Programmation Synchrone Fonctionnelle

Marc Pouzet
LRI

Marc.Pouzet@lri.fr

AFADL, 16/03/06

Systèmes embarqués temps-réel: caractéristiques

Contraintes temporelles fortes

- le système est soumis aux contraintes de son environnement
- **à la vitesse imposée par celui-ci** (la physique n'attend pas!)

$\hookrightarrow$ temps de réponse et mémoire statiquement bornés

Environnement hétérogène

- environnement physique **continu**: capteur de température, actionneurs
- et/ou **discret**: boutons, seuils, autres calculateurs

$\hookrightarrow$ le formalisme doit permettre de décrire cette hétérogénéité

Systèmes concurrents et déterministes

- systèmes **en boucle fermée** (closed loop): le controleur de chaudière et la chaudière fonctionnent en parallèle
- la concurrence est le moyen naturel de **composer des systèmes**: *contrôler en même temps roulis et tangage*
- le système doit rester **déterministe**
- aussi **plus simple**: reproductibilité, facilite la mise au point, la simulation

$\hookrightarrow$ le formalisme doit permettre de composer les systèmes en parallèle

$\hookrightarrow$ modèle de description conciliant concurrence et déterminisme

La sûreté est importante

- systèmes **critiques**: pilotage, freinage, airbag, appareils médicaux
- certains systèmes n'ont pas de position stable de replis (avion?)

$\hookrightarrow$ propriétés garanties statiquement: “dynamique” = “trop tard”

$\hookrightarrow$ langages ayant une sémantique bien fondée, vérification formelle

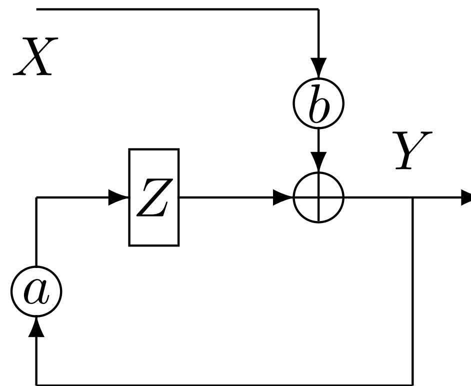
Concevoir des langages dédiés

- Le modèle logiciel classique est inadapté: Turing complet, trop expressif, trop dur à vérifier
- la complexité n'est parfois pas la où on l'attend: arithmétique de pointeurs, allocation dynamique, etc.
- a t'on besoin de tout ça?
- le parallélisme et le déterminisme sont absents alors qu'ils sont fondamentaux!
- inventer des langages spécifiques ayant un pouvoir expressif limité, une sémantique formelle, adaptés à la culture du domaine
- cette culture est multiple:
  - contrôle continu (théorie du controle, automatique)
  - contrôle discret (théorie des automates)

Controle continu (Automatique, traitement du signal...)

- un signal/évenement est représenté par une suite de valeurs (un flot)
  $\hookrightarrow$ *équations de suites, fonctions génératrices, transformée en z*
  $\hookrightarrow$ *formalismes graphiques pour décrire des réseaux d'opérateurs*
- transcription manuelle de ces équations en programmes impératifs
- pénible, source d'erreur

$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$

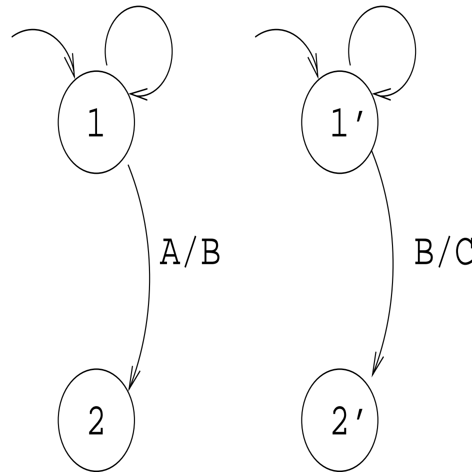


Idée de Lustre (et Signal) (1984):

- programmer directement en écrivant des équations de suites
- fournir un compilateur et des outils d'analyse

Contrôle discret

- Systèmes de transitions (automates), Machines de Mealy/Moore
- composition synchrone d'automates + encapsulation, hiérarchie, etc.
- calculs de processus (SCCS de Milner)

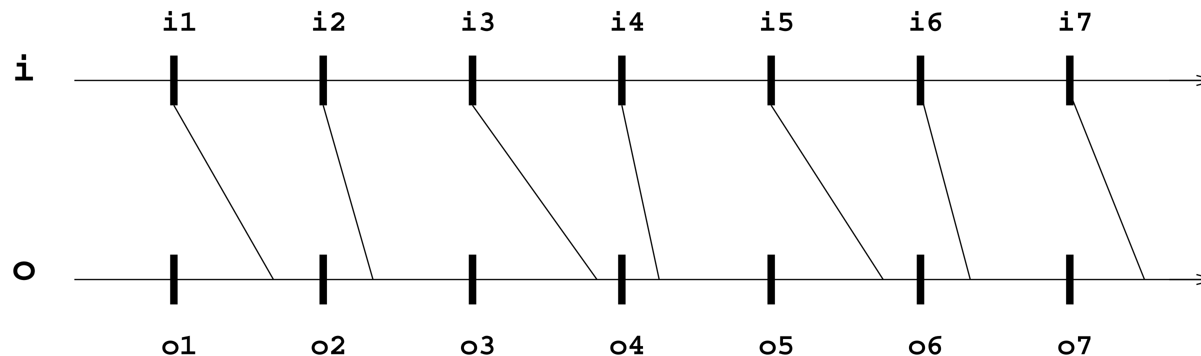


Esterel (82):

