

# Extensions du langage Lustre

julien.forget@cert.fr (ONERA)

December 7, 2007

# Implémentations récentes: SCADE

- Commercialisé par Esterel Technologies;
- Environnement de développement complet: compilateur, outil de vérification, IHM, passerelle Simulink → Scade, etc.
- Génération de code certifié pour logiciels embarqués critiques (norme DO178B);
- Principaux clients : aéronautique (Airbus), défense, transport ferroviaire, automobile;
- Prototype Scade version 6 (été 2007) intégrant les nouvelles fonctionnalités abordées dans ce cours.

## Implémentations récentes: Lucid Sychrone

- Compilateur universitaire développé au LRI, par Marc Pouzet;
- Téléchargeable gratuitement;
- Lustre combiné avec des fonctionnalités à la OCaml (fortement typé, ordre supérieur);
- Construit comme une surcouche d'OCaml;
- Intègre également les fonctionnalités abordées dans cette section du cours.

# Bibliographie

- Scade : [www.esterel-technologies.com](http://www.esterel-technologies.com);
- Lucid Sychrone : [www.lri.fr/~pouzet/lucid-synchrone](http://www.lri.fr/~pouzet/lucid-synchrone);
- Automates par J.L. Colaço et Marc Pouzet: *A Conservative Extension of Synchronous Data-flow with State Machines* (cf page pouzet)
- Iterateurs sur les tableaux par Lionel Morel : *Efficient Compilation of Array Iterators for Lustre* ([www.irisa.fr/espresso/Equipe/Morel/index.php?content=publis](http://www.irisa.fr/espresso/Equipe/Morel/index.php?content=publis))

# Construction merge

Permet de combiner deux flots définis sur des horloges complémentaires :

h	True	False	False	True	True	...
x	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	...
y	$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	...
z=x when h	$x_1$			$x_4$	$x_5$	...
t=y when not h		$y_2$	$y_3$			...
merge(h,z,t)	$x_1$	$y_2$	$y_3$	$x_4$	$x_5$	...

## Construction merge (2)

Différent du **if –then–else** qui porte sur deux flots de même horloge.

Pour obtenir un flot identique à :

```
merge(h, x when h, y when not h);
```

On doit ajouter un **current** dans le **if –then–else** :

```
if h then  
  current(x when h)  
else  
  current(y when not h);
```

# Horloges énumérées

```

type t={v1 , v2 , v3 };
var h:t;
o=x when h match v1;

```

h	v1	v3	v2	v1	v1	...
x	5	2	3	8	9	...
x when h match v1	5			8	9	...

- **x when c**  $\leftrightarrow$  **x when c match true**
- **x when not c**  $\leftrightarrow$  **x when c match false**

## merge sur les énumérés

h	v1	v3	v2	v1	v1	...
x	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	x <sub>4</sub>	x <sub>5</sub>	...
y	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	y <sub>4</sub>	y <sub>5</sub>	...
z	z <sub>1</sub>	z <sub>2</sub>	z <sub>3</sub>	z <sub>4</sub>	z <sub>5</sub>	...
xx=x when h match v1	x <sub>1</sub>			x <sub>4</sub>	x <sub>5</sub>	...
yy=y when h match v2			y <sub>3</sub>			...
zz=z when h match v3		z <sub>2</sub>				...
merge <sub>n</sub> (h, v1 → xx, v2 → yy, v3 → zz)	x <sub>1</sub>	z <sub>2</sub>	y <sub>3</sub>	x <sub>4</sub>	x <sub>5</sub>	...

# Comportement d'un automate simple

```
node N(i:int; c:bool)
  returns (o:int)
  let
    automaton A
      initial state S1
      let
        o=i-1;
      tel
      until c do resume S2;
      state S2
      let
        o=i+1;
      tel
      until c do resume S1;
    tel
```

c	true	false	true	false	...
i	4	3	7	5	...
état	S1	S2	S2	S1	...
o	3	4	8	4	...

# Sémantique d'exécution

A chaque tick :

- L'automate est dans un état  $S$  (état initial au premier tick);
- Le bloc d'équations défini dans l'état  $S$  est évalué;
- Les transitions définies dans l'état  $S$  sont évaluées et déterminent l'état de l'automate pour le tick suivant.

**NB** : Chaque bloc doit définir le même ensemble de variables.

# Imbrications noeuds/automates

- Un noeud peut contenir un nombre arbitraire d'automates;
- Un automate peut être défini à l'intérieur de l'état d'un autre automate (automates hiérarchiques);
- Toute sorte d'équation peut être utilisée dans le corps d'un état  $\Rightarrow$  en particulier des instanciations de noeuds.

