Programmation de systèmes embarqués : l'approche synchrone

Julien FORGET
julien.forget@onera.fr

7 décembre 2007

Plan du cours

1. Introduction
   - Les systèmes embarqués
   - Les systèmes embarqués réactifs
   - L'approche synchrone
2. Le langage synchrone flot de donneés Lustre
3. Le langage synchrone flot de contrôle ESTEREL
4. Compilation de programme synchrone, l'exemple Lustre
5. Extensions récentes des langages synchrones : automates et tableaux
6. Synchrone temps-téel et distribué : la méthodologie Adéquation-Algorithme-Architecture et SYNDEX.

Plan

Définition

Definition

An embedded system is a special-purpose computer system designed to perform one or a few dedicated functions. It is usually embedded as part of a complete device including hardware and mechanical parts. (Wikipedia)

- Industrie légère : téléphones portables, appareils ménagers
- Industrie lourde : aéronautique, aérospatiale, transport ferroviaire, etc.

Caractéristiques principales

- Gestion d'un système physique dans son environement
- Souvent soumis à des contraintes temps réel
  - molles : optimisation du temps de réponse
  - dures : non-respect des échéances = conséquences catastrophiques
- Systèmes dédiés $\Rightarrow$ optimisation forte (coût matériel, consommation électrique, poids, etc)

Systèmes réactifs : définition

Systèmes transformationnels

Lecture des entrées (initialisation) - calculs - production des sorties - terminaison.

Systèmes interactifs

Interaction permanente avec l'environnement. Réaction du système déterminée par les événements actuels (entrées) et passés (état). Temps de réaction optimisé mais non borné.

Systèmes réactifs

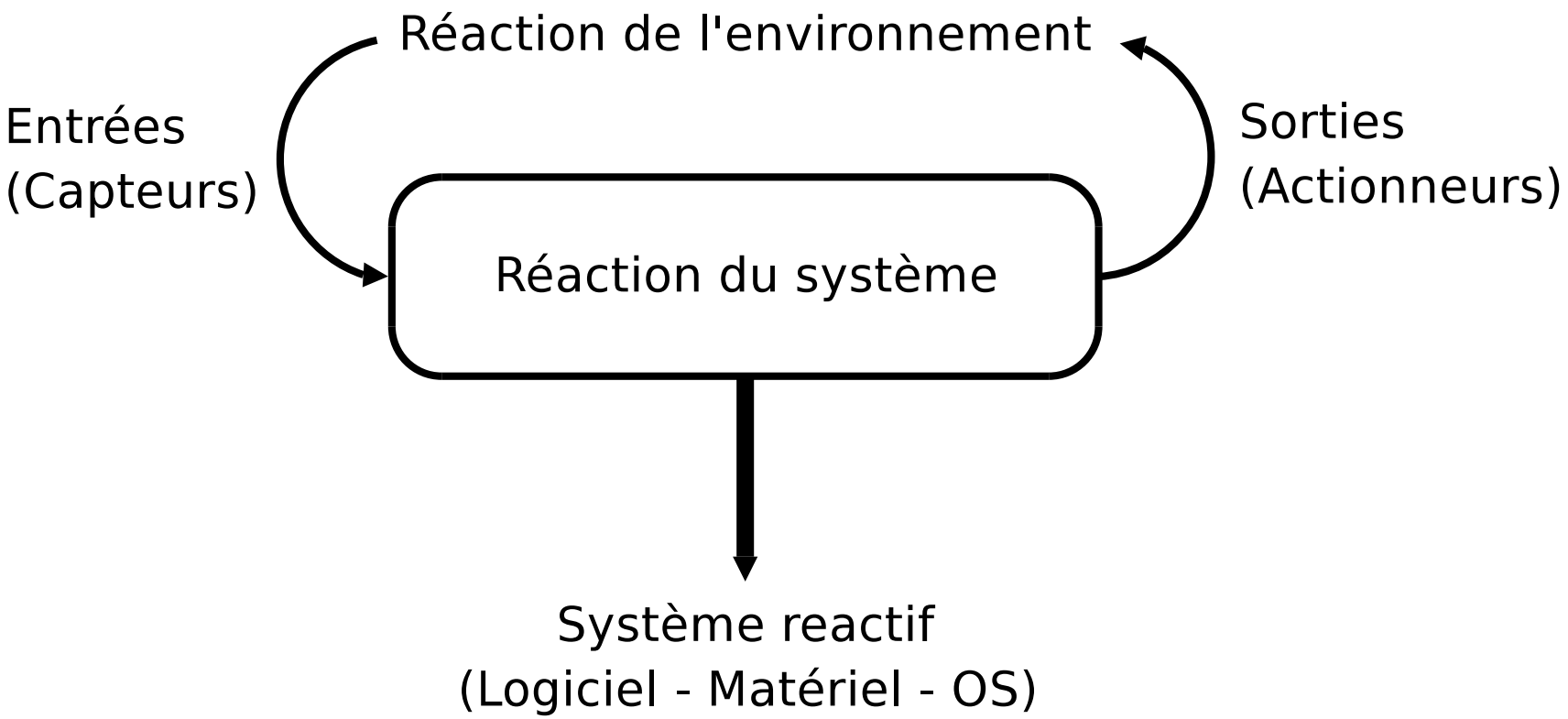
Identiques aux système interactifs, mais temps de réaction borné $\Rightarrow$ rythme du système déterminé par son environnement.

