuel. La dernière colonne du table contient les *débits*, c'est-à-dire le nombre d'états produits par seconde, calculés pour *max* et *min*.

N	états	arcs	temps	débit
5	11	35	0:27–0:22	0.4–0.5
10	21	120	0:28–0:18	0.8–1.2
15	31	255	0:30–0:17	1.0–1.8
20	41	440	0:30–0:17	1.4–2.4
25	51	675	0:34–0:19	1.5–2.7
30	61	960	0:32–0:17	1.9–3.5
35	71	1295	0:38–0:17	1.9–4.2
40	81	1680	0:29–0:15	2.8–5.4
45	91	2115	0:30–0:16	3.0–6
50	101	2600	0:31–0:19	3.3–5.3
70	141	5040	0:33–0:22	4.2–6.4

N	états	arcs	temps	débit
5	3 576	11 317	1:09–0:43	52–83
10	12 346	39 527	1:33–1:37	133–127
15	26 316	84 637	2:28–2:02	177–130
20	45 486	146 647	4:01–2:56	189–258
25	69 856	225 557	5:35–4:05	209–285
30	99 426	321 367	8:13–4:51	201–342
35	134 196	434 077	11:08–6:32	200–342
40	174 166	563 687	23:39–9:11	122–316
45	219 336	710 197	159:49–11:11	23–326
50	269 706	873 607	?–14:48	?–304
70	523 186	1 696 247	?–80:53	?–108

Pour `N` valant 5, 10 et 15, l'utilisation du logiciel ALDEBARAN [Fer88] a permis de montrer que le graphe du protocole était observationnellement équivalent à celui du service.

Pour des valeurs élevées de `N` la taille du graphe croît rapidement et avec elle le temps nécessaire pour comparer le protocole au service. Dans de telles situations, le recours aux logiques temporelles s'impose. Voici un exemple de propriétés, exprimées dans la logique R$\overset{\star}{\text{I}}$CO, qui devraient être vérifiées par le graphe correspondant au protocole du bit alterné :

- il n'y a pas de blocage :

  not stop

- il est impossible d'avoir deux actions `PUT` successives :

  never `(PUT ?M1:MSG . i* . PUT ?M2:MSG)`

- il est impossible d'avoir deux actions `GET` successives :

  never `(GET ?M1:MSG . i* . GET ?M2:MSG)`

- il est impossible qu'une action `PUT !M1` soit suivie d'une action `GET !M2` si la valeur de `M2` est différente de celle de `M1` :

  all `M1:MSG` **in never** `(PUT !M1:MSG . i* . GET ?M2:MSG [not (M1 = M2)])`

- toute action `PUT !M` est inévitablement[1] suivie d'une action `GET !M` :

all `M:MSG` **in** `(PUT !M =>` **finev** `(`**i**`,GET !M))`

C.9 Notes bibliographiques

Le protocole du *bit alterné* a longtemps eu la faveur des concepteurs de systèmes de vérification automatique, principalement en raison de la petite taille de son graphe (quelques dizaines d'états). La version en LOTOS proposée ici permet, lorsque l'on donne à `N` une valeur élevée, d'obtenir des graphes de grande taille, démontrant ainsi les capacités réelles de CÆSAR.

La modélisation proposée s'inspire principalement de deux sources :

- [RSV86] et [RV87] : il s'agit d'une description du protocole, dans le formalisme ATP, qui prend en compte les contraintes de délai. Le medium entre les deux entités est défini sous la forme de deux processus identiques fonctionnant en parallèle. Les messages et les acquittements peuvent être perdus sans que le medium le signale
- [Lon87] : cette description représente le medium comme un processus unique. Les messages et les acquittements peuvent être perdus, mais le medium doit alors envoyer au destinataire une indication de perte de message ou d'acquittement

Par rapport à [RSV86], [RV87] et [Lon87], la transmission des messages (considérés comme des valeurs entières) est effectivement modélisée. Les deux cas d'erreur de transmission ont été modélisés : perte silencieuse et perte signalée au destinataire.

Enfin les contraintes temporelles n'ont pas été conservées parce qu'elles ne peuvent pas s'exprimer simplement en LOTOS. Ceci appelle quelques remarques :

- lorsqu'on remplace une contrainte de délai par un événement "**i**", le nouveau comportement est un sur-ensemble de l'ancien
- toutefois, en règle générale, les "bons algorithmes" ne dépendent pas des valeurs des délais et conservent leurs "bonnes propriétés" (*speed independance*)
- autrement dit l'ajustage des délais permet d'améliorer les performances du système mais il ne doit pas servir à rendre correct un algorithme qui serait faux pour des valeurs de délais quelconques
- toutefois, la modélisation d'un délai par un événement "**i**" peut modifier le comportement du système. Dans le cas du bit alterné, si l'on choisit judicieusement les valeurs des délais ([RSV86], [RV87]), à chaque instant, au plus un message ou acquittement circule sur l'ensemble des deux media (la liaison est *half-duplex*). Si les délais sont mal choisis ou modélisés de manière asynchrone, plusieurs messages ou acquittements peuvent transiter simultanément, dans les deux sens (la liaison devient *full-duplex*)