- proposer un langage de haut niveau de description de systèmes concurrents séquentiels
- basé sur la composition synchrone de processus
- préservant le déterminisme

Le modèle du temps synchrone

- ces langages sont fondés sur le modèle dit **zéro délai**
  - le temps est **logique** vu comme la séquence de réactions atomiques du système aux événements d'entrée
  - le système est la **composition parallèle** de sous-systèmes s'exécutant (virtuellement) en parallèle
  - vérifier la **correspondance** entre le temps logique et le temps physique: la machine est-elle suffisamment rapide?



temps de réaction maximum $max_{n \in IN}(t_n - t_{n-1}) \leq bound$

Langages synchrones

- fondés sur un modèle commun mais avec des styles différents
- impératif (automates): **Esterel**, **SyncCharts**, **Argos**
- déclaratif (dataflow): **Lustre**, **Signal**
- compilateur (et environnement) industriel Scade/Lustre, Esterel-studio (Esterel-Technologies), Signal/Sildex (TNI)
- sémantique formelle, compilation vers hardware ou software d'un même code, outils de test et vérification automatique
- répartion automatique de code (systèmes distribués), taches asynchrones (OS temps-réel), etc.
- succès industriel important: avionique (Airbus, Dassault, Eurocopter), transport (Matra, Audi), circuits (Xilink, TI, Intel)

Mais aussi...

- outils de simulation de systèmes: Simulink/StateFlow (The MathWorks), etc.
- plateforme très riche pour la simulation du système **et** de son environnement
- fondés sur des techniques d'analyse numérique, de simulation
- générateurs de code (partiels), outils de vérification, etc.
- ces outils ne sont pas si éloignés de ceux du synchrone
- une description proche des planches Scade (avec Simulink) ou des SyncCharts (avec StateFlow) mais...
- ils n'ont pas été conçus dans une optique de programmation (où ce que l'on modélise est exactement ce qui s'exécute)
- sémantique imprécise (code certifié ou de bonne qualité?), outils formels de preuve/vérif?
- ce qui est simulé doit être ce qui est exécuté!

Exemple: usine logicielle Catia + LCM (Dassault-Systèmes)

Besoin d'extension des outils du synchrone

- maîtriser la complexité et le passage à l'échelle
  - que faire de tous ces transistors?
  - les systèmes les plus critiques sont devenus gros: 500 000 lignes de code pour les commandes de vol de l'A380
  - certains fabricants ne font que spécifier puis assembler le code fait par d'autres
- modularité (librairies), mécanismes d'abstraction
- outils de type "langage" (vs vérification) garantissant des propriétés à la compilation: inférence de type et d'horloge (**obligatoire** dans un outil graphique), absence de blocage, etc.
- comment décrire du dataflow (e.g., Lustre) et du control-flow (e.g., Esterel) dans un cadre unifié?
- liens avec les outils de preuve formelle (code certifiée, norme DO 178B)

Les origines de Lucid Synchrone

En 95, avec Paul Caspi (VERIMAG)

Quelles sont les relations entre:

- réseaux de Kahn
- programmation synchrone data-flow (e.g., Lustre)
- outils/modèles de l'automatique/traitement du signal
- programmation fonctionnelle paresseuse (e.g., Haskell)
- types et horloges
- machines a état et fonctions de suites

Que peut-on apprendre du rapprochement entre le synchrone et la programmation fonctionnelle?

Observations

- synthèse automatique des types: ça existe dans Scade (et Simulink)
- synthèse automatique des horloges: idem
- polymorphisme (utile pour écrire des librairies): idem
- mais tout ça est un peu ad-hoc
- compilation modulaire: peut-on l'expliquer simplement?
- ordre supérieur (écrire un noeud paramétré par un autre): le faire à la main

Questions

- utiliser des techniques conventionnelles de typage?
- types et horloges (suréchantillonnage, synthèse)?
- comment expliquer (simplement) la compilation modulaire?
- peut-on avoir un mécanisme propre d'abstraction générale (fonctions curryfiées, ordre supérieur)?
- définir un ensemble d'analyses de programmes modulaires?
- qu'est ce qui est spécifique la dedans?

Lucid Synchrone

Un langage laboratoire

- étudier des extensions de Lustre (synchrone fonctionnel)
- expérimenter jusqu'au bout et écrire des programmes!
- Version 1 (1995), Version 2 (2001), V3 (2006)

Sémantique

- Réseaux de Kahn synchrones [ICFP'96]
- Calcul d'horloge = types dépendants [ICFP'96]
- fonctions de suites synchrones et fonctions de transition (co-induction *vs* co-itération) [CMCS'98]
- Calcul d'horloge à la ML [Emsoft'03]
- analyse de causalité [ESOP'01]
- analyse d'initialisation [SLAP'03, STTT'04]
- Ordre supérieur général et typage [Emsoft'04]
- Flot de données et machines à état [Emsoft'05]
- Réseaux de Kahn N-Synchrone [Emsoft'05, POPL'06]

Quelques exemples (V3)

- `int` désigne le type des flots d'entiers,
- `1` désigne le flot (infini) constant de valeurs 1,
- les primitives usuelles sont appliquées point-à-point

