

# **SYNCHRONICS**

**Action d'Envergure INRIA**

**2008 – 2012**

**LANGAGE(S) POUR LA CONCEPTION ET L'IMPLÉMENTATION DES SYSTÈMES  
EMBARQUÉS**

Rencontres INRIA industrie

Toulouse, le 17 mai 2010

## Équipes

Coordination : **Marc Pouzet** et **Alain Girault**

- LIENS (**Marc Pouzet**) et INRIA Saclay (PROVAL) (**Louis Mandel**)
- INRIA Saclay, équipe ALCHEMY (**Albert Cohen**)
- INRIA Grenoble, équipe POP ART (**Alain Girault** et **Bertrand Jeannot**)
- INRIA Rennes, équipe S4 (**Benoît Caillaud**)
- Laboratoire Verimag, équipe Synchrone (**Pascal Raymond** et **Erwan Jahier**)

## Expertise

- langages synchrones ;
- compilation pour architectures séquentielles et parallèles ;
- vérification formelle, test ;
- systèmes hybrides (continu/discret)

## La thématique générale de l'action

Proposer de nouveaux langages (ou des extensions des langages existants) pour la conception de systèmes embarqués :

- fondés sur **un modèle de concurrence synchrone**
- **formellement définis** et offrant des garanties de sûreté à la compilation

### Nouvelles questions :

- Extensions pour prendre en compte des **phénomènes asynchrones** (e.g., gigue, temps d'exécution, communication par buffer)
- **extension hybride** (mélange continu/discret)
- Comment **simuler efficacement des milliers de processus** synchrones créés/tués dynamiquement ?
- Programmation/compilation pour **le calcul intensif** (e.g., vidéo) : comment générer du code parallèle d'efficacité garantie ?

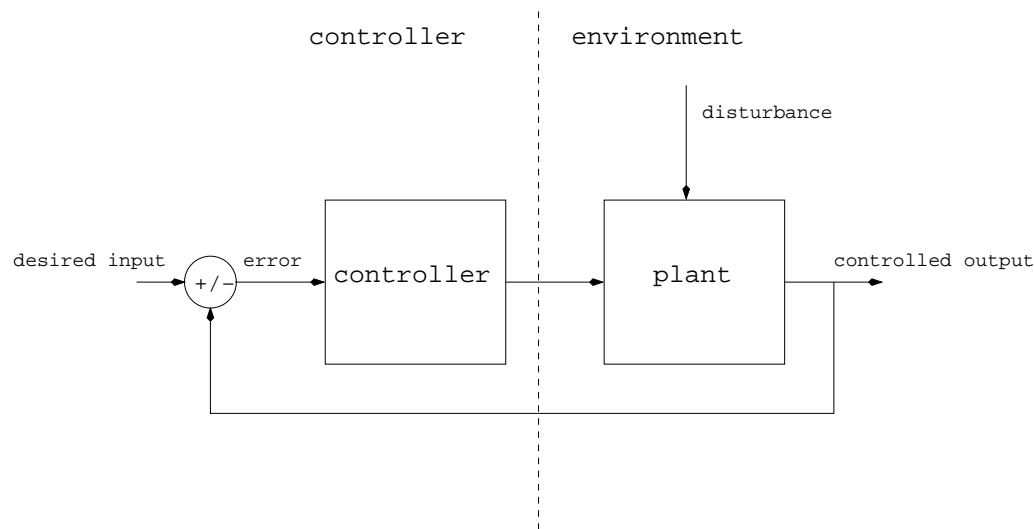
# Systemes embarqués temps-réel

Des systemes réactifs en interaction permanente avec leur environnement.

- avec un **environnement physique** (e.g., commandes de vol, contrôle-moteur)
- ou **d'autres systemes numériques** (e.g., téléphone, TV)

Le temps réel est une notion relative, toujours **liée à l'environnement**. Pour assurer la sûreté, **“quel est le pire cas ?”** ?

L'environnement n'est pas connu précisément : les systemes fonctionnent en **boucle fermée**



Comment programmer/simuler de tels systemes, en se concentrant d'abord sur la **fonction mathématique** indépendamment de certains détails d'implémentation ?