[1] sous l'hypothèse d'équité

Annexe D

Application 2 : convolution systolique

D.1 Produit de convolution

Soient n nombres $w_1, \ldots\ w_n$ appelés *poids* (*weights*). Soit (x_i) une suite de nombres. On appelle *produit de convolution* de la suite (x_i) par les poids $w_1, \ldots\ w_n$ la suite (y_i) définie par :

$$y_i = \sum_{j=1}^{n} w_j x_{i+j-1}$$

Remarque D-1
D'autres définitions du produit de convolution sont possibles, moyennant un renommage des coefficients $w_1, \ldots\ w_n$ qui sont en nombre fini. ■

D.2 Réseau systolique asynchrone

On cherche à construire un programme LOTOS calculant le produit de convolution. Le système possède une *porte d'entrée*, sur laquelle il reçoit successivement les valeurs de la suite (x_i), et une *porte de sortie*, sur laquelle il émet les valeurs (y_i) correspondantes.

Ce système doit être basé sur les principes systoliques : il est constitué par la juxtaposition de cellules identiques (autant de cellules que de poids w_i) exécutant chacune le même algorithme. Cet algorithme est cyclique et n'utilise que des opérations simples (addition, multiplication).

La plupart du temps, les algorithmes systoliques sont décrits pour un modèle synchrone utilisant une notion de temps global, déterminé par une horloge unique. A chaque top d'horloge, chaque cellule communique avec les cellules voisines.

Le modèle synchrone s'impose tout naturellement lorsque l'on envisage une implémentation en technologie VLSI ou autre. En revanche il ne convient pas pour d'autres types d'implémentations (réseaux de micro-processeurs, par exemple) ni pour les réseaux comprenant plusieurs sortes de cellules, dans lesquels la différence de vitesse entre les cellules interdit l'emploi d'une horloge commune.

Dans un modèle asynchrone, les cellules communiquent par rendez-vous. Chacune cellule ne peut pas

émettre ou recevoir simultanément sur deux portes distinctes. En outre, en LOTOS, deux rendez-vous distincts n'ont jamais lieu simultanément.

Le comportement synchrone peut être considéré comme une abstraction du comportement asynchrone, obtenue en considérant que plusieurs rendez-vous ont lieu simultanément.

Il est possible de mettre sous forme asynchrone la plupart des algorithmes systoliques : la structure du réseau est conservée, ainsi que le comportement des cellules. En revanche la spécification asynchrone impose des contraintes supplémentaires sur l'ordonnancement des rendez-vous afin d'éviter les interblocages.

Remarque D-2
Pour permettre l'initialisation du système et le chargement des n premières valeurs $x_1, \ldots\ x_n$, il faut établir une distinction entre le *comportement transitoire* d'une cellule à l'initialisation du système et le *comportement permanent* qui lui succède ; typiquement le régime transitoire prend fin avec la réception de la $n^{\text{ième}}$ valeur x_n. ■

Dans la suite on présente quatre architectures systoliques qui calculent la convolution. Ces quatre schemas sont connus sous les appellations suivantes : B1, F, W1 et W2.

Remarque D-3
On s'est restreint aux exemples dans lesquels chaque cellule est associée à un poids constant (*weights stay*) à l'exclusion des schémas dans lesquels les poids se déplacent : B2, R1 et R2. Ce choix n'est pas dicté par une limitation quelconque de LOTOS ou de CÆSAR. ■

D.3 Environnement

Le comportement du réseau systolique est paramétré par la suite des valeurs (x_i) qu'il reçoit en entrée : il ne s'agit pas d'un système fermé.

Or CÆSAR ne peut traiter que des systèmes fermés puisqu'il est généralement impossible de construire le graphe d'un comportement lorsque l'on ne connaît pas les valeurs qu'ont les variables de ce comportement. Cette restriction interdit la vérification du réseau pris isolément.

Une première solution consiste à connecter l'entrée du réseau à un générateur qui fournit une suite fixée de valeurs (x_i). Pour que le graphe correspondant au comportement de l'ensemble *réseau+générateur* soit fini, il faut que la suite (x_i) fournie par le générateur soit finie ou périodique à partir d'un certain rang.

Dans le cas présent, il serait toutefois préférable d'effectuer une vérification formelle du réseau paramétré et ne pas se contenter de prouver que le réseau fonctionne correctement lorsqu'on l'instancie par une suite (x_i) fixée.

Une seconde solution solution utilise une technique d'*évaluation symbolique*. Comme dans la première solution, l'entrée du réseau systolique est reliée à un générateur qui énumère une suite finie ou périodique (x_i).

La différence provient du fait que les éléments x_i et w_j ne sont plus des valeurs numériques mais des noms symboliques de variables. Les opérateurs d'addition et de multiplication opèrent alors sur des expressions symboliques et non plus sur des nombres.