Systèmes réactifs : exemple

Système de contrôle-commande d'un avion

Mission	Stabiliser un avion naturellement instable. Calcul des lois de commandes et asservissement des gouvernes.
Entrées	Etat et assiette de l'avion + ordres pilote.
Sorties	Ordres de gouvernes + alarmes.
Réactivité	Boucle de contrôle de durée 1ms.

Systèmes réactifs : fonctionnement



Problématique

Correction fonctionnelle

Le programme calcule les bonnes sorties.

Correction temporelle

Le programme calcule suffisament rapidement.

On commence par s'intéresser à l'aspect fonctionnel :

- La suite des sorties S_i dépend uniquement de la suite des entrées E_i.
- Le calcul doit se faire en utilisant une mémoire bornée M_i.

Programmation des aspects fonctionnels

Identifier

- Les entrées
- Les sorties

Définir

- La fonction de sortie $S_i = f(E_i, M_i)$
- La fonction de transition (changement d'état) $M_i = g(E_i, M_i)$

Programmer le tout

NB : Vision automatique. En pratique la séparation entre f et g n'est pas forcément si nette.

Difficultés de la programmation temps réel

- Les entrées n'arrivent pas toutes en même temps
  - Quand commencer les traitements ?
  - Comment interpréter le retard d'une entrée : absence ? Délai ?
  - Difficulté de datation des entrées : des entrées mesurées simultanément peuvent parvenir au système réactif avec des dates différentes $\Rightarrow$ comment savoir que ces entrées étaient synchrones ?
- Les temps d'exécution ne sont jamais évaluables très précisément $\Rightarrow$ difficulté pour choisir un ordre entre les traitements (parallélisme potentiel).

Programmation temps réel : l'approche asynchrone

Traitements parallélisables (niveau matériel ou logiciel)
$\Rightarrow$ Implémentation multi-tâches concurrentes

Gestion de l'ordonnancement

Durées d'exécution difficilement prévisibles

Gestion des communications entre tâches

Ordre des communications difficilement maîtrisable (priorités, rendez-vous, sémaphores, etc.)

Globalement indéterministe

Programmation temps réel : l'approche synchrone

On simplifie en prenant une vision abstraite du temps.

- Temps logique = suite d'instants
- On décrit les traitements effectués dans un instant (répétés indéfiniment).
- On décrit les dépendances de données entre les différents traitements :
  - Description fonctionnelle standard.
  - Description temporelle limitée à l'ordre entre les traitements (dépendance de donnée = précédence).

Et c'est tout !

L'hypothèse synchrone

On parle souvent un peu abusivement *d'hypothèse de temps nul*.

- On ne considère pas la durée des traitements.
- Les traitements d'un instant sont "simultanés" (synchrones).
- On ne distingue pas de "début" et de "fin" d'instant.

Validité de l'hypothèse synchrone

Les traitement effectués au cours d'un instant doivent se terminer avant le début de l'instant suivant.

Intégration dans le cycle de développement

1. Ecriture du programme synchrone (formel, forte abstraction)
2. Compilation $\Rightarrow$ génération de code C, Ada, ML (abstraction moyenne)
3. Ecriture du programme d'intégration
   1. Lecture des entrées sur les capteurs
   2. Activation du programme synchrone
   3. Exécution des ordres sur les actionneurs
4. Nouvelle compilation : code synchrone + code intégration (assembleur, bas niveau)
5. OS requis très léger (pas d'ordonnancement, pas de synchronisations)

Qualités

- Formel : sémantique bien définie, possibilité de faire de la preuve formelle, confiance dans le processus de compilation.
- Fort niveau d'abstraction : simplifie la conception.
- Taille mémoire et temps d'exécution borné.
- Pratiquement pas d'OS.