# Langages/outils dédiés pour l'embarqué

## Historiquement, deux cultures/approches différentes :

- langages de simulation : Simulink/StateFlow, Modelica, etc.
- Langages de programmation : langages synchrones (Lustre, Esterel, Signal)

## Standard *de fait* dans l'embarqué :

- **langages de haut niveau** (temps et concurrence)
- proche des formalismes mathématiques de l'ingénieur (**schéma/bloc**, **automates hiérarchiques**), mélange **continu/discret**
- une seule notation pour décrire à la fois le **contrôleur** (i.e., le programme) et son **environnement**
- **simulation et exécution précoce**
- **simulation et compilation** : le même schéma sert à simuler et générer du code embarqué (“what you simulate is what you execute” [Berry, 80's])

## Langages de simulation (e.g., Simulink/StateFlow)

- formalismes de haut niveau (ODE, DAE, automates hybrides)
- permet de décrire fidèlement l'environnement physique (continu)
- outillage riche : solveurs numériques, calcul de pas, stabilité, etc.
- de plus en plus de **génération de code embarqué** (séquentiel/parallèle)

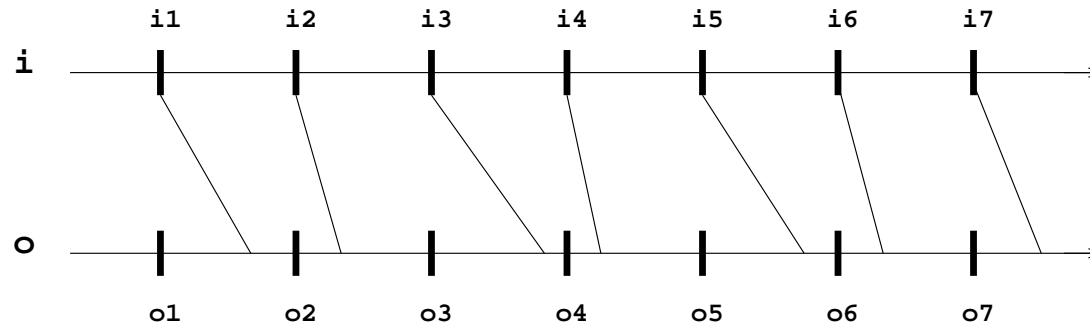
### Limitations

- **sémantique imprécise** (ou non spécifiée)
- certaines constructions sont **peu sûres** (e.g., variables partagées, StateFlow)
- frein à la génération de code efficace et sûr et à la vérification formelle
- problèmes a **l'interface discret/continu** : pas de sémantique de l'ensemble

**Comprendre la richesse offerte par Simulink dans un cadre aussi rigoureux que celui des langages synchrones ?**

# Langages de programmation (e.g., SCADE/Lustre, Esterel, Signal)

Séparer la **fonctionnalité** du système de son **implémentation**



- le temps est logique = c'est la succession de réactions atomiques du système
- on raisonne idéalement (en pire-cas) et on vérifie **a posteriori** que l'implémentation est suffisamment rapide (repose sur l'analyse WCET)

## Avantages/Limitations

- formellement définis ; outils de vérification/test
- propriétés de sûreté garanties (temps/mémoire bornés, déterminisme)
- génération de code embarqué
- mais... pas de prise en compte du temps continu
- génération de code parallèle (calcul intensif) peu étudiée

## Exemple : SCADE/Lustre

Modéliser/programmer le logiciel embarqué critique.

**L'idée de Lustre (Caspi et Halbwachs, 84) :**

- écrire directement des équations de suites vues comme des **spécifications exécutables**
- fournir un **compilateur** et des outils d'analyse dédiés pour produire du code

E.g, le filtre linéaire défini par :

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

est programmé en écrivant :