Exemple D-1
Le résultat de "w_1" `*` "x_1" est l'expression symbolique "w_1x_1" ; de même que le résultat de "w_1x_1" `+` "w_2x_2" est l'expression symbolique "$w_1x_1 + w_2x_2$". ■

Les expressions symboliques sont décrites par le type abstrait `EXP`. Les éléments des suites $(x_1, \ldots\ x_m)$

et ($w_1, \ldots\ w_n$) sont dénotés par des fonctions d'arité nulle `X1`, ... `Xm` et `W1`, ... `Wn`. L'addition et la multiplication sont représentées par les opérateurs binaires `+` et `*`. On note `0` l'élément neutre pour l'addition ; cette propriété suffit pour simplifier les expressions symboliques.

```
type EXP is
   sorts EXP
   opns 0 : -> EXP
        X1, ... Xm : -> EXP
        W1, ... Wn : -> EXP
        _+_, _*_ : EXP, EXP -> EXP
   eqns
      forall X:EXP ofsort EXP
         X + 0 = X;
         0 + X = X
endtype
```

On utilise aussi un type entier noté `NATURAL`, comportant une sorte `NAT`, les notations d'entiers de 0 à n et les opérations usuelles de soustraction et de comparaison :

```
type NATURAL is BOOLEAN
   sorts NAT
   opns 0, 1, ... n : -> NAT
        _-_ : NAT, NAT -> NAT
        _eq_, _gt_ : NAT, NAT -> BOOL
endtype
```

Le réseau systolique est dénoté par le processus `ARRAY [X, Y]` où `X` est la porte d'entrée et `Y` la porte de sortie. On note n le nombre de cellules et (w_j) leurs poids respectifs. Diverses définitions du processus `ARRAY` seront proposées dans les sections suivantes.

Le générateur fournit une suite finie (x_i) pour i variant entre 1 et un entier m. Il est décrit par le processus `GENERATOR`. On donne également la définition d'un processus, noté `ZERO`, qui sera utilisé par la suite.

```
specification SYSTOLIC_CONVOLUTION [Y] : noexit behaviour
   hide X in
      (
      GENERATOR [X]
      |[X]|
      ARRAY [X, Y] (W1, ... Wn)
      )
where
   process GENERATOR [X] : noexit :=
      X !X1;
         X !X2;
            ...
               X !Xm;
                  stop
   endproc

   process ZERO [Y] : noexit :=
      Y !(0 of EXP);
         ZERO [Y]
   endproc
endspec
```

Remarque D-4

Par la suite, on donnera aussi les noms X1, ... Xn à des portes, sachant que LOTOS le permet et

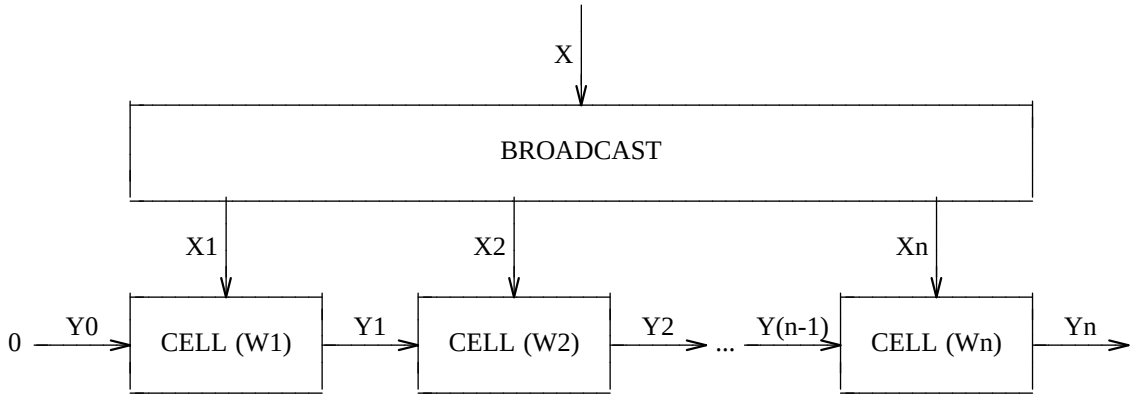
qu'il n'y a aucune confusion possible entre portes et opérations ■

D.4 Architecture B1

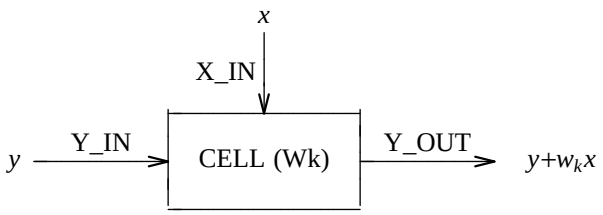
Le schema B1 (*broadcast*) se compose de n processus `CELL` disposés en pipe-line. Le flux de données entre les cellules est constitué de sommes partielles ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des expressions de la forme :

$$y_i^k = \sum_{j=1}^{k} w_j x_{i+j-1}$$

La valeur courante de x_i est reçue par un processus `BROADCAST` et diffusée ensuite à toutes les cellules. Il ne s'agit pas à proprement parler d'une architecture systolique pure.