Défauts

- Code produit (parfois) moins efficace qu'un code manuel.
- Difficile à mettre en place sur des architectures distribuées (communications synchrones, synchronisation des différents processeurs).
- Mal adapté aux systèmes multi-rythmes (quel est le rythme de base des instants ?).

Diffusion

R & D

A peu prêt tous les domaines de l'embarqué (téléphonie, automobile, avionique, aérospatiale, transports ferroviaires, centrales nucléaires)

Opérationnel

Surtout les systèmes (très) critiques :

- Avionique : Airbus
- Centrales nucléaires : Schneider Electric, EDF
- Circuits : TI

Parfois utilisé au niveau conception et simulation uniquement (pas de génération de code embarqué).

Acteurs principaux

- Lustre (Verimag) : `www-verimag.imag.fr/SYNCHRONE/`
- Esterel (INRIA) : `www-sop.inria.fr/meije/esterel/esterel-eng.html`
- Esterel Technologies (commecialisation, Scade/Lustre, Esterel) : `www.esterel-technologies.com/`
- Signal (IRISA) : `www.irisa.fr/espresso/source/publications.html`

Mais aussi :

- SynDEx (Synchrone distribué, INRIA) : `www.syndex.org`
- Lucid Synchrone (LRI, Synchrone avec traits à la ML) : `www.lri.fr/ pouzet/lucid-synchrone/`
- ...

Plan

Flots et horloges

- En Lustre, toute expression ou variable est un *flot*
- Flot : suite de valeurs + *horloge*
- Horloge : définit les instants de présence d'un flot
- A chaque instant de présence correspond une valeur de la suite

Example

x	3	4	5	2	6	...
y	True			False	True	...

Extension des opérateur classiques

Les opérateurs classiques sont étendus point à point sur les flots.

Example

c	True	False	True	False	...
x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
x+y	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
if c **then** x **else** y	x_1	y_2	x_3	y_4	...

Opérateur de retard

- L'opérateur pre désigne la valeur précédente d'un flot.
- Sa première valeur est non initialisée.
- Utilisé conjointement avec l'opérateur d'initialisation ->. x−>y vaut x au premier instant, y ensuite.

Example

x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
pre x		x_1	x_2	x_3	...
y−>**pre** x	y_1	x_1	x_2	x_3	...

Echantillonnage

- L'opérateur when sous échantillonne un flot. x **when** c n'est présent que quand *c* vaut vrai et vaut alors *x*
- L'opérateur current remplace les absences introduites par **when** par la dernière valeur présente du flot.

Example

c	True	False	False	True	...
x	x_1	x_2	x_3	x_4	...
y=x **when** c	x_1			x_4	...
current y	x_1	x_1	x_1	x_4	...

current(x **when** c)$\neq$x

Structuration : les nœuds

- Un programme Lustre est un ensemble de nœuds.
- Le nœud principal est spécifié à la compilation.
- Chaque nœud contient un ensemble d'équations définissant les valeurs des flots.
- Les équations ne sont pas ordonnées.
- Les nœuds peuvent être utilisés dans les expressions sur les flots pour définir d'autres flots (définition hiérarchique).

Et puis...

Et puis...

C'est tout !

Et puis...

C'est tout !

Mais il faut s'habituer au style de programmation.

Exemple : un compteur réinitialisable

Code

```
node counter(reset:bool) returns (count:int)
let
  count=0->if reset then 0 else pre(count)+1;
tel
```

Comportement

reset	False	False	False	True	False	...
count	0	1	2	0	1	...

Assignation unique

Ne pas écrire

```
node counter(reset:bool) returns (count:int)
let

   if reset then count=0 else count=0->pre(count)+1;
tel
```

Chaque flot doit avoir une seule définition (similaire aux langages fonctionnels, Lisp, ML).

Double initialisation

Comportement

osc	true	true	false	false	true	...

Double initialisation

Comportement

osc	true	true	false	false	true	...