`c`	t	f	t	$\ldots$
`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`if c then x else y`	x_0	y_1	x_2	$\ldots$

Fonctions combinatoires

Exemple: additionneur 1-bit

```
let xor x y = (x & not (y)) or (not x & y)

let full_add(a, b, c) = (s, co)
  where
      s = (a xor b) xor c
  and co = (a & b) or (b & c) or (a & c)
```

Le compilateur synthétise automatiquement la signature de type et d'horloge.

```
val full_add :  bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
```

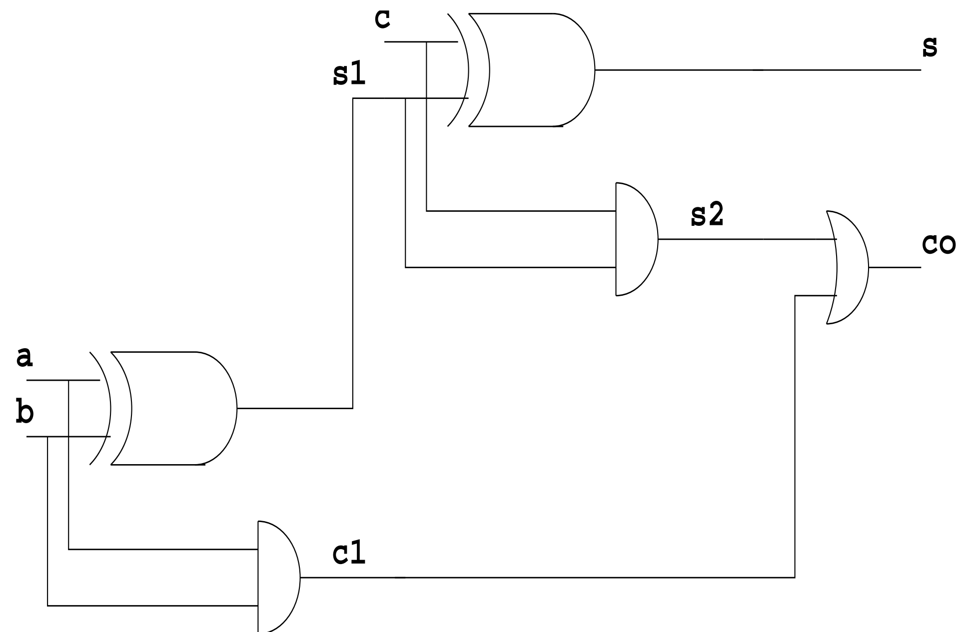
Full Adder (hiérarchique)

Composer deux “half adder”

```
let half_add(a,b) = (s, co)
   where
       s = a xor b
   and co = a & b
```

L’instancier deux fois

```
let full_add(a,b,c) = (s, co)
  where
      (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c1 or c2
```



Fonctions séquentielles

Opérateurs `fby`, `->`, `pre`

- `fby`: délai (retard) unitaire initialisé
- `->`: opérateur d'initialisation
- `pre`: délai non initialisé (registre)

`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`x fby y`	x_0	y_0	y_1	$\ldots$
`pre x`	nil	x_0	x_1	$\ldots$
`x -> y`	x_0	y_1	y_2	$\ldots$

Fonctions séquentielles

- Les fonctions peuvent dépendre du passé (système à état)
- Exemple: détection d'un front montant

```
let node edge x = x -> not (pre x) & x

val sum :  int => int
val sum :: 'a -> 'a
```

x	t	f	t	t	t	f	$\dots$
edge x	t	f	t	f	f	f	$\dots$

Dans la V3, on distingue les fonctions combinatoires (`->`) des fonctions séquentielles (`=>`)

Polymorphisme (réutilisation de code)

```
let node delay x = x -> pre x

val delay :  'a => 'a
val delay :: 'a -> 'a

let node edge x = false -> x <> pre x

val edge :  'a => 'a
val edge :: 'a -> 'a
```

En Lustre, le polymorphisme est limité à un jeu d'opérateurs prédéfini (e.g, `if/then/else`, `when`) et ne passe pas l'abstraction.

Librairie et Curryfication

```
(* module Numerical *)
let node integr dt x0 dx = x where
  rec x = x0 -> pre x +. dx *. dt

val integr :  float -> float -> float => float
val integr :: 'a -> 'a -> 'a -> 'a



(* module Main *)
let static dt = 0.001

let integr = integr dt

val integr :  float -> float => float
val integr :: 'a -> 'a -> 'a
```

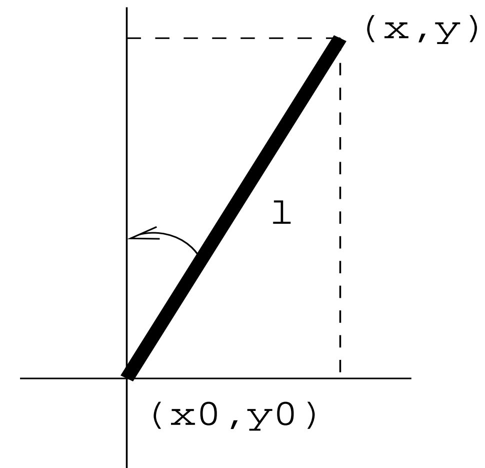
Exemple: le pendule inversé

Spécification: controler un pendule inversé

$$l * \frac{d^2\theta}{dt^2} = sin(\theta) * (\frac{d^2 y_0}{dt^2} + g) - (cos(\theta) * \frac{d^2 x_0}{dt^2})$$

$$x = x_0 + l.sin(\theta)$$

$$y = y_0 + l.cos(\theta)$$



Module principal:

Constantes:

```
let static dt = 0.001 (* pas d'echantillonnage *)
and static l = 10.0   (* longueur   *)
and static g = 9.81   (* acceleration *)

(* application partielle fixant le pas *)
let integr = Numerical.integr dt
let deriv = Numerical.deriv dt
```

L'équation du pendule

```
let node pendulum d2x0 d2y0 = theta where
  rec theta = integr (integr ((sin thetap)*.(d2y0 +. g)
                             -.(cos thetap)*.d2x0)/.l)
  and thetap = 0.0 -> pre theta
```

Rejeter des programmes

Rejeter des programmes qui ne peuvent être exécutés séquentiellement

```
let node pendul d2x0 d2y0 = theta
  where rec theta =
        integr (integr ((sin theta)*.(d2y0 +. g)
                                 ^^^^^^
                              -.(cos theta)*.d2x0)/.l)
thetap depends instantaneously on itself
```

- un critère syntaxique: récursion à travers un retard
- un système de types (avec rangées), avec Pascal Cuoq [ESOP'01]
- donc, des signatures (interfaces)
- modulaire et ordre supérieur

Rejeter des programmes

Rejeter les programmes dont le résultat dépend de retards non initialisés

```
let node pendul d2x0 d2y0 = theta
  where rec theta =
        integr (integr ((sin (pre theta)*.(d2y0 +. g)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                          -.(cos (pre theta))*.d2x0)/.l)
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
this expression may not be initialized
```

- abstraction 1-bit pertinente
- un système de types (avec sous-typage), avec JL-Colaço [SLAP'02, STTT'04]
- ça marche très bien pour Scade (discipline de programmation)
- testé sur des exemples industriels (75000 lignes) chez Esterel Tech.

Horloges: plusieurs échelles de temps

- mélanger des processus lents avec des processus rapides de manière sure?
- systèmes multi-échantillonnés (logiciel), multi-horloges (hardware)
- introduit dans Lustre et Signal dès l'origine
- présent dans Simulink aussi (systèmes périodiques échantillonnés)

En **Lucid Synchrone**, une horloge est un type et est synthétisée automatiquement

Deux opérateurs

`when` (sous-échantillonnage) et `merge` (sur-échantillonnage)

`c`	t	t	f	f	t	f	$\ldots$
`x`	x_0	x_1	x_2	x_3	x_4	x_5	$\ldots$
`x when c`	x_0	x_1			x_4		$\ldots$
`x whenot c`			x_2	x_3		x_5	$\ldots$
`y`	y_0	y_1			y_2		$\ldots$
`merge c y (x whenot c)`	y_0	y_1	x_2	x_3	y_2	x_5	$\ldots$

Exemple d'échantillonnage

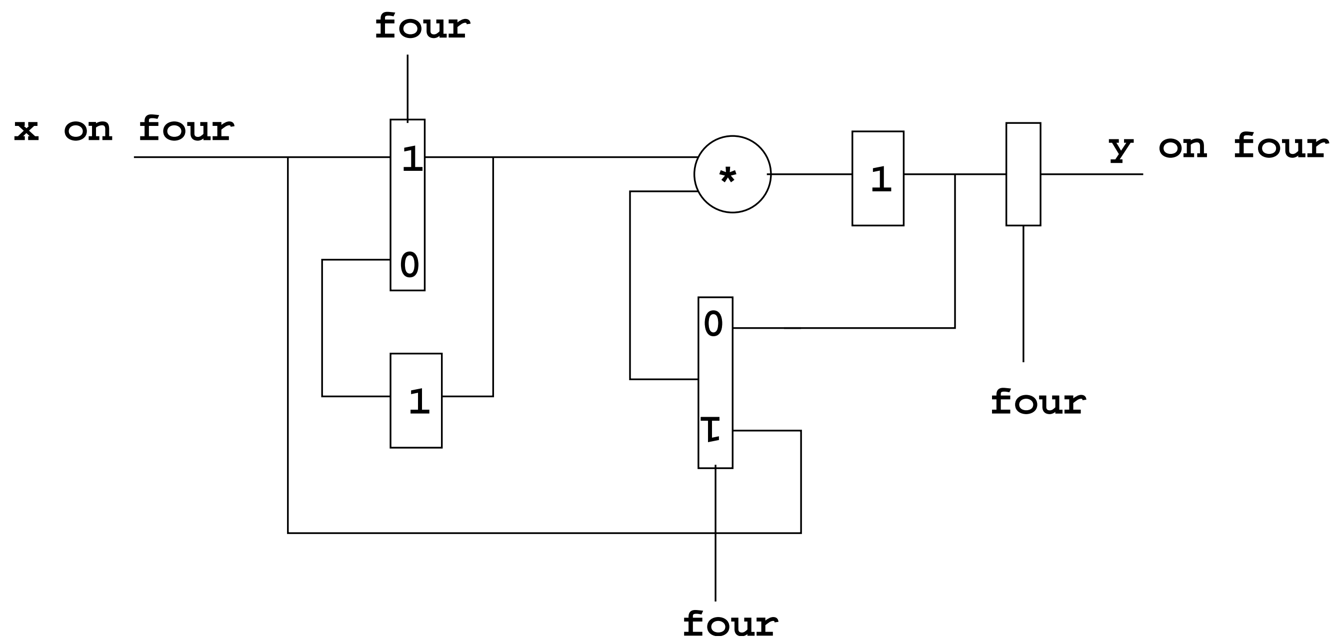
```
let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum :  int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0

let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten

val ten :  bool
val ten :: 'a
val sum_ten :  int => int
val sum_ten :: 'a -> 'a on ten
```