La $k^{\text{ième}}$ cellule a pour poids w_k ; elle possède deux portes d'entrée `X_IN` et `Y_IN` et une porte de sortie `Y_OUT`. En comportement permanent, elle lit une valeur x sur `X_IN` puis une valeur y sur `Y_IN` et émet $y + w_k x$ sur `Y_OUT`. En comportement transitoire elle lit (sans les traiter) $(k-1)$ valeurs sur la porte `X_IN`.



```
process ARRAY [X, Yn] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn in
      (
      BROADCAST [X, X1, ... Xn]
      |[X1, ... Xn]|
         (
         hide Y0, ... Y(n-1) in
            (
            ZERO [Y0]
            |[Y0]|
            CELL [X1, Y0, Y1] (W1, 1)
            |[Y1]|
            CELL [X2, Y1, Y2] (W2, 2)
            |[Y2]|
            ...
            |[Y(n-1)]|
            CELL [Xn, Y(n-1), Yn] (Wn, n)
            )
         )
      )
where
   process CELL [X_IN, Y_IN, Y_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            CELL [X_IN, Y_IN, Y_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               Y_OUT !(Y + (W * X));
                  CELL [X_IN, Y_IN, Y_OUT] (W, 1)
   endproc

   process BROADCAST [IN0, OUT1, ... OUTn] : noexit :=
      IN0 ?X:EXP;
         OUTn !X;
            OUT(n-1) !X;
               ...
                  OUT1 !X;
                     BROADCAST [IN0, OUT1, ... OUTn]
   endproc
endproc
```

Le tableau suivant aide à mieux comprendre le fonctionnement du réseau systolique. Chaque ligne correspond à un instant donné et les lignes sont rangées chronologiquement.

La colonne de gauche contient la séquence des événements qui ont lieu dans le système. Chaque événement est constitué d'un nom de porte et de la valeur échangée au moment du rendez-vous. Les événements visibles à l'extérieur du réseau (entrée d'une valeur x_i ou sortie d'une valeur y_i) sont marqués d'une étoile.

Les autres colonnes décrivent l'état courant de chaque processus. L'état d'attente pour émettre (*resp.* pour recevoir) une valeur sur une porte `G` est noté "`G !`" (*resp.* "`G ?`"). Lorsque l'état d'un processus n'est pas modifié par l'événement courant, on met un guillemet dans la case correspondante.

Le tableau décrit le régime transitoire et l'établissement du régime permanent. Les parties non hachurées mettent en évidence le premier cycle du comportement de chaque processus, une fois atteint le régime permanent.

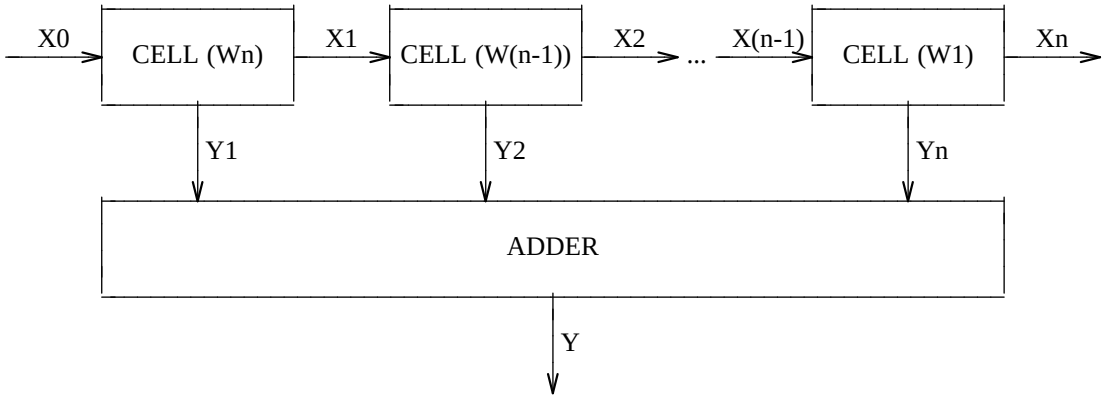
événement	*CELL (W1)*	*CELL (W2)*	*CELL (W3)*	*BROADCAST*
initialement :	`X1 ?` *(K = 1)*	`X2 ?` *(K = 2)*	`X3 ?` *(K = 3)*	`X ?`
`X !`x_1 *	"	"	"	`X3 !`
`X3 !`x_1	"	"	`X3 ?` *(K = 2)*	`X2 !`
`X2 !`x_1	"	`X2 ?` *(K = 1)*	"	`X1 !`
`X1 !`x_1	`Y0 ?`	"	"	`X ?`
`X !`x_2 *	"	"	"	`X3 !`
`X3 !`x_2	"	"	`X3 ?` *(K = 1)*	`X2 !`
`X2 !`x_2	"	`Y1 ?`	"	`X1 !`
`Y0 !`0	`Y1 !`	"	"	"
`Y1 !`w_1x_1	`X1 ?`	`Y2 !`	"	"
`X1 !`x_2	`Y0 ?`	"	"	`X ?`
`X !`x_3 *	"	"	"	`X3 !`
`X3 !`x_3	"	"	`Y2 ?`	`X2 !`
`Y2 !`$w_1x_1 + w_2x_2$	"	`X2 ?`	`Y3 !`	"
`Y3 !`$w_1x_1 + w_2x_2 + w_3x_3$ *	"	"	`X3 ?`	"
`X2 !`x_3	"	`Y1 ?`	"	`X1 !`
`Y0 !`0	`Y1 !`	"	"	"
`Y1 !`w_1x_2	`X1 ?`	`Y2 !`	"	"
`X1 !`x_3	`Y0 ?`	"	"	`X ?`
`X !`x_4 *	"	"	"	`X3 !`
`X3 !`x_4	"	"	`Y2 ?`	`X2 !`
`Y2 !`$w_1x_2 + w_2x_3$	"	`X2 ?`	`Y3 !`	"
`Y3 !`$w_1x_2 + w_2x_3 + w_3x_4$ *	"	"	`X3 ?`	"