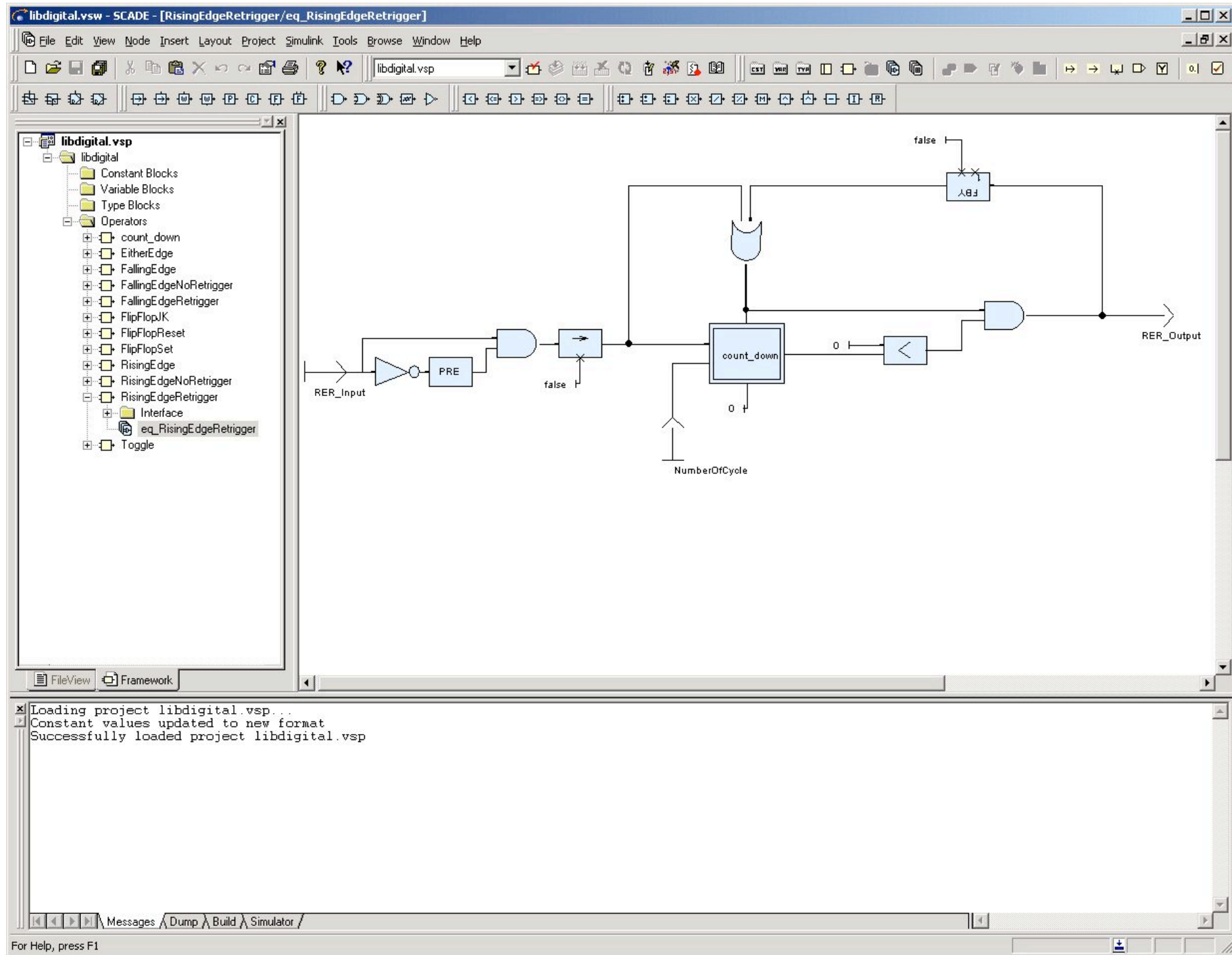
$$Y = (0 \rightarrow a * \text{pre}(Y)) + b * X$$

on écrit des **invariants**

- d'autres opérateurs permettent de composer des processus lents et rapides (sous/sur-échantillonnage) ; non nécessairement périodiques



# Un exemple de planche SCADE (V5)



# Programmation fonctionnelle synchrone

Lustre est un **langage fonctionnel** : un système est une fonction qui reçoit des suites en entrée pour produire des suites en résultat

Quel lien avec la théorie des langages fonctionnels typés (e.g., Ocaml, Haskell) ?

## Répondre à plusieurs types de questions :

- augmenter la **modularité/expressivité**, la **réutilisation** : synthèse de types, polymorphisme, ordre supérieur
- mélanger **contrôle** (automates) et **données** (équations data-flow)
- proposer des analyses et systèmes de types dédiés : le programme est-il synchrone ? est-il causal ? est-il à mémoire finie ? déterministe ?
- définir la sémantique formellement ainsi que le processus de traduction vers du code exécutable

## Lucid Synchrone

Comment étendre Lustre en conservant ses propriétés essentielles ?