Code n'ayant pas le comportement attendu

```
node oscil() returns (osc:bool)
let
   osc = true->true->not(pre(pre(osc)))
tel
```

Double initialisation

Comportement

osc	true	true	false	false	true	...

Code corrigé

```
node oscil() returns (osc:bool)
let
  o=true->pre(true->not pre(o));
tel
```

Activation des deux branches du **if then else**

On utilise le compteur défini précédement ainsi qu'un nœud qui compte en décroissant.
decounter(false) : 0,-1,-2,-3,...

Code

```
node count2(reset, c:bool) returns (count:int)
let
  count=0->if c then counter(reset)
                else decounter(reset);
tel
```

Activation des deux branches du **if then else**

On utilise le compteur défini précédement ainsi qu'un nœud qui compte en décroissant.
decounter(false) : 0,-1,-2,-3,...

Code

```
node count2(reset, c:bool) returns (count:int)
let
  count=0->if c then counter(reset)
                else decounter(reset);
tel
```

Comportement

c	True	True	True	False	True	...
count	0	1	2	-3	4	...

Désactiver un traitement : le **when**

Code

```
node count3(reset, c:bool) returns (count:int)
  var c1, c2:int;
let
  c1=current(counter(reset when c));
  c2=current(counter2(reset when not c));
  count=if c then c1 else c2;
tel
```

Désactiver un traitement : le **when**

Code

```
node count3(reset, c:bool) returns (count:int)
  var c1, c2:int;
let
  c1=current(counter(reset when c));
  c2=current(counter2(reset when not c));
  count=if c then c1 else c2;
tel
```

Comportement

c	True	True	True	False	True	...
count	0	1	2	0	3	...

Définitions récursives

Ne pas écrire

```
node error(reset:bool) returns (count:int)
let
   x=y+1; y=x+2;
tel
```

Pas de boucles immédiates !

- Vérifié par l'analyse de causalité.
- Il manque vraisemblablement un **pre**.

Horloges et valeurs indéfinies

- Le sous-échantillonnage (**when**) produit des valeurs indéfinies, qui ne doivent pas être consultées. On dit que le flot est absent.
- On ne combine que des flots de mêmes horloges : x+x **when** c est interdit !

Défaut d'initialisation

x	1	3	5	0	-2	...
c	Faux	Faux	Vrai	Faux	Vrai	...
current (x when c)	nil	nil	5	5	-2	...

Astuce :

- utiliser des horloges initialement vraies : x **when** (true−>c)
- forcer une valeur par défaut

Hiérarchie d'horloges

- L'imbrication des **when** forme un arbre d'horloges dont la racine est l'horloge de base du nœud.
- L'horloge de base d'un nœud est l'horloge de la plus rapide de ses entrées.
- Tandis que le **when** crée une nouvelle horloge (horloge fille) dans l'arbre d'horloges, le **current** permet de remonter d'un niveau et un seul dans l'arbre (horloge mère).
- L'horloge de **current** (x **when** c1 **when** c2) est notée $\hat{x}$ on $c1$, soit l'horloge de x ($\hat{x}$) réduite aux instants où $c1$ vaut vrai.

Plan

Les Signaux

Les éléments d'un programme communiquent en échangeant des signaux

- Présence ou absence
- Signal pur (Présent/Absent)
- Signal valué (Présence+valeur)

Actions associées aux signaux :

- émettre, attendre, tester un signal

Composer des actions :

- Séquence
- Répéter
- Interrompre, etc.

Caractéristiques

- Langage de style impératif (flot de contrôle vs flot de données Lustre).
- Sémantique synchrone (on ne considère pas la durée des instructions, juste leur enchaînement).
- Langage complet très riche ayant évolué avec la demande des clients (circuits).
- Sémantique parfois déroutante (réincarnation, schizophrénie, etc).

On va se concentrer sur le noyau original du langage.

Instructions sur les signaux

Emission

- emit S
- Emet *S* et termine (dans l'instant)

Attente

- await S
- Se met en attente, déblocage sur la prochaine émission future de *S*

Présence d'un signal

present S **then** c1 **else** c2

- Si *S* présent, passe le contrôle à *c*1. Se termine quand *c*1 se termine.
- Si *S* absent, passe le contrôle à *c*2. Se termine quand *c*2 se termine.
- Formes dégénérées :
  - present S **then** c
  - present S **else** c

Test sur l'absence.

Composition d'actions

Séquence

c1 ; c2

- Passe le contrôle à $c1$.
- Quand $c1$ se termine, passe le contrôle à $c2$.
- Termine quand $c2$ se termine.