D.5 Architecture F

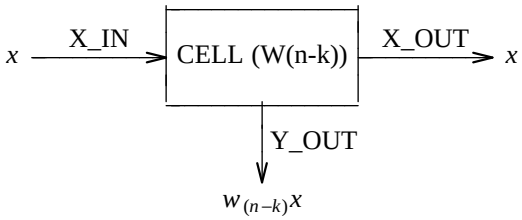
Le schema F (*fan-in*) se compose de n processus `CELL` disposés en pipe-line. Le flux de données entre les cellules est constitué des valeurs successives de x_i ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :

$$x_i^k = x_{i+n-k-1}$$

Chaque cellule effectue un produit partiel $w_{n-k}x_{i+n-k-1}$; ces valeurs sont recueillies par un processus `ADDER` qui en calcule la somme. Il ne s'agit pas à proprement parler d'une architecture systolique pure.



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède une porte d'entrée `X_IN` et deux portes de sortie `X_OUT` et `Y_OUT`. En comportement permanent, elle lit une valeur x sur `X_IN` puis envoie successivement $w_{n-k}x$ sur `Y_OUT` et x sur `X_OUT`. En comportement transitoire elle lit (sans les traiter) $n-k$ valeurs sur la porte `X_IN`.



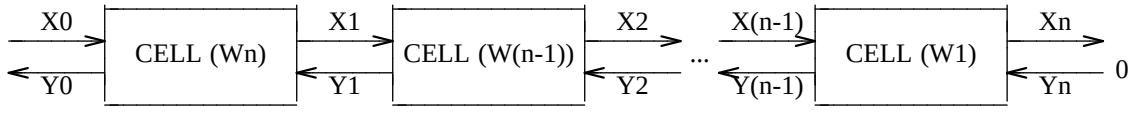

```
process ARRAY [X0, Y] (W1, ... Wn:EXP) : noexit :=
   hide Y1, ... Yn in
      (
      hide X1, ... Xn in
         (
         CELL [X0, Y1, X1] (Wn, n)
         |[X1]|
         CELL [X1, Y2, X2] (W(n-1), (n-1))
         |[X2]|
         ...
         |[X(n-1)]|
         CELL [X(n-1), Yn, Xn] (W1, 1)
         )
      |[Y1, ... Yn]|
      ADDER [Y1, ... Yn, Y]
      )
where
   process CELL [X_IN, Y_OUT, X_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_OUT, X_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_OUT !(W * X);
               X_OUT !X;
                  CELL [X_IN, Y_OUT, X_OUT] (W, 1)
   endproc

   process ADDER [IN1, ... INn, OUT] : noexit :=
      IN1 ?Y1:EXP;
         IN2 ?Y2:EXP;
            ...
               INn ?Yn:EXP;
                  OUT !(Y1 + Y2 + ... Yn);
                     ADDER [IN1, ... INn, OUT]
   endproc
endproc
```

