Automates et Structures de Contrôle en Lustre: II

Marc Pouzet
LIP6

Marc.Pouzet@lip6.fr

en collaboration avec Jean-Louis Colaço et Bruno Pagano
(Esterel-Technologies)

Motivations (rappel ACI, janvier 2005)

- combiner des parties data-flow et des parties impératives
- condition d'activation entre plusieurs "modes"
- réinitialisation modulaire et automates
- au sein d'un même language (outil)
- dans un formalisme uniforme (certif., qualité du code, lisibilité)
- accepter tout **Lustre** et conserver la sémantique du noyau
- légère (si possible) pour conserver le générateur qualifié existant

Solutions I

- travaux de Maraninchi & all, Poigné & all
- Mécanisme de type "édition de lien". Un noeud a mémoire est toujours plus ou moins représenté par une paire $f : M \times I \rightarrow O \times M \ + \ mo : M$. Il suffit de s'accorder sur les conventions d'appel
- ça existe dans les outils du domaine (e.g., Scade+Esterel, Simulink+StateFlow)
- trop grande séparation et pas/peu d'imbrication fine
- certification? Efficacité/simplicité du code?
- mieux? *modes* en Simulink (cf. exposé Paul Caspi)

Solutions II

- automates de mode: Maraninchi & Rémond (ESOP98, CC00, SCP)
- fournir une construction d'automate où les noeuds sont définis par des équations **Lustre**
- hiérarchie et concurrence naturelles (i.e., celles de **Lustre**)
- compilation *ad-hoc* assurant qu'une seul état est actif à chaque instant
- restrictions sur le langage **Lustre** considéré (e.g., `pre`, `when`, `current`, appel de noeuds)
- un mode de transition (Moore avec reset)

Solutions III

- Signal GTI (Eric Rutten)
- structures de contrôle fondées sur le mécanisme des horloges de **Signal**
- tout **Signal**, uniformité avec le langage de base
- pas de compilation particulière (**Signal** s'en charge)
- calcul d'horloge plus complexe que celui de **Lustre**
- demande beaucoup au compilateur
- écriture peu pratique

Expériences IV

- **Lucid Synchrone**: constructions `when`, `merge` et `reset`
- calcul d'horloge simple (essentiellement celui de **Lustre**)
- compilation modulaire où un calcul est effectué seulement lorsque son horloge est vraie
- `when`+`merge`+`reset` permettent de coder les automates de modes (PPDP'00)
- code efficace (i.e., un seul état actif à chaque instant)
- le calcul d'horloge permet d'expliquer précisément ce qui se passe
- pas de restriction sur le langage **mais** trop bas niveau
- pas de syntaxe particulière (les automates doivent être codés à la main), écriture peu pratique, peu lisible