Sur-échantillonnage

- Définir des systèmes qui ont un rythme plus rapide à l'intérieur qu'à l'extérieur?
- exprimer des contraintes temporelles, de scheduling, de ressources

Exemple: Calcul de `x^5`

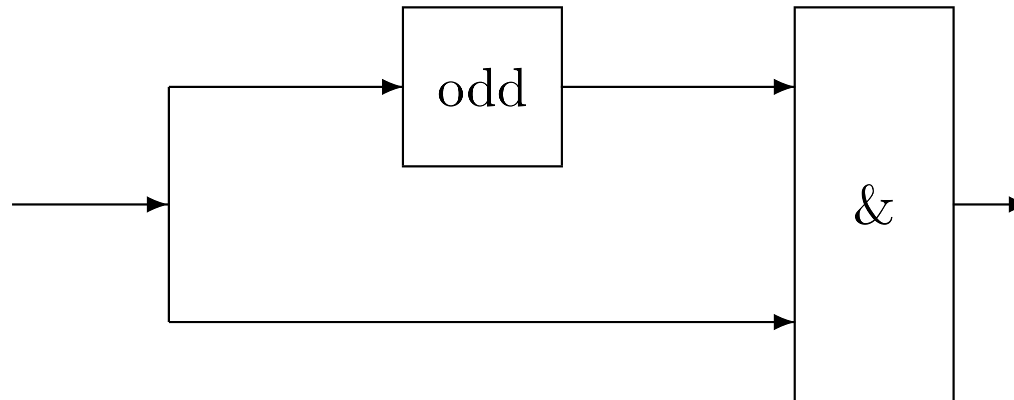
```
let node power x = x * x * x * x * x


let clock four = count 4 true
let node spower x = y where
  rec i = merge four x ((1 fby i) whenot four)
  and o = 1 fby (i * merge four x (o whenot four))
  and y = o when four

val power  :: 'a -> 'a
val spower :: 'a on four -> 'a on four
```



four	t	f	f	f	t	f	f	f	t	f	f	...
x	x_0				x_1				x_2			...
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2	x_2	x_2	...
o	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5	x_2^2	x_2^3	...
spower x	1				x_0^5				x_1^5			...
power x	x_0^5				x_1^5				x_2^5			...

Propriété: `1 fby (power x)` et `spower x` sont observationnellement équivalents

Contraintes d'horloges et synchronisme



Le calcul de $(x_n \,\&\, x_{2n})_{n \in IN}$ n'est pas temps-réel

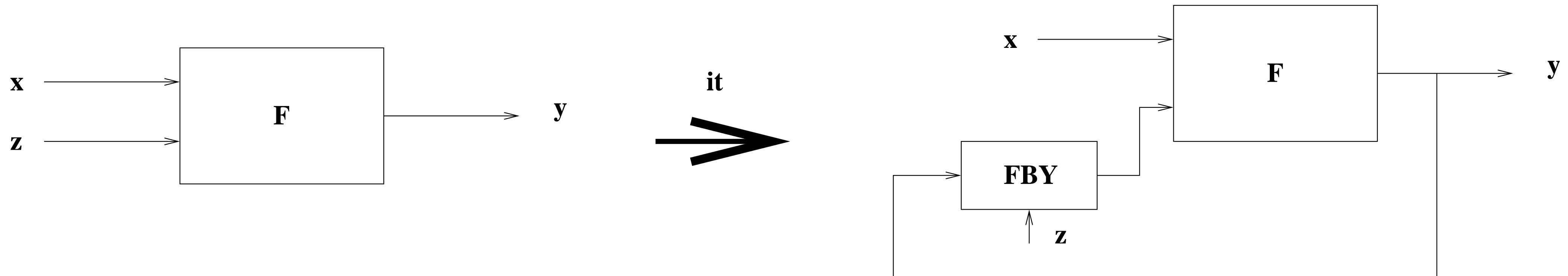
```
let odd x = x when half
let non_synchronous x =  x & (odd  x)
                             ^^^^^^^^^

This expression has clock 'a on half, but is used with clock 'a.
```

Exécution à buffers non bornés!!!

- les horloges = une information sur le comportement des flots
- horloges = types
- reprise du `merge` et du calcul d'horloge par typage dans le compilateur ReLuC de Scade

Un peu d'ordre supérieur

Rebouclage (itérateur):




```
let node it f z x = y
    where rec y = f x (init fby y)
```

```
val it :  ('b -> 'a -> 'a) -> 'a -> 'b => 'a
val it ::  ('b -> 'a -> 'a) -> 'a -> 'b -> 'a
```

Exemple:

```
let node sum x = it (+) 0 x
let node mult x = it (*) 1 x
```

Systèmes mixtes (dataflow + automates)

Systèmes dominés "données": systèmes continus échantillonnés, formalismes schéma/bloc

$\hookrightarrow$ Outils de simulation: Simulink, etc.