événement	*CELL (W3)*	*CELL (W2)*	*CELL (W1)*	*ADDER*
initialement :	`X0 ?` *(K = 3)*	`X1 ?` *(K = 2)*	`X2 ?` *(K = 1)*	`Y1 ?`
`X0 !`x_1 *	`X1 !`	"	"	"
`X1 !`x_1	`X0 ?` *(K = 2)*	`X2 !`	"	"
`X2 !`x_1	"	`X1 ?` *(K = 1)*	`Y3 !`	"
`X0 !`x_2 *	`X1 !`	"	"	"
`X1 !`x_2	`X0 ?` *(K = 1)*	`Y2 !`	"	"
`X0 !`x_3 *	`Y1 !`	"	"	"
`Y1 !`w_3x_3	`X1 !`	"	"	`Y2 ?`
`Y2 !`w_2x_2	"	`X2 !`	"	`Y3 ?`
`Y3 !`w_1x_1	"	"	`X3 !`	`Y !`
`Y !`$w_3x_3 + w_2x_2 + w_1x_1$ *	"	"	"	`Y1 ?`
`X3 !`x_1	"	"	`X2 ?`	"
`X2 !`x_2	"	`X1 ?`	`Y3 !`	"
`X1 !`x_3	`X0 ?`	`Y2 !`	"	"
`X0 !`x_4 *	`Y1 !`	"	"	"
`Y1 !`w_3x_4	`X1 !`	"	"	`Y2 ?`
`Y2 !`w_2x_3	"	`X2 !`	"	`Y3 ?`
`Y3 !`w_1x_2	"	"	`X3 !`	`Y !`
`Y !`$w_3x_4 + w_2x_3 + w_1x_2$ *	"	"	"	`Y1 ?`
`X3 !`x_2	"	"	`X2 ?`	"
`X2 !`x_3	"	`X1 ?`	"	"
`X1 !`x_4	`X0 ?`	"	"	"

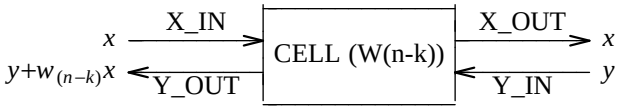
D.6 Architecture W1

Le schema W1 (*pure systolic — weights stay*) se compose de n processus `CELL` disposés en double pipe-line. Deux flux de données circulent dans des directions opposées :

- les valeurs successives de x_i circulent dans un sens ; la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :
$$x_i^k = x_{i+n-k-1}$$
- les sommes partielles circulent dans le sens inverse ; la $k^{\text{ième}}$ cellule transmet à la $(k-1)^{\text{ième}}$ des expressions de la forme :
$$y_i^k = \sum_{j=k}^{n} w_j x_{i+j-1}$$



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède deux portes d'entrée `X_IN` et `Y_IN` et deux portes de sortie `X_OUT` et `Y_OUT`. En comportement permanent, elle lit successivement une valeur x sur `X_IN` puis une valeur y sur `Y_IN` avant d'envoyer x sur `X_OUT` puis $y + w_{n-k}x$ sur `Y_OUT`. En comportement transitoire elle lit $n-k$ valeurs x sur la porte `X_IN` qu'elle retransmet immédiatement sur la porte `X_OUT`.



```
process ARRAY [X0, Y0] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn, Y1, ... Yn in
      (
      CELL [X0, Y0, X1, Y1] (Wn, n)
      |[X1, Y1]|
      CELL [X1, Y1, X2, Y2] (W(n-1), (n-1))
      |[X2, Y2]|
      ...
      |[X(n-1), Y(n-1)]|
      CELL [X(n-1), Y(n-1), Xn, Yn] (W1, 1)
      |[Yn]|
      ZERO [Yn]
      )
where
   process CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               X_OUT !X;
                  Y_OUT !(Y + (W * X));
                     CELL [X_IN, Y_OUT, X_OUT, Y_IN] (W, 1)
   endproc
endproc
```

événement	*CELL (W3)*	*CELL (W2)*	*CELL (W1)*
initialement :	X0 ? $(K = 3)$	X1 ? $(K = 2)$	X2 ? $(K = 1)$
X0 !x_1 *	X1 !	"	"
X1 !x_1	X0 ? $(K = 2)$	X2 !	"
X2 !x_1	"	X1 ? $(K = 1)$	Y3 ?
X0 !x_2 *	X1 !	"	"
X1 !x_2	X0 ? $(K = 1)$	Y2 ?	"
X0 !x_3 *	Y1 ?	"	"
Y3 !0	"	"	X3 !
X3 !x_1	"	"	Y2 !
Y2 !w_1x_1	"	X2 !	X2 ?
X2 !x_2	"	Y1 !	Y3 ?
Y1 !$w_1x_1 + w_2x_2$	X1 !	X1 ?	"
X1 !x_3	Y0 !	Y2 ?	"
Y0 !$w_1x_1 + w_2x_1 + w_3x_3$ *	X0 ?	"	"
X0 !x_4 *	Y1 ?	"	"
Y3 !0	"	"	X3 !
X3 !x_2	"	"	Y2 !
Y2 !w_1x_2	"	X2 !	X2 ?
X2 !x_3	"	Y1 !	Y3 ?
Y1 !$w_1x_2 + w_2x_3$	X1 !	X1 ?	"
X1 !x_4	Y0 !	Y2 ?	"
Y0 !$w_1x_2 + w_2x_3 + w_3x_4$ *	X0 ?	"	"