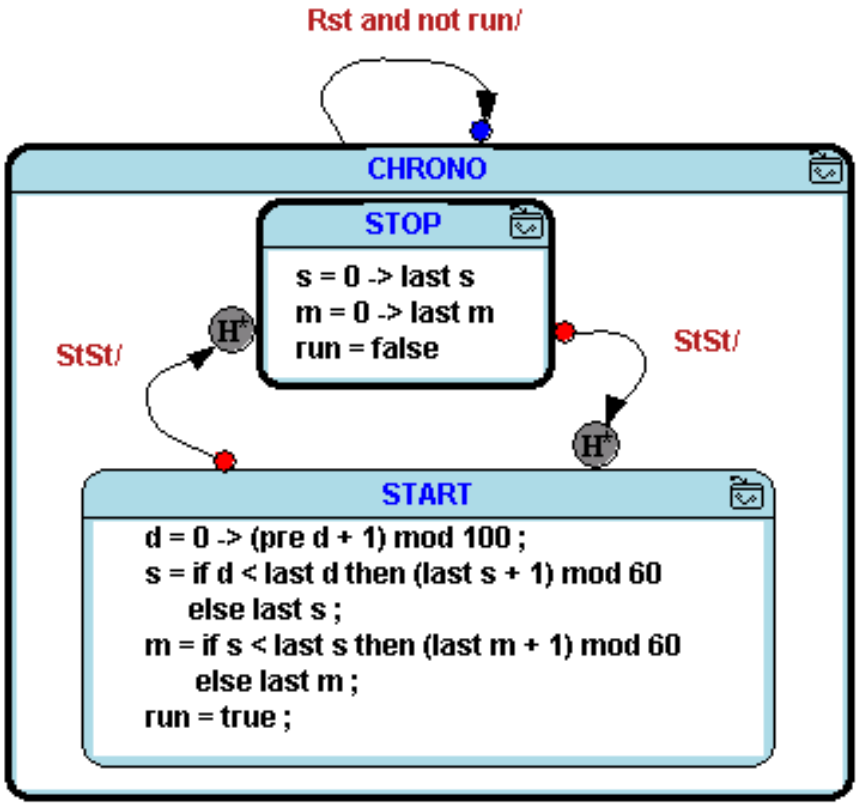
Approche proposée

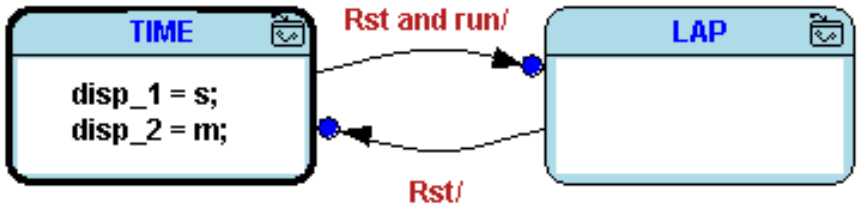
- proposition d'extension de Scade/Lustre avec une construction d'automates (avec Jean-Louis Colaço, Bruno Pagano)
- fondé sur l'utilisation de `when`, `merge` et `reset` et (donc) les horloges
- fournir une syntaxe *ad-hoc* et suivre un principe de compilation *dirigée par les horloges*
- essentiellement de la transformation source-à-source
- expérimentation dans **ReLuC** (SYNCHRON, Jean-Louis Colaço) et **Lucid Synchrone** (démo, ACI Alidecs, janvier 2005)

Principes

- demander peu à la génération de code (fournir les structures de contrôles qui se compilent bien)
- préserver les principes du data-flow: une seule définition par cycle, principe de substitution (liaison statique)
- constructions ayant une sémantique simple (qui puisse passer la certification)
- sémantique *par traduction* dans un noyau data-flow de base
- s'appuyer sur les horloges: sémantique précise + on sait les compiler efficacement
- transformation source-à-source préservant les types, horloges, etc.
  - $H \overset{k}{\vdash} e : ty$ ssi $H \overset{k}{\vdash} T(e) : ty$ et $H \vdash e : cl$ ssi $H \vdash T(e) : cl$
  - idem pour la causalité + analyse d'initialisation

Le Chronomètre





```
ssm_node CHRONO(StSt, Rst: bool)
returns (disp_1, disp_2: int)
var s, m: int;   -- internal time streams
    run : bool;  -- flag to commute Rst button
let automaton
    initial state CHRONO
    let automaton
         initial state STOP
         unless StSt resume START ;
         let
           s = 0 -> last s;
           m = 0 -> last m;
           run = false;
         tel
        state START
        unless StSt resume STOP ;
        var d: int; -- 1/100 th time stream
        let
          d = 0 -> (pre d + 1) mod 100;
          s = if d < pre d then (last s + 1) mod 60
              else last s;
          m = if s < last s then (last m + 1) mod 60
              else last m;
          run = true;
        tel;
    tel
    until Rst and not run restart CHRONO;
  automaton DISPLAY
    initial state TIME
      let disp_1 = s; disp_2 = m; tel
      until Rst and run restart LAP;
    state LAP until Rst restart TIME;
tel;
```

```
let node chrono (stSt, rst) = (disp_1, disp_2) where
  rec automaton
        CHRONO ->
         do automaton
              STOP ->
               do  s = 0 -> last s
               and m = 0 -> last m
               and run = false
               unless stSt continue START
            | START ->
               let d = 0 -> (pre d + 1) mod 100 in
               do s = if d < pre d
                      then (last s + 1) mod 60
                      else last s
               and m = if s < last s
                       then (last m + 1) mod 60
                       else last m
               and run = true
               unless stSt continue STOP
            end
          until rst & not run then CHRONO
      end and
      automaton
        TIME ->
         do disp_1 = s
         and disp_2 = m
         until rst & run then LAP
      | LAP ->
         do until rst then TIME
      end
```