# Compilation

Automates compilables en Lustre avec le **merge** et les horloges énumérées :

- Les états de l'automate définissent un type énuméré  $T_{etat}$ ;
- Une variable  $v_{etat}$  codant l'état de l'automate est introduite;
- Cette variable est une horloge énumérée de type  $T_{etat}$ ;
- Les blocs d'équations des états sont rythmés par l'horloge  $v_{etat}$ ;
- La définition complète des flots apparaissant dans les états de l'automate est obtenue en appliquant un **merge** sur ces équations rythmées;
- Les transition sont codées avec des simples **if – then – else** (ou plutôt des **match** pour traiter les énumérés).

# Compilation : un exemple

```

node N(i:int; c:bool)
  returns (o:int)
  let
    automaton A
      initial state S1
      let
        o=i-1;
      tel
      until c do resume S2;
      state S2
      let
        o=i+1;
      tel
      until c do resume S1;
    tel
  tel

```

```

type T_etat = {S1; S2};

node N(i:int; c:bool)
  returns (o:int)
  var etat_suiv:T_etat;
  let
    o=merge(pre(etat_suiv),
            S1->(i-1) when h match S1,
            S2->(i+1) when h match S2);
    etat_suiv=
      S1->match(pre(etat_suiv) with
                S1 -> if c then S2
                    else S1
                S2 -> if c then S1
                    else S2;
  tel

```

# Une extension conservative

Les automates peuvent être compilés avec le langage de base auquel on ajoute les horloges énumérées (le merge est optionnel)  $\Rightarrow$  extension conservative :

- Conserve la sémantique du langage de base;
- La traduction des automates peut être traitée comme une phase amont des compilateurs Lustre traditionnels.

# Les différents types de transitions

- Transitions fortes/faibles :
  - **unless** : on teste d'abord la condition; si elle est vraie on quitte l'état sans évaluer les équations;
  - **until** : on teste la condition après avoir évalué les équations; si elle est vraie on change d'état (pour le tick suivant).
- Arrivée dans un état :
  - **resume** : on revient dans l'état avec les mémoires telles qu'elles étaient au dernier passage dans cet état;
  - **restart** : on réinitialise les mémoires de l'état en entrant.

# Les différents types de transitions : **until/unless**

```
node N(i:int; c:bool)
returns (o:int)
let
  automaton A
    initial state S1
    unless c do resume S2
    let
      o=i+1;
    tel;
    state S2
    let
      o=i-1;
    tel
  until c do resumeS1;
tel
```

c	true	true	false	...
i	6	4	8	...
état	S2	S2	S1	...
o	5	3	9	...

# Les différents types de transitions : resume/restart

```

node N(c:bool)
returns (o:int)
let
  automaton A
    initial state S1
    var cpt_S1:int;
    let
      cpt_S1=0->pre(cpt_S1)+1;
      o=cpt_S1;
    tel
    until c do resume S2;
    state S2
    var cpt_S2;
    let
      cpt_S2=5->pre(ctp_S2)+1;
      o=cpt_S2;
    tel
    until c do restart S1;
  tel

```

c	false	true	false	...
état	S1	S1	S2	...
o	0	1	5	...

c	...	true	true	false	...
état	...	S2	S1	S2	...
o	...	6	0	7	...

## Le **pre** dans les états : problématique

Les équations définies dans un état sont rythmées par la variable d'état

⇒ attention à l'évolution des "mémoires" **pre**. Les mémoires utilisées dans un état sont des mémoires **locales à l'état** !

# Le pre dans les états : exemple

```

node N(c:bool)
  returns (o:int)
  let
    automaton A
      initial state S1
      let
        o=7->pre(o)+1;
      tel
      until c do resume S2;
      state S2
      let
        o=5->pre(o)-1;
      tel
      until c do resume S1;
    tel
  tel

```

c	true	false	true	false	...
S	S1	S2	S2	S1	...
$O_{S1}$	7			8	...
$O_{S2}$		5	4		...
o	7	5	4	8	...

# Le **pre** dans les états : introduction du **last**

```

node N(c:bool)
  returns (o:int)
  let
    automaton A
      initial state S1
      let
        o=7->last(o)+1;
      tel
      until c do resume S2;
      state S2
      let
        o=5->last(o)-1;
      tel
      until c do resume S1;
    tel
  tel

```

c	true	false	true	false	...
S	S1	S2	S2	S1	...
$O_{S1}$	7			6	...
$O_{S2}$		6	5		...
o	7	6	5	6	...