$\hookrightarrow$ Langages de programmation: Scade/Lustre, Signal, etc.

Systèmes dominés "contrôle": systèmes de transition, systèmes à événements, formalismes par automates

$\hookrightarrow$ StateFlow, StateCharts

$\hookrightarrow$ SyncCharts, Argos, Esterel, etc.

Quid des systèmes mixtes?

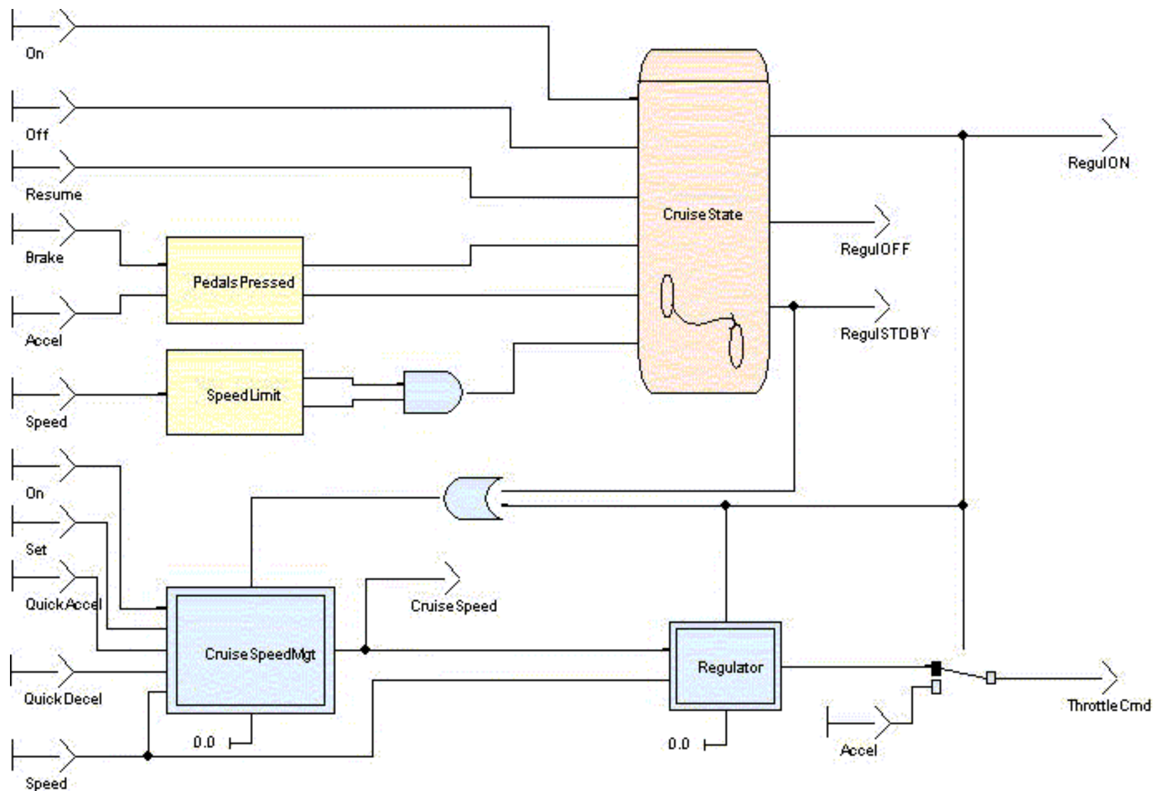
- les systèmes réels mélangent les deux: les systèmes ont des **"modes"**
- chaque mode est défini par une loi de contrôle, naturellement décrite par des équations data-flow
- une partie contrôle de transition entre ces modes et naturellement décrite par un automate

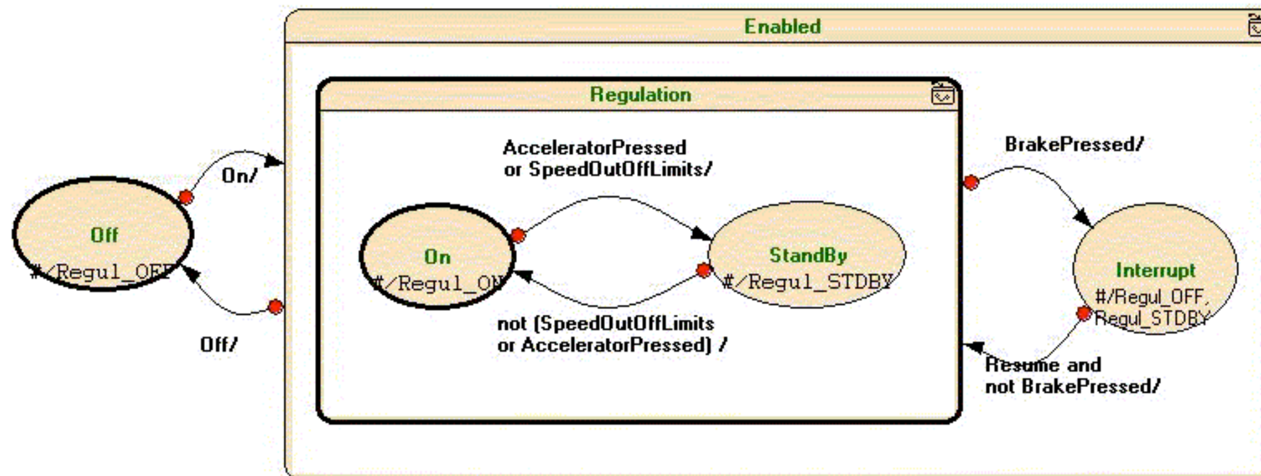
Etendre Scade/Lustre avec des machines à états

Solution existante

- deux (ou plus) langages spécifiques: un pour la partie données, un pour la partie contrôle
- mécanismes de type "édition de liens": un système séquentiel est toujours représenté plus ou moins par:
  - une fonction de transition $f : S \times I \rightarrow O \times S$
  - une mémoire initiale $M_0 : S$
- se mettre d'accord sur une représentation commune + colle
- existe dans la plupart des outils académiques ou industriels
- PtolemyII, Simulink + StateFlow, Lustre + Esterel Studio SSM, etc.