Un noyau data-flow

$$
\begin{array}{rcl}
e & ::= & C \mid x \mid e \ \texttt{fby}\ e \mid (e, e) \mid x(e) \\
& & \mid \texttt{let}\ D\ \texttt{in}\ e \mid x(e)\ \texttt{every}\ e \\
& & \mid e\ \texttt{when}\ C(e) \mid \texttt{merge}\ e\ (C \rightarrow e)\ ...\ (C \rightarrow e)
\end{array}
$$

$$
\begin{array}{rcl}
D & ::= & D\ \texttt{and}\ D \mid p = e \mid \texttt{clock}\ x = e \\
p & ::= & x \mid (p, p) \\
d & ::= & \texttt{let}\ x = e \mid \texttt{let fun}\ x(pat) = e \\
& & \mid \texttt{let node}\ x(pat) = e \mid d; d \\
td & ::= & \texttt{type}\ t \mid \texttt{type}\ t = C_1 + ... + C_n \mid td; td
\end{array}
$$

Merge n-aire

`merge` combine deux flots complémentaires (d'horloges opposées).

```
let alternate c =
  let rec nat1 = 0 -> pre nat1 + 1 in
  let rec nat2 = 0 -> pre nat2 + 1 in
  let rec o = merge c (nat1 when c) nat2 in
  o
```

- `merge` peut être généralisé à n entrées en introduisant des types énumérés. Si t est un type énuméré ($t = \{C_1, ..., C_n\}$), on pourra écrire, par exemple:
- `merge` c $(C_1 \rightarrow e_1$ `when` $C_1(c))$ ... $(C_n \rightarrow e_n$ `when` $C_n(c))$
- e `when` c s'écrit maintenant e `when True`(c)
- la sémantique s'étend facilement et on sait bien le compiler
- donc bonne base pour la compilation

Réinitialisation modulaire

- s'applique à une instance de noeud
- $e_1(e_2)$ `every` e_3 ré-initialise l'application à chaque fois que e_3 est vrai.
- Est-ce primitif? Oui et non
- traduction (modulaire) du noyau avec réinitialisation en un noyau sans (essentiellement en modifiant le code de l'initialisation `fby`)
- e_1 `fby` e_2 devient `if` c `then` e_1 `else` e_1 `fby` e_2
- demande beaucoup à la génération de code alors que c'est trivial à compiler!
- traduction utile pour la vérif, primitif pour la compilation
- donc, bonne base pour la compilation

Typage

$$
\begin{array}{lll}
\sigma & ::= & \forall \alpha_1, ..., \alpha_n . t \\
t & ::= & B \mid t \xrightarrow{k} t \mid t \times t \mid \alpha \\
B & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \\
k & ::= & 0 \mid 1 \\
H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n] \\
T & ::= & [C_{11} + ... + C_{1i}/t_1, ..., C_{k1} + ... + C_{kj}/t_k] \\
\text{Jugement} & ::= & H \vdash e : t
\end{array}
$$

- $\texttt{int} \xrightarrow{1} \texttt{int}$ désigne une fonction à mémoire
- $\texttt{int} \xrightarrow{0} \texttt{int}$ désigne une fonction combinatoire
- cette séparation est normale car elles ont une représentation différente

$$
\begin{aligned}
H_0 = [\ & \texttt{.fby.} : \forall\alpha.\alpha \times \alpha \xrightarrow{1} \alpha, \\
& \texttt{pre}\,(.) : \forall\alpha.\alpha \xrightarrow{1} \alpha, \\
& \texttt{if . then . else .} : \forall\alpha.\texttt{bool} \times \alpha \times \alpha \xrightarrow{0} \alpha]
\end{aligned}
$$