Boucle

loop c end

- Passe le contrôle à c.
- Quand c se termine, repasse le contrôle à c.

Parallélisme

c1||c2

- Passe le contrôle à $c1$ et à $c2$.
- Se termine quand le dernier des deux se termine.

Plusieurs branches parallèles peuvent émettre le même signal.

- Signal pur présent si émis dans au moins une des branches.
- Signal valué : combinaison des différentes valeurs, +, or, etc. (dangereux)

Les instants ?

Le comportement des signaux peut se représenter de manière similaire aux flots Lustre.

loop await A; emit X end

Entrées	A		A	A		...
Sorties			X	X		...

- Horloge de base (discrète) sur laquelle arrivent les signaux d'entrée.
- Instant : lecture des entrées - exécution des traitements jusqu'à un point de contrôle bloquant - émission des sorties

Un exemple complet

Code

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```

Comportement

Entrées	A		A	A,B		...
Sorties	X		X,Y	X,Y,Z		...

Automates et points de contrôle

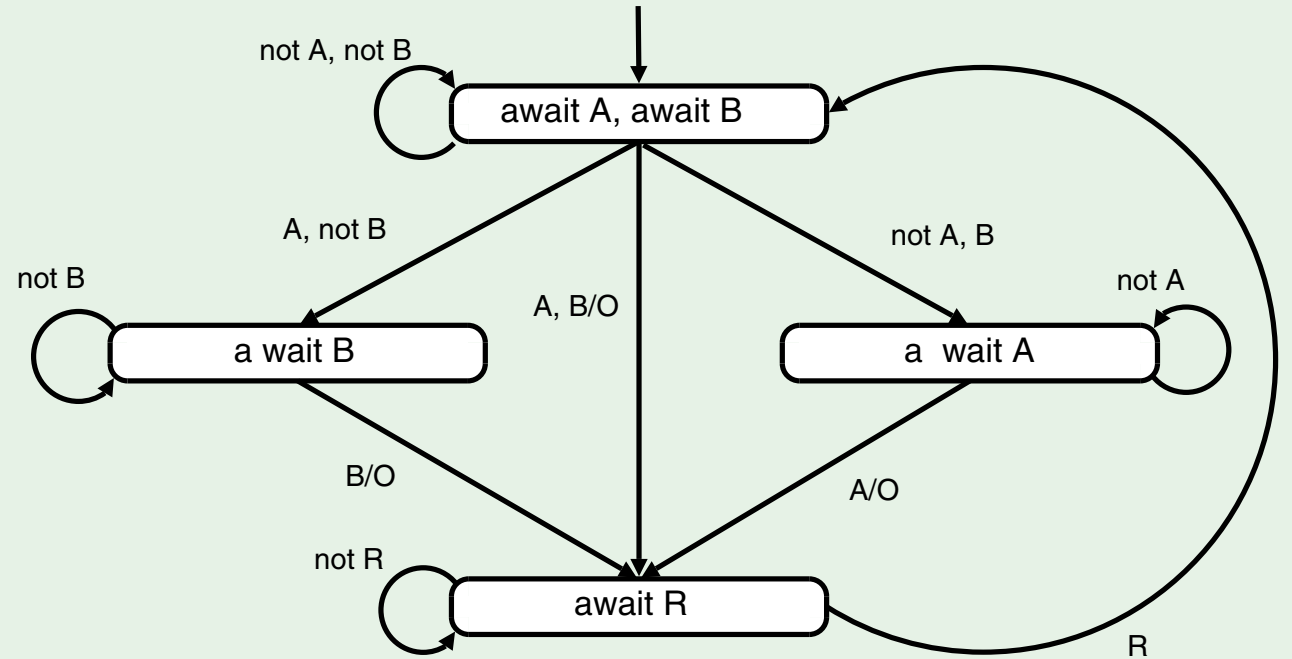
Un code ESTEREL correspond à un automate avec

- Emission de signaux sur les transitions.
- Etat = point de contrôle.

Code

```
loop
   [
      await A
   ||
      await B
   ] ;
   emit O;
   await R
end
```

Automate



Esterel et horloge de base

Vision réactive

- Approche d'origine.
- On ne réagit qu'à la présence de signaux d'entrée.
- L'absence de toute entrée n'est pas un événement.