### **Construire un langage “laboratoire”**

- étudier et proposer des extensions de Lustre
- mettre en oeuvre, expérimenter des extensions/analyses statiques en gérant toute la chaîne de compilation

### **en suivant quelques principes :**

- conserver la sémantique de Lustre
- formuler de data-flow synchrone dans le cadre des langages fonctionnels typés
- composition de fonctions, propriétés statiques décrites par des types

**Un exemple d’extension :** mélange des automates hiérarchique et du data-flow.

# Extensions : comment mélanger automates et data-flow ?

**Systemes dominés “données”** : typiquement, à partir de systèmes continus échantillonnés

- schémas de type Simulink ou SCADE

**Systemes dominés “contrôle”** : contrôle discret, systèmes de transition

- StateFlow, StateCharts ou Esterel/SSM

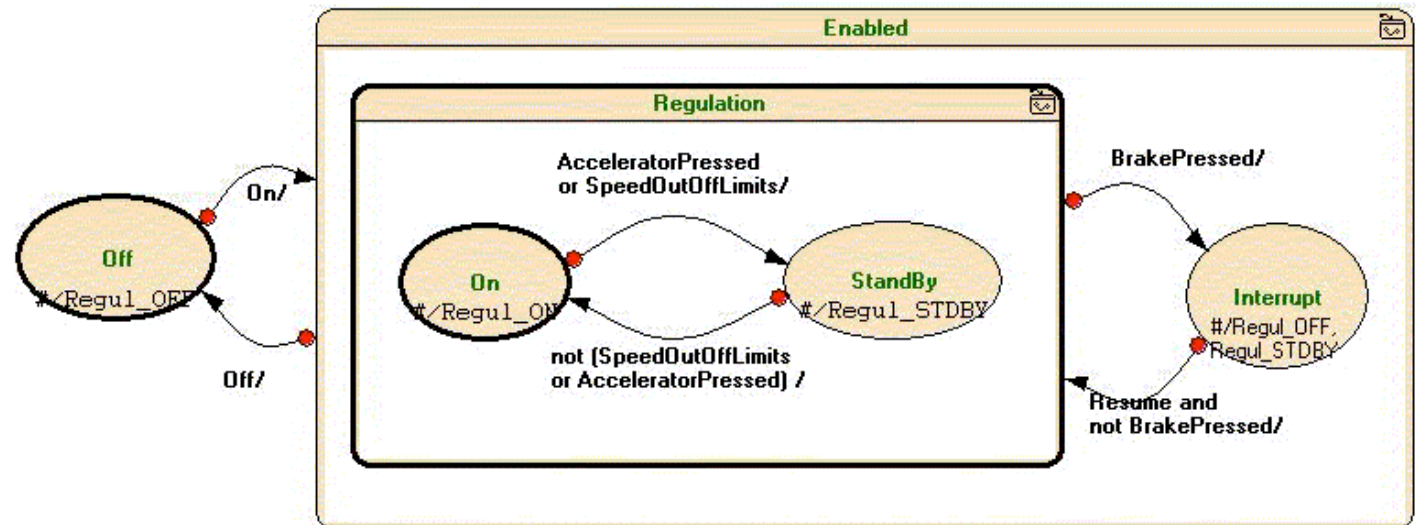
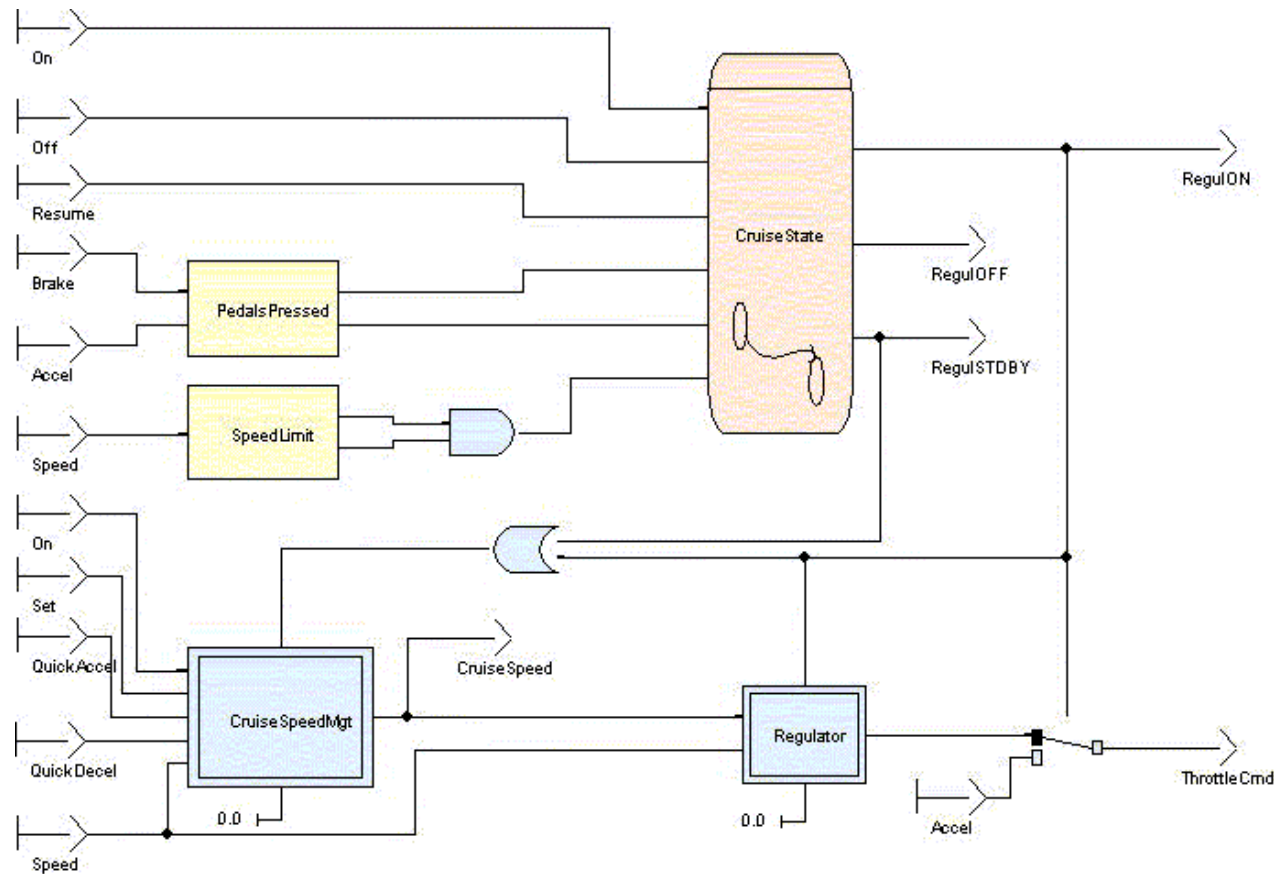
**Les systèmes réels combinent les deux**