Horloges

$$
\begin{array}{lcl}
\rho & ::= & \forall\alpha_1, ..., \alpha_n.\forall X_1, ..., X_m.cl \mid cl \\
cl & ::= & cl \to cl \mid cl \times cl \mid (c : ck) \mid ck \\
ck & ::= & \texttt{base} \mid \alpha \mid ck \ \texttt{on}\ C(c) \\
c & ::= & X \mid m \\
H & ::= & [x_1 : \rho_1, ..., x_n : \rho_n] \\
Jugement & ::= & H \vdash e : cl
\end{array}
$$

$$
\begin{array}{l}
H_0 = [\ \ . \ \texttt{fby}\ . : \forall\alpha.\alpha \times \alpha \to \alpha, \\
\qquad\quad \texttt{pre}\,(.) : \forall\alpha.\alpha \to \alpha, \\
\qquad\quad \texttt{if}\ .\ \texttt{then}\ .\ \texttt{else}\ . : \forall\alpha.\alpha \times \alpha \times \alpha \to \alpha]
\end{array}
$$

Le langage étendu

$$
\begin{array}{lcl}
e & ::= & C \mid x \mid e \ \texttt{fby}\ e \mid (e, e) \mid x(e) \mid \texttt{last}\ x \\
 & & \mid \texttt{let}\ D\ \texttt{in}\ e \mid x(e)\ \texttt{every}\ e \\
 & & \mid e\ \texttt{when}\ C(e) \mid \texttt{merge}\ e\ (C \rightarrow e)\ ...\ (C \rightarrow e) \\
D & ::= & D\ \texttt{and}\ D \mid p = e \mid \texttt{clock}\ x = e \mid \texttt{let}\ D\ \texttt{in}\ D \\
 & & \mid \texttt{match}\ e\ \texttt{with}\ C \rightarrow D\ ...\ C \rightarrow D \\
 & & \mid \texttt{reset}\ D\ \texttt{every}\ e \\
 & & \mid \texttt{automaton}\ S \rightarrow u\ s\ ...\ S \rightarrow u\ s \\
u & ::= & \texttt{let}\ D\ \texttt{in}\ u \mid \texttt{do}\ D\ w \\
s & ::= & \texttt{unless}\ e\ \ \texttt{then}\ S\ s \mid \texttt{unless}\ e\ \ \texttt{continue}\ S\ s \mid \epsilon \\
w & ::= & \texttt{until}\ e\ \ \texttt{then}\ S\ w \mid \texttt{until}\ e\ \ \texttt{continue}\ S\ w \mid \epsilon
\end{array}
$$

Sémantique par traduction

- on enchaine un certain nombre de passes qui éliminent les constructions étendues au fur et a mesure
- doit preserver les types (au sens général)

Les étapes

- compilation du `reset`
- compilation des structures de contrôle en `merge`
- compilation des automates en structures de contrôle (`match/with`)

Traduction

$$
\begin{array}{lcl}
T(\texttt{reset}\ D\ \texttt{every}\ e) & = & \texttt{let}\, x = T(e)\, \texttt{in}\, CReset_x\ T(D) \\
 & & \text{where } x \notin fv(D) \cup fv(e) \\
T(\texttt{match}\ e\ \texttt{with}\ C_1 \to D_1\ ...\ C_n \to D_n) & = & CMatch\ (T(e)) \\
 & & \quad (C_1 \to (T(D_1), Def(D_1))) \\
 & & \quad ... \\
 & & \quad (C_n \to (T(D_n), Def(D_n))) \\
T(\texttt{automaton}\ S_1 \to u_1\ s_1\ ...\ S_n \to u_n\ s_n) & = & CAutomaton \\
 & & \quad (S_1 \to (T_{S_1}(u_1), T_{S_1}(s_1))) \\
 & & \quad ... \\
 & & \quad (S_n \to (T_{S_n}(u_n), T_{S_n}(s_n)))
\end{array}
$$

$$
\begin{array}{lcl}
T_S(\epsilon) & = & \emptyset, S, \texttt{False} \\
T_{S_0}(\texttt{until}\ e\ \ \texttt{then}\ S\ w) & = & (\ x = e\, \texttt{and}\, D, \texttt{if}\ x\ \texttt{then}\ S\ \texttt{else}\ se, \\
 & & \quad \texttt{if}\ x\ \texttt{then True else}\ re) \\
 & & \text{where } x \notin fv(w) \cup fv(e) \text{ and } D, se, re = T_{S_0}(w)
\end{array}
$$