Un exemple: le régulateur de vitesse (SCADE V4.2)





Observations

- les automates apparaissent seulement dans des feuilles du modèle data-flow: on a besoin d'une intégration plus fine
- force le programmeur à prendre des décisions au tout début de la conception (quelle est la bonne méthodologie?)
- la structure de contrôle n'est pas explicite et est cachée dans des variables booléennes: rien n'indique que les modes sont exclusifs
- certification de code?
- efficacité/simplicité du code?
- comment exploiter cette information dans les analyses de programmes et les outils de vérification?

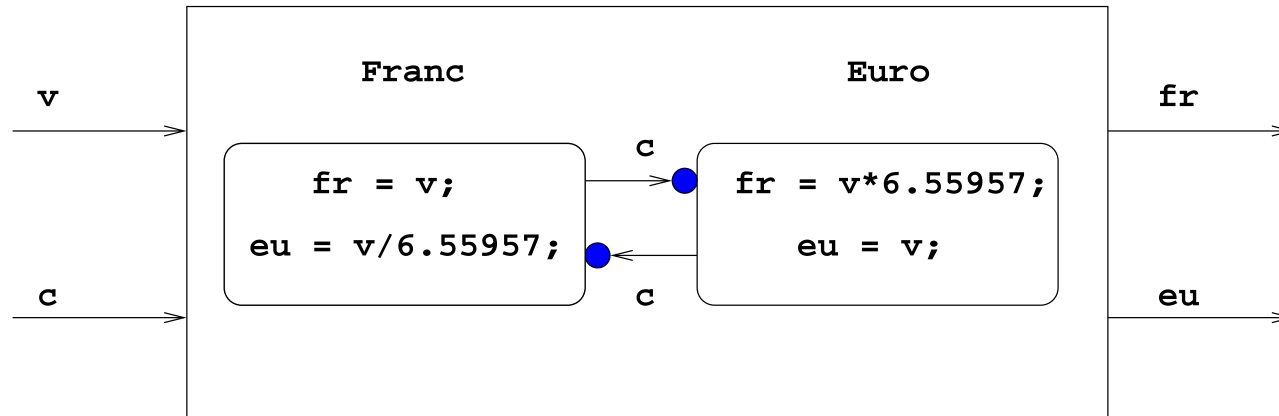
Approche choisie

- étendre un langage synchrone dataflow primitif (Lustre) avec une construction d'automates
- en la fondant sur l'utilisation des horloges: les programmes du langage étendu sont traduits dans un noyau avec horloges
- essentiellement une transformation source-à-source
- produit du code efficace (comparable aux méthodes ad-hoc)
- conservatif (accepte tout Lustre)