Vision échantillonnée

- Approche plus récente.
- Les entrées sont consultées avec une certaine fréquence.
- On peut accéder à l'horloge de base, signal tick (toujours vrai).
- On peut réagir en l'absence de toute entrée.

Préemption forte/Préemption faible

abort c **when** x

- Termine le comportement *c* au plus tard à la prochaine occurence future de *x*.
- *c* peut se terminer avant l'occurence de *x*.
- Si *x* est émis, l'interruption est immédiate.

weak abort c **when** x

- Identique mais *c* termine sa réaction courante avant d'être interrompu.

Préemption : exemple

Code

```
      loop
         await S;
(weak)   abort
            await A;
            emit O
         when E
      end
```

Comportement

Entrées		S,A	A	S	A,E	...
Sorties			O		rien/O	...

Préemption et traitement d'exceptions

Possibilité de préciser un traitement sur la condition d'interruption (exception).

Code

```
loop
   await S;
   abort
      await A;
      emit O
   when E do
      emit E;
      await R
   end
end
```

Introduction de nouvelles structures

present X else await X

- Très courant.

$\Rightarrow$ On introduit une nouvelle structure.

- await immediate X

Tendance à l'inflation du langage.

- Justifié du point de vue utilisateur.
- Handicapant pour les outils (compilateur).
- Rend l'apprentissage du langage plus compliqué.

$\Rightarrow$ Compiler vers un petit noyau.

Un noyau ESTEREL

- emit, loop, present, ; et ||
- abort
- pause (intuitivement await tick)
- halt (intuitivement await [not tick])

Traduction vers le noyau

Maintien

- sustain X : émet *X* à chaque instant, ne termine pas (sauf interruption).
- ```
  loop
     emit X ; pause
  end
  ```

Délai strict

- **do** c upto X : exécute c, termine strictement sur le prochain X.
- ```
  abort
     c ; halt
  when X
  ```

Traduction vers le noyau (2)

Echantillonnage

- loop c each X : (re)démarre c à chaque *X*, que *c* ait fini ou pas.
- ```
  loop
     do c upto X
  end
  ```

Echantillonnage avec départ différé

- every X **do** c end : pareil, mais attend un premier *X* pour commencer.
- `await X ; loop c each X`
  ```

Brièvement : les signaux valués

- Présence/Absence + valeur (typée)
- Emission valuée : emit X(3)
- On peut aussi définir des variables auxilliaires : var v :integer
- Et les modifier : v :=v+2

Un chronomètre : spécification

- Version simplifiée.
- Deux boutons : *Start*/*Stop* et *Reset*.
- Une sortie : le temps chronométré.
- Le compte du temps débute avec *Start* et s'arrête avec *Stop*.
- *Reset* remet le temps à 0.

Un chronomètre : programmation

Version basique

```
module BASIC_STOPWATCH :
input START_STOP, CLK;
output TIME(integer);
var t :=0 : integer in
  loop
    emit TIME(t);
    await START_STOP;
    do
      every CLK do
        t:=t+1;
        emit TIME(t)
      end
    upto START_STOP
  end
end
```

Un chronomètre : programmation

Version basique

```
module BASIC_STOPWATCH :
input START_STOP, CLK;
output TIME(integer);
var t :=0 : integer in
  loop
    emit TIME(t);
    await START_STOP;
    do
      every CLK do
        t:=t+1;
        emit TIME(t)
      end
    upto START_STOP
  end
end
```

Avec reset