Remarque D-5
Dans le modèle synchrone, la suite de valeurs qu'il faut fournir en entrée du réseau n'est pas la suite

(x_i) mais la suite (x'_i) définie par :

$$\begin{cases} x'_{2i} & = x_i \\ x'_{2i+1} & = 0 \end{cases}$$

La suite (x'_i) est deux fois plus lente que la suite (x_i). Ce n'est pas le cas dans le modèle asynchrone : il suffit d'alimenter le réseau avec la suite (x_i). ■

D.7 Architecture W2

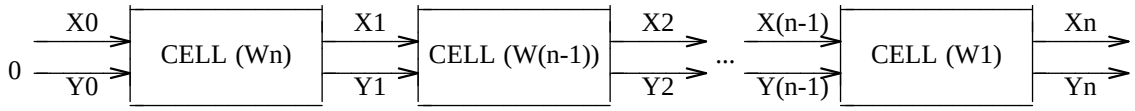
Le schema W2 (*pure systolic — weights stay*) se compose de n processus `CELL` disposés en double pipe-line. Deux flux de données circulent dans la même direction :

- les valeurs successives de x_i : la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des valeurs de la forme :

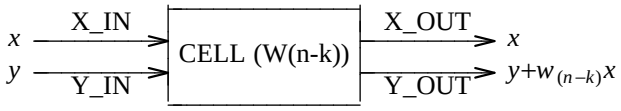
$$x_i^k = x_{i+n-k-1}$$

- les sommes partielles : la $k^{\text{ième}}$ cellule transmet à la $(k+1)^{\text{ième}}$ des expressions de la forme :

$$y_i^k = \sum_{j=k}^{n} w_j x_{i+n-k-1}$$



La $k^{\text{ième}}$ cellule a pour poids w_{n-k} ; elle possède deux portes d'entrée `X_IN` et `Y_IN` et deux portes de sortie `X_OUT` et `Y_OUT`. En comportement permanent, elle lit successivement une valeur x sur `X_IN` puis une valeur y sur `Y_IN` avant d'envoyer $y + w_{n-k}x$ sur `Y_OUT` puis x sur `X_OUT`. En comportement transitoire elle lit $n-k$ valeurs x sur la porte `X_IN` qu'elle retransmet immédiatement sur la porte `X_OUT`.



```
process ARRAY [X0, Yn] (W1, ... Wn:EXP) : noexit :=
   hide X1, ... Xn, Y0, ... Y(n-1) in
      (
      ZERO [Y0]
      |[Y0]|
      CELL [X0, Y0, X1, Y1] (Wn, n)
      |[X1, Y1]|
      CELL [X1, Y1, X2, Y2] (W(n-1), (n-1))
      |[X2, Y2]|
      ...
      |[X(n-1), Y(n-1)]|
      CELL [X(n-1), Y(n-1), Xn, Yn] (W1, 1)
      )
where
   process CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W:EXP, K:NAT) : noexit :=
      [K gt 1] ->
         X_IN ?X:EXP;
            X_OUT !X;
               CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W, K - 1)
      []
      [K eq 1] ->
         X_IN ?X:EXP;
            Y_IN ?Y:EXP;
               Y_OUT !(Y + (W * X));
                  X_OUT !X;
                     CELL [X_IN, Y_IN, X_OUT, Y_OUT] (W, 1)
   endproc
endproc
```

événement	*CELL (W3)*	*CELL (W2)*	*CELL (W1)*
initialement :	X0 ? $(K = 3)$	X1 ? $(K = 2)$	X2 ?
X0 ! x_1 *	X1 !	"	"
X1 ! x_1	X0 ? $(K = 2)$	X2 ! $(K = 2)$	"
X2 ! x_1	"	X1 ?	Y2 ?
X0 ! x_2 *	X1 ! $(K = 2)$	"	"
X1 ! x_2	X0 ?	Y1 ?	"
X0 ! x_3 *	Y0 ?	"	"
Y0 ! 0	Y1 !	"	"
Y1 ! w_3x_3	X1 !	Y2 !	"
Y2 ! $w_3x_3 + w_2x_2$	"	X2 !	Y3 !
Y3 ! $w_3x_3 + w_2x_2 + w_1x_1$ *	"	"	X3 !
X3 ! x_1	"	"	X2 ?
X2 ! x_2	"	X1 ?	Y2 ?
X1 ! x_3	X0 ?	Y1 ?	"
X0 ! x_4 *	Y0 ?	"	"
Y0 ! 0	Y1 !	"	"
Y1 ! w_3x_4	X1 !	Y2 !	"
Y2 ! $w_3x_4 + w_2x_3$	"	X2 !	Y3 !
Y3 ! $w_3x_4 + w_2x_3 + w_1x_2$ *	"	"	X3 !
X3 ! x_2	"	"	X2 ?
X2 ! x_3	"	X1 ?	"
X1 ! x_4	X0 ?	"	"