Deux implantations

- compilateur ReLuC de Scade à Esterel-Tech.
- Lucid Synchrone V3

Un exemple: le convertisseur Franc/Euro



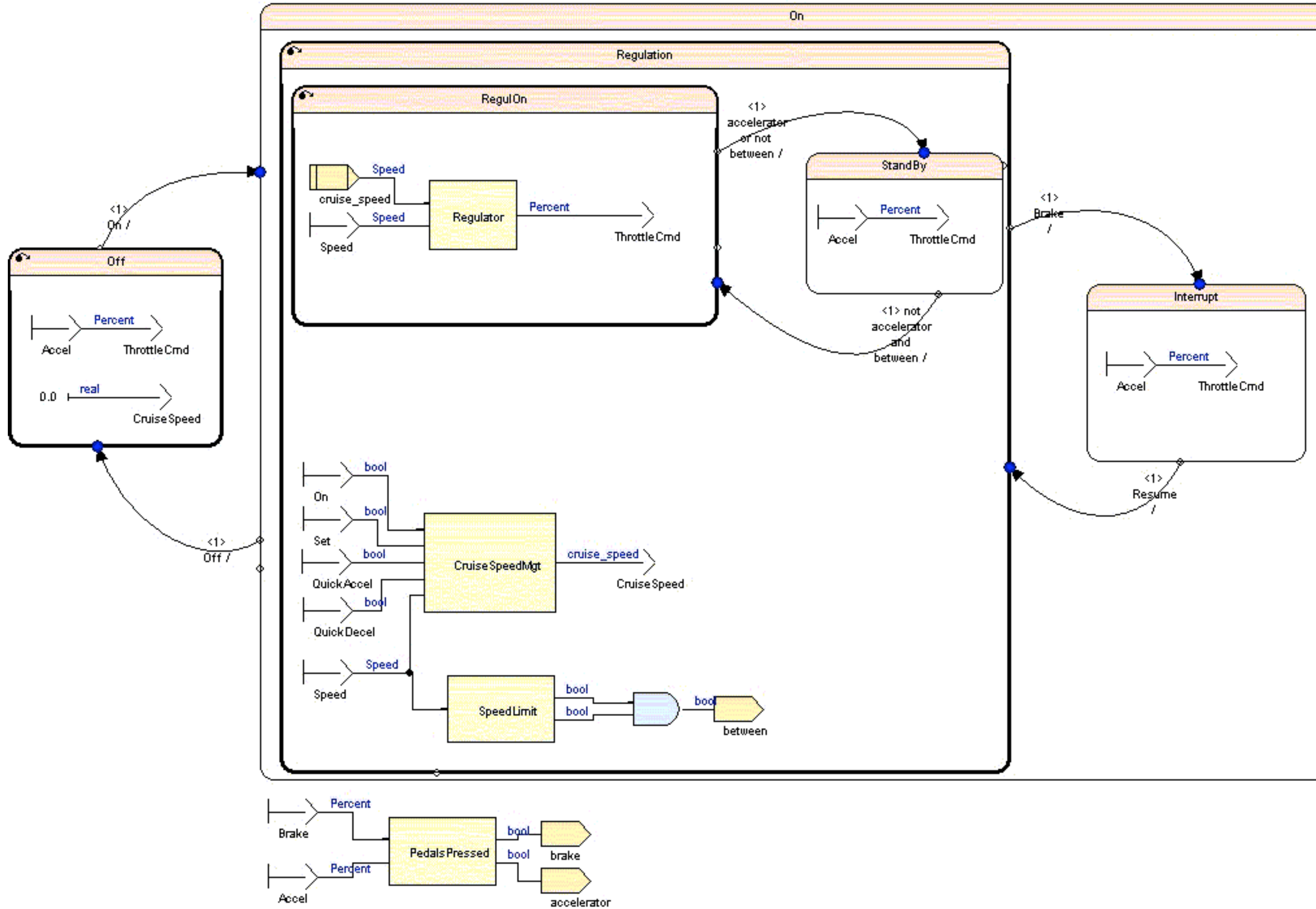
En syntaxe **Lucid Synchrone**:

```
let node converter v c = (euro, fr) where
  automaton
    Franc -> do fr = v and eur = v / 6.55957
                 until c then Euro
  | Euro -> do fr = v * 6.55957 and eu = v
                until c then Franc
  end
```

Le contrôleur de souris

```
let node counting e = cpt where
  rec cpt = if e then 1 -> pre cpt + 1 else 0 -> pre cpt

let node controler click top = (simple, double) where
  automaton
    Await ->
     do simple = false and double = false
     until click then One
  | One ->
     do simple = false and double = false
     unless click then Emit(false, true)
     unless (counting top = 4) then Emit(true, false)
  | Emit(x1, x2) ->
     do simple = x1 and double = x2
     until true then Await
  end
```

Le contrôleur de vitesse (Scade V6)



Autres exemples

- le contrôleur de vitesse
- le contrôleur d'une chaudière
- la machine à café (de Milner)

Langage laboratoire?

Collaboration avec l'équipe Scade depuis 2000

- le compilateur ReLuC de Scade reprend (et améliore) des techniques introduites dans Lucid Synchrone
- typage, calcul d'horloge
- certaines constructions (e.g., `merge`)
- analyses de programmes (initialisation)
- conception/sémantique de Scade V6

Collaboration avec Athys (Dassault-Systèmes) pour l'extension de Catia avec un outil de programmation des automates industriels (LCM)

- synthèse automatique des types (avec polymorphisme)
- autres analyses par typage

Modélisation/simulation de systèmes

Thèse de Louis Mandel (avril 2006)

- concevoir un langage pour la simulation/programmation de systèmes dynamiques complexes
- un ensemble de processus synchrones réactifs
- les ajouter/retirer dynamiquement (e.g., agents, réseaux mobiles, jeux, interfaces graphique)
- observer/debugger des programmes synchrones
- peut-on "relacher" un peu le modèle synchrone?

Proposer un langage fondé sur le modèle "réactif" de Boussinot et combinant:

- **composition parallèle synchrone**, **communication par diffusion instantanée**
- **création dynamique** de processus

ReactiveML

- construit au dessus de **Objective Caml** étendu avec des constructions **reactives**
- Sémantique formelle, programmmes statiquement typéss
- techniques de compilation/ordonnancement efficaces
- distribution (`www-spi.lip6.fr/~mandel/rml`)

Applications:

- simulation de protocoles de routage dans les réseaux mobiles ad-hoc:
- connecté à des outils de modélisation de l'environnement physique (automates non-déterministes, VERIMAG)
- application à la simulation de réseaux de capteurs (concevoir des protocoles basse consommation), VERIMAG+France Telecom

Conclusion et perspectives

Compilation, sémantique

- autres extensions, analyses de programmes, etc.
- compilation certifiée, preuve assistée

Synchronisme "assoupli" pour le calcul vidéo

- relacher (un peu) le calcul d'horloge pour accepter de composer des systèmes non strictement synchrones mais pouvant l'être à insertion de buffer près
- modèle des réseaux de Kahn N-Synchrones [Emsoft'05, POPL'06]
- avec l'équipe Alchémy (INRIA) et Philips NatLabs

Prise en compte des ressources

- comment modéliser le "vrai" temps, les ressources?
- comment compiler le synchrone sur une machine pipeline (et/ou à parallélisme compilé)?