```
module STOPWATCH_1:
input START_STOP, CLK, RESET;
output TIME(integer)
loop
  run BASIC_STOPWATCH
each RESET
```

Plan

Les analyses statiques du synchrone

Vérifier la correction du programme avant de générer du code.

Analyse de causalité

- Vérifie l'absence de cycle dans les définitions de flots.
- S'apparente à la recherche de cycle dans un graphe.

Analyse d'initialisation

- Vérifie que l'on n'accède pas à des valeurs non initialisées (**pre**).
- Analyse plus récente.

Calcul d'horloges

- Vérifie que l'on ne combine que des flots d'horloges identiques.

$\Rightarrow$ Pas d'accès à des valeurs de flots absents.

L'approche par inférence d'horloge

- Idée : reprendre les techniques d'inférence de types (classique en ML)
- Calcul d'horloge = système de types.
- Système de types = ensemble de règles d'inférence.

Règles d'inférence

Type d'une constante

$$\frac{c \in dom(E)}{E \vdash c : E(c)}$$

- E est un environnement associant un type (ou une horloge) à une expression.
- $E \vdash x : t$ est un jugement de type déclarant que x a le type t dans l'environnement E.
- Une règle d'inférence dit que si les prémisses de la règle (partie haute) sont satisfaites alors on peut en déduire que la conclusion (partie basse) est vraie.
- Ci-dessus on déclare qu'une constante c a pour type $E(c)$ si elle fait partie du domaine de E.
- On dit ainsi que c est bien typée si elle est liée (déclarée) dans E.

Type fonctionnel

Type de l'addition

$$E \vdash + : int \rightarrow int \rightarrow int$$

- $\rightarrow$ désigne un type fonctionnel.
- On déclare que $+$ est une fonction, qui s'applique à deux entiers et renvoie un entier.
- On déclare aussi que l'application de $+$ à un seul entier produit une nouvelle fonction, qui s'applique cette fois à un seul entier (ex : $(3+)4$).
- En pratique cette règle rentre dans la règle sur les constantes vue précédement (constante fonctionnelle).

Type polymorphe

La fonction identité

$$\textit{identité} : \forall \alpha, \alpha \rightarrow \alpha$$

- On déclare que la fonction identité prend un paramètre de type α et renvoie une valeur de type α.
- Et ce, quel que soit α.
- Ainsi *identité* s'applique aussi bien à des entiers, qu'à des chaînes de caractères et même à des valeurs fonctionnelles.
- Par contre on renvoie à chaque fois une valeur du même type que le type du paramètre.

Règle d'inférence d'horloges

Horloge de l'addition

$$H \vdash + : \forall \alpha, \alpha \rightarrow \alpha \rightarrow \alpha$$

- L'addition prend deux flots de même horloge.
- Cette horloge est quelconque.
- Si les flots ne sont pas de même horloge, ils sont mal synchronisés, la règle ne peut être appliquée.

Définition de fonction

Règle d'inférence

$$\frac{H, x : cl_1 \vdash e : cl_2}{H \vdash \lambda x.e : cl_1 \rightarrow cl_2}$$

- $\lambda x.e$ est une fonction qui associe la valeur e au paramètre x (Ex : $\lambda x.x + 3$).
- Pour calculer l'horloge de cette expression, on rajoute x dans l'environnement avec son horloge.
- Intuitivement, cela revient à déclarer les paramètres de la fonction dans l'environnement.
- Dans cet environnement enrichi, on calcule ensuite l'horloge de e.
- L'horloge obtenue pour e est l'horloge de retour de la fonction.
- L'expression résultante a une horloge de type fonctionnel.

Application de fonction

Règle d'inférence

$$\frac{H \vdash f : cl_1 \rightarrow cl_2 \qquad H \vdash x : cl_1}{H \vdash f\ x : cl_2}$$

- $f\ x$ est l'application de la fonction f à x.
- Pour que $f\ x$ soit bien synchronisé, il faut que :
  - f soit une fonction (prémisse gauche).
  - x ait l'horloge du paramètre attendu par f (prémisse droite).
- $f\ x$ a alors pour horloge l'horloge de la valeur retournée par f.
- Exemple d'application : $(\lambda x.x + 3)\ 5$, résultat 8.

Le **when**

Règle d'inférence

$$\texttt{when} \ : \forall \alpha, \forall X, \alpha \rightarrow (X : \alpha) \rightarrow \alpha \ \texttt{on} \ X$$

- Le premier paramètre de **when** est sur une horloge quelconque α.
- Le second est une condition, représentée par X, ayant elle aussi pour horloge α.
- Le résultat (x when c) est sur une nouvelle horloge, l'horloge α restreinte aux instants où la condition X est vraie.
- L'horloge α peut elle-même être de la forme β on y.
- On peut ainsi imbriquer les niveaux d'horloges.

Le **current**

Règle d'inférence

$$\texttt{current} : \forall \alpha, \forall X, \alpha \text{ on } X \rightarrow \alpha$$

- L'inverse du **when**, du point de vue des horloges.
- Le paramètre de **current** est sur une l'horloge α restreinte aux instants où une condition X est vraie.
- Le résultat est sur l'horloge α.
- L'horloge α peut elle-même être de la forme β on y.
- On ne "supprime" qu'un niveau d'horloge, on ne revient pas directement à l'horloge de base.

Preuve de synchronisation

- L'ensemble des règles forme un système de preuves.
- On utilise ce système pour prouver qu'une expression complexe est bien synchronisée.
- On calcule en même temps son horloge.
- Si une expression n'admet pas de preuve dans le système de type, alors elle est mal synchronisée, rejetée (poliment) par le compilateur.

Exemple de preuve de synchronisation

Programme à inférer