Réinitialisation

$$
\begin{array}{lcl}
CReset_x\ (D_1\ \texttt{and}\ D_2) & = & D'_1\ \texttt{and}\ D'_2 \\
 & & \text{where } D'_1 = CReset_x\ D_1 \\
 & & \text{and } D'_2 = CReset_x\ D_2 \\
CReset_x\ (p = e) & = & p = CResE_x\ e \\
CResE_x\ (y(e)) & = & y(e')\ \texttt{every}\ x \\
 & & \text{where } e' = CResE_x\ e \\
CResE_x\ (y(e_1)\ \texttt{every}\ e_2) & = & y(e'_1)\ \texttt{every}\ x\ \texttt{or}\ e'_2 \\
 & & \text{where } e'_1 = CResE_x\ e_1 \\
 & & \text{and } e'_2 = CResE_x\ e_2 \\
CResE_x\ (e_1\ \texttt{fby}\ e_2) & = & \texttt{let}\ \ y = CResE_x\ e_1\ \texttt{in} \\
 & & \texttt{if}\ \ x\ \texttt{then}\ \ y \\
 & & \texttt{else}\ \ y\ \texttt{fby}\ (CReset_x\ e_2) \\
 & & \text{where } y \notin fv(e_1) \cup fv(e_2)
\end{array}
$$

Conditions d'activation sur des types énumérés

$$
\begin{array}{lcll}
proj_{y}^{C(x)}(D) & = & e & \text{if } (y = e) \in D \\
 & = & (\texttt{pre}\,(y))\ \texttt{when}\ C(x) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
COn\ D\ C(x) & = & D\,[x_1\ \texttt{when}\ C(x)/x_1, ..., \\
 & & \quad x_n\ \texttt{when}\ C(x)/x_n \\
 & & \quad (\texttt{pre}\,(x_1))\ \texttt{when}\ C(x)/\texttt{last}\ x_1, ..., \\
 & & \quad (\texttt{pre}\,(x_n))\ \texttt{when}\ C(x)/\texttt{last}\ x_n] \\
 & & \text{where } \{x_1, ..., x_n\} = fv(D)
\end{array}
$$

$$
\begin{aligned}
&\mathit{CMatch}\ (e)\ (C_1 \to (D_1, N_1))...(C_n \to (D_n, N_n)) = \\
&\qquad\qquad D'_1\ \texttt{and}\ \ldots\ \texttt{and}\ D'_2\ \texttt{and} \\
&\qquad\qquad \texttt{clock}\ x = e\ \texttt{and} \\
&\qquad\qquad y_1 = \texttt{merge}\ x \\
&\qquad\qquad\qquad\qquad (C_1 \to \mathit{proj}_{y_1}^{C_1(x)}(G_1)) \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\qquad (C_n \to \mathit{proj}_{y_1}^{C_n(x)}(G_n)) \\
&\qquad\qquad \texttt{and}\ \ldots\ \texttt{and} \\
&\qquad\qquad y_k = \texttt{merge}\ x \\
&\qquad\qquad\qquad\qquad (C_1 \to \mathit{proj}_{y_k}^{C_1(x)}(G_k)) \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\qquad (C_n \to \mathit{proj}_{y_k}^{C_n(x)}(G_k))
\end{aligned}
$$

where $\forall i, x \notin \mathit{fv}(e) \cup \mathit{fv}(D_i)$ and $\{y_1, ..., y_k\} = N_1 \cup ... \cup N_n$

and $D'_1, G_1 = \mathit{Split}_{N_1}(\mathit{COn}\ D_1\ C_1(x))\ ...$

and $D'_n, G_n = \mathit{Split}_{N_n}(\mathit{COn}\ D_n\ C_n(x))$

Variable et mémoire partagée: `last` et `pre`

```
match x with
  Left -> let cpt = 1 -> last o1 + 1 in
          do o1 = 2 * cpt done
| Right -> do o2 = 1 -> pre o2 + 1
            and o1 = 0 done
end
```

`cpt` est une variable locale alors que `o1` et `o2` sont des variables partagées. `last o1` est une mémoire partagée. Ce code est traduit en:

```
    clock c = x
and cpt = 1 -> ((pre o1) when Left(c)) + 1
and o1 = merge c (Left -> 2 * cpt)
                 (Right -> 0)
and o2 = merge c
           (Left -> (pre o2) when Left(c))
           (Right -> (1 -> pre (o2 when Right(c)) + 1))
```