Remarque D-6
Dans le modèle synchrone, le flux des (y_i) est deux fois plus rapide que le flux des (x_i). Ce n'est pas le cas dans le modèle asynchrone : les deux flux ont la même "vitesse". Plus précisément, si l'on note

$\Delta(e_1, e_2)$ le nombre d'événements ayant lieu entre les deux événements e_1 et e_2, alors :

$$(\exists c)\ \ (\forall i)\ \ \Delta(\text{“}\texttt{X0 !}x_i\text{”}, \text{“}\texttt{X0 !}x_{i+1}\text{”}) = \Delta(\text{“}\texttt{Y3 !}y_i\text{”}, \text{“}\texttt{Y3 !}y_{i+1}\text{”}) = c$$

■

D.8 Validation

L'utilisation de CÆSAR a permis de mettre au point et de valider les descriptions en LOTOS des différents réseaux systoliques. Dans chaque cas il a fallu donner des valeurs particulières au nombre n de cellules et au nombre m de valeurs x_i fournies par le générateur. En revanche, les valeurs x_i ont été manipulées sous forme d'expressions symboliques[2] afin de prouver la correction du réseau pour toute suite d'entrée (x_i) de longueur m.

A l'aide de CÆSAR les graphes correspondants aux réseaux systoliques ont été construits, ce qui a permis la détection de plusieurs erreurs : blocages, inversion des coefficients w_i, ... On constate que les graphes ainsi obtenus comportent approximativement autant d'arcs que d'états, ce qui se justifie par le fait que les différentes parties du réseau sont fortement synchronisées : à chaque instant, le nombre de rendez-vous possibles avoisine 1. Ce degré de non-déterminisme n'est pas le même pour les quatre schémas systoliques, ce qui explique les différences importantes entre les tailles des graphes.

Le tableau suivant regroupe les résultats expérimentaux obtenus. L'interprétation des temps et des débits est la même que celle donnée pour l'exemple du bit alterné (§ C.8, p. 8).

réseau	n	m	places	transitions	variables	états	arcs	temps	débit
B1	3	9	54	46	29	101 451	151 146	286:50–10:47	5–157
F	7	19	65	69	34	438	669	5:23–3:51	1.3–1.9
W1	3	6	33	37	14	64 085	87 775	88:44–4:54	12–218
W2	7	19	78	97	34	355	463	5:30–3:23	1.1–1.7

Ces graphes ont été minimisés par le logiciel ALDEBARAN [Fer88] selon la relation d'équivalence observationnelle ; dans tous les cas on a ainsi pu vérifier que le graphe minimal est une chaîne de $m - n + 1$ arcs dont le $i^{\text{ème}}$ a pour label :

$$\texttt{Y !}w_1x_i + w_2x_{i+1} + \ldots + w_nx_{i+n-1}$$

D.9 Notes bibliographiques

Le produit de convolution est caractéristique d'un grand nombre d'applications qui admettent une solution systolique : filtrage, reconnaissance de formes, corrélation, interpolation, évaluation polynômiale, transformation de Fourier discrète, addition et division polynômiale.

Les architectures B1, F, W1 et W2 sont tirées de [Kun82] qui recense et compare les différents schémas systoliques connus. Plusieurs tentatives ont été faites pour décrire des réseaux systoliques dans des langages parallèles et pour valider cette spécification. Les langages employés sont généralement synchrones : [HP86] est basé sur LUSTRE, [Gri88] utilise une variante synchrone de CSP.

[2] implémentées par des chaînes de caractères

Bibliographie

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), mai 1988.

[Gri88] E. Pascal Gribomont. Proving Systolic Arrays. In Max Dauchet and Maurice Nivat, editors, *CAAP'88 13th Colloquium on Trees in Algebra and Programming, Nancy, France*, pages 185–199, Berlin, mars 1988. Springer Verlag.

[HP86] Nicolas Halbwachs and Daniel Pilaud. Use of a Real-Time Declarative Language for Systolic Array Design and Simulation. In Will Moore, Andrew McCabe, and Roddy Urquhart, editors, *Proceedings of the International Workshop on Systolic Arrays, Oxford*, pages 81–90, Bristol and Boston, juillet 1986. Adam Hilger.

[Kun82] H. T. Kung. Why systolic architectures ? *Computer*, 15(1):37–46, janvier 1982.

[Lon87] Brigitte Lonc. *Techniques formelles de vérification des systèmes distribués : application aux protocoles de communication des systèmes ouverts*. Thèse de Doctorat, Conservatoire National des Arts et Métiers (Paris), juin 1987.

[RSV86] Jean-Luc Richier, Joseph Sifakis, and Jacques Voiron. ATP — An Algebra for Timed Processes. Rapport technique SPECTRE C1, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, décembre 1986.

[RV87] Jean-Luc Richier and Jacques Voiron. Spécification et analyse d'un protocole de communication à l'aide du système XESAR. *BIGRE+GLOBULE*, 55:137–148, juillet 1987.