```
node N (c: bool, a: int when c, b)
  returns (o:int when c)
let
  o=(a+2)*(b when c);
tel
```

- La déclaration du nœud contraint déjà beaucoup la preuve de départ.
- Il suffit de prouver que l'équation est cohérente avec la déclaration du nœud.

Exemple de preuve de synchronisation (2)

$$\dfrac{(\text{APP})\ \dfrac{H \vdash a : \alpha \text{ on } c \qquad H \vdash 2 : \alpha \text{ on } c}{H \vdash a + 2 : \alpha \text{ on } c} \qquad (\text{WHEN})\ \dfrac{H \vdash b : \alpha \quad H \vdash c : (c : \alpha)}{H \vdash b \text{ when } c : \alpha \text{ on } c}}{H \vdash (a + 2) * (b \text{ when } c) : \alpha \text{ on } c}\ (\text{APP})$$

$$H \vdash o : \alpha \text{ on } c$$

Preuve de synchronisation sans annotations d'horloges

Programme à inférer

```
node N (c, a) returns (o)
let
  o=(a+2)*(b when c)
tel
```

On arrive exactement au même résultat !

Complétude de l'inférence d'horloges

- Unification d'horloges par nom : deux horloges sont égales (unifiables) si elles ont le même nom.
- Bien entendu, si $c_1 = c_2$ et $c_2 = c_3$:
  - c_1 et c_3 sont unifiables.
  - c on c_1 et c on c_3 sont unifiables (unification récursive).
- Par contre, c on $(a \; and \; b)$ et c on $(a \; and \; d)$ ne sont pas unifiables, que les conditions b et d soient des conditions identiques ou pas.
- On ne fait pas de calcul d'équivalence sur les expressions booléennes (NP-complet).

Structure générale d'un programme réactif

Structure

```
Systeme(E,S)
  memoire M
  M:=M0
  a chaque periode faire
    lire(E)
    S=f(M,E)
    M=g(M,E)
    ecrire(S)
  fin
```

Le compilateur doit :

- Identifier la mémoire M à allouer.
- Calculer les valeurs initiales de $M0$.
- Fournir le code du corps de la boucle effectué à chaque période.
- En essayant d'optimiser le tout.

Compilation en boucle simple

- Simple : un nœud = une procédure (C).
- Force une analyse de causalité modulaire $\Rightarrow$ certains programmes corrects sont rejetés.
- Possibilité d'expanser certains nœuds à la demande pour régler ce problème.
- Intérêt : le code produit est compréhensible (solution SCADE).
- Autre solution : compilation en automates (solution Lustre). Plus puissant, mais compilation plus complexe, code moins lisible.

$\Rightarrow$ On va s'intéresser à la compilation en boucle simple.

Principes de compilation (boucle simple)

Traduction Lustre→ C

Code Lustre	Code C
Définition de flot	Affectation
Opérateur point-à-point (+, and)	Opérateur standard
–>	Mémoire initiale
pre	Valeur mémorisée
when	**if**

+ Séquentialiser le système d'équations.

⇒ Une séquence existe car le programme est causal.

Principes de compilation (2)

Plus précisément :

- **pre** x : on alloue une variable de plus (px).
- Un booléen *init* détermine si on est en phase d'initialisation.

$\Rightarrow$ $x->y$: **if** init **then** x **else** y

Exemple

Lustre

```
node counter(reset:bool)
  returns (cpt:int)
let
  cpt=0->if reset then 0
           else
             pre(cpt)+1;
tel
```

C

```
int cpt;
bool reset;
int pcpt;
bool init=true;
void counter_step()
{
  cpt=if (init) 0
        else if (reset) 0
             else pcpt+1;
  pcpt=cpt;
  init=false;
}
```

Optimisations

On n'a présenté que la base, des optimisations sont possibles :

- Regroupement des instructions effectuées sous une même condition (un seul **if**).
- Réutiliser des variables pour réduire la mémoire utilisée (durée de vie d'une variable).
- Elimination des variables locales inutiles (ex a=b; b=c;).
- ...