## Le pre dans les états : implémentation du **last**

Soit  $h$  l'horloge d'un automate et  $S1$  un des états de cet automate

- **pre(o)** apparaissant dans  $S1$  est compilé en :  
**pre(o when h match S1);**
- **last(o)** apparaissant dans  $S1$  est compilé en :  
**(pre (o)) when h match S1 .**

## Un exemple complet : le chronomètre

- Deux boutons : Start/Stop et Reset;
- Deux flots pour l'affichage : Minutes et Secondes;
- Le compte du temps débute avec Start et s'arrête avec Stop;
- Quand le chrono est arrêté, Reset le remet à 0;
- Quand le chrono tourne, Reset gèle l'affichage du temps, un nouveau Reset dégèle l'affichage.

# Opérations classiques sur les tableaux

- Accès :  $A[i]$  ( $A$  de taille  $n$ ,  $0 \leq i \leq n - 1$ )
- Constructeurs :  
[0, 3, 2]  
 $\text{true}^3 = [\text{true}, \text{true}, \text{true}]$
- Concaténation :  
 $A|B = [A[0], A[1], \dots, A[n - 1], B[0], B[1], \dots, B[m - 1]]$
- Tranches :

$$A[i..j] = \begin{cases} [A[i], A[i + 1], \dots, A[j]] & \text{si } i \leq j \\ [A[i], A[i - 1], \dots, A[j]] & \text{si } j < i \end{cases}$$

# Limitation des opérations classiques

- Pas de structures de boucles;
- Pas d'opérations applicables point à point aux tableaux (somme de deux tableaux, etc.)
- Programmation sur les tableaux possible mais lourde;
- Peu efficace du point de vue de la compilation : pas de génération de boucles pour les traitements sur les tableaux

# Itérateur *map*

$T2 = \text{map}\langle\langle N, \text{size} \rangle\rangle(T1);$

Donne pour tout  $i, 0 \leq i \leq (\text{size} - 1) : T2[i] = N(T1[i]).$

Exemple, incrémenter toutes les valeurs d'un tableau :

```
const size = 4;
node Incr(i: int) returns (o: int)
let
  o = i + 1;
tel

node IncrTab(T1: int^size) returns (TO: int^size)
let
  TO = map<<Incr, size>>(T1);
tel
```

# Itérateur *red*

```
o=red<<N,size>>(init, T);
```

Signifie :  $N ((\dots, N(N(\text{init}, T[0]), T[1]), \dots), T[N])$

Exemple, calculer la somme des valeurs d'un tableau :

```
const size = 4;  
node Sum(a,b:int) returns (c:int)  
let  
  c=a+b;  
tel  
  
node SumTab(T:int^size) returns (s:int)  
let  
  s=red<<Sum, size >>(0, T);  
tel
```

# Itérateur *map\_red*

acc, TO=**map\_red**<<N,size>>(init, TI);

Combine le comportement du *map* et du *red*.

Exemple, construire la liste des *size* premiers nombres entiers :

```
const size = 4;
node SumDup(a,b:int) returns (c1, c2:int)
  let
    c1=a+b;
    c2=c1;
  tel

node IntList() returns (s:int)
  var dummy:int;
  let
    dummy, s=map_red<<SumDup, size >>(0, 1^size);
  tel
```

# Compilation des itérateurs : un exemple

```
o=red<<N,size>>(init, T);
```

```
_accu_=init;  
for (i=0; i<size; i++) {  
    _accu_=N(_accu_, T[i]);  
}  
o=_accu_;
```

# Un exemple complet : le produit matrice vecteur

- Le produit vectoriel  $PV$  de deux vecteurs de taille  $n$  :

$$PV(u, v) = \sum_{i=0}^n u_i \cdot v_i$$

- Le produit d'une matrice  $M(m, n)$  et d'un vecteur  $v(n)$  est un vecteur  $z$  de taille  $m$ , tel que :

$$\forall i, 1 \leq i \leq m, z_i = PV(M_i, v)$$

- On utilisera l'opérateur *transp* qui permet de transposer deux dimensions d'un tableau :  $transp(1,2, M)$  transpose la première et la deuxième dimension du tableau  $M$ .

# Quelques axes de recherche actuels autour du langage Lustre

- Relâcher l'hypothèse synchrone stricte pour introduire une part d'asynchronisme : systèmes Globally Asynchronous Locally Synchronous (GALS);
- Implanter Lustre sur des architectures distribuées;
- Introduire des concepts temps réel : périodes, latence, etc.