Automates

- préemption forte et préemption faible au sein d'un même automate
- Un seul etat actif par automate et par cycle
- Une seule transition par cycle?
- on peut accepter un peu plus: enchainer une transition forte et une transition faible seulement
- on peut traduire un automate avec la combinaison `merge`, `when` et `reset`

$$
\begin{array}{l}
\mathit{CAutomaton} \hfill = \\
\quad S_1 \rightarrow (D_1, es_1, er_1)\ (D'_1, es'_1, er_1)\ \ldots\ S_n \rightarrow (D_n, es_n, er_n)\ (D'_n, es'_n, er_n) \\
\qquad \texttt{match}\ pns\ \texttt{with} \\
\qquad\quad S_1 \rightarrow \texttt{reset}\ s = es'_1\ \texttt{and}\ r = er'_1\ \texttt{and}\ D'_1\ \texttt{every}\ pnr \\
\qquad\quad \ldots \\
\qquad\quad S_n \rightarrow \texttt{reset}\ s = es'_n\ \texttt{and}\ r = er'_n\ \texttt{and}\ D'_n\ \texttt{every}\ pnr \\
\qquad \texttt{and}\ \ \texttt{match}\ s\ \texttt{with} \\
\qquad\qquad S_1 \rightarrow \texttt{reset}\ ns = sw_1\ \texttt{and}\ nr = rw_1\ \texttt{and}\ D_1\ \texttt{every}\ r \\
\qquad\qquad \ldots \\
\qquad\qquad S_n \rightarrow \texttt{reset}\ ns = sw_n\ \texttt{and}\ nr = rw_n\ \texttt{and}\ D_n\ \texttt{every}\ r \\
\qquad \texttt{and}\ \ \texttt{clock}\ pns = S_1\ \texttt{fby}\ ns \\
\qquad \texttt{and}\ \ \texttt{clock}\ pnr = \texttt{False}\ \texttt{fby}\ nr \\
\text{where } \forall i. s, ns, r, nr \notin FV(es_i) \cup FV(er_i) \cup FV(D_i) \cup FV(D'_i)
\end{array}
$$

Analyses statiques

- Elles doivent mimer ce que fait la traduction
- Les programmes sources bien typés sont traduits en des programmes bien typés

Typage:

- assez facile
- vérifier l'unicité des définitions
- peut-on écrire `last x` pour n'importe quelle variable?
- Non (en **Lucid Synchrone**): possible pour les variables partagées seulement
- Risque de confusion avec `pre` sinon et principe de substitution

Calcul d'horloge

assez facile sous certaines conditions:

- les variables libres sont toutes s sur la même horloge
- idem pour les variables partagées
- correspond exactement à la sémantique par traduction en `merge`

$$(H\ on_{ck}\ C(c))(x) = H(x)\ \texttt{on}\ C(c) \text{ pourvu que } H(x) = ck$$

Analyse d'initialisation

Plus subtil: profiter des infos de structure des automates

```
let node two x = o where
  automaton
    S1 -> do o = 0 -> last o + 1
          until x continue S2
  | S2 -> do o = last o - 1 until x continue S1
  end
```

o est clairement totalement défini. Cette information est un peu cachée dans le programme traduit.

```
let node two x = o where
   o = merge s
             (S1 -> 0 -> (pre o) when S1(s) + 1)
             (S2 -> (pre o) when S2(s) - 1)
  and
   ns = merge s
            (S1 -> if x when S1(s) then S2 else S1)
            (S2 -> if x when S2(s) then S1 else S2)
  and
```

Le programme suivant n'est pas bien initialisé:

```
let node two x = o where
  automaton
    S1 -> do o = 0 -> last o + 1
          unless x continue S2
  | S2 -> do o = last o - 1
          until x continue S1 end
```

- on peut faire un raisonnement local
- pas d'enchainement de transitions durant une réaction
- calculer les variables partagées nécessairement définies dans la première réaction
- intersection des variables définies dans l'état initial et des variables définies dans les successeurs *strong* de l'état initial
- implanté dans **Lucid Synchrone**

Conclusion

Extensions:

- ajout d'initialisation d'une mémoire partagée:

```
let node f n = o where
  rec last o = n
  and match e with
        true -> do o = last o + 1 done
      | false -> do o = last o - 1 done
```

- signaux et émission