- ils ont des **modes** : “en plongée” vs “en surface”
- chacun des mode est défini par une loi de commande exprimée sous forme d'équations data-flow
- la partie contrôle est décrite par un automate hiérarchique

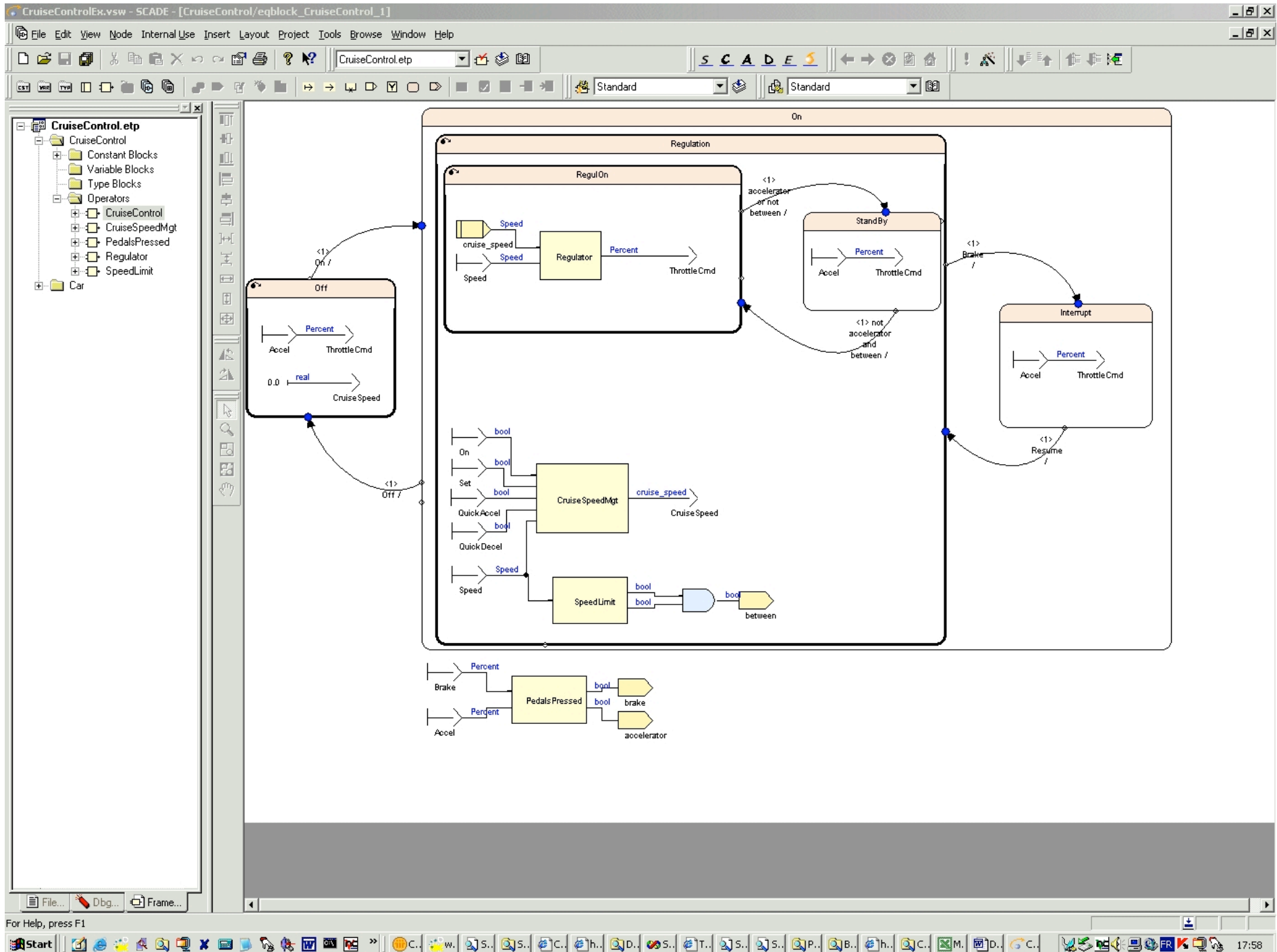
Les outils existants permettent de combiner les deux styles : Simulink/StateFlow, SCADE/Esterel SSM, Ptolemy, etc.

Un peu *ad-hoc* ; pas de sémantique précise donnée à l'ensemble

# Contrôle de vitesse en SCADE+SSM (V4)



# Scade 6 (2008)



## Extensions du data-flow synchrone

Le plongement du data-flow synchrone dans le cadre fonctionnel s'est révélé très utile pour proposer de nouvelles extensions à SCADE/Lustre.

Plusieurs traits initialement introduits dans **Lucid Synchrone** ont été repris dans des outils industriels.

- le compilateur ReLuC de SCADE (maintenant SCADE 6) ont repris (et amélioré) certaines techniques utilisées dans Lucid Synchrone
- même approche : systèmes de types dédiés, modularité
- constructions de programme (e.g., `merge`), structures de contrôle
- analyses statiques (analyse d'initialisation, calcul d'horloge)

De nouvelles applications aux logiciels de conception 3D en mélangeant simulation et programmation.

## Actions de recherche en cours dans Synchronics

Un langage synchrone fonctionnel servant de base pour des expérimentations et extensions nouvelles.

### Principales questions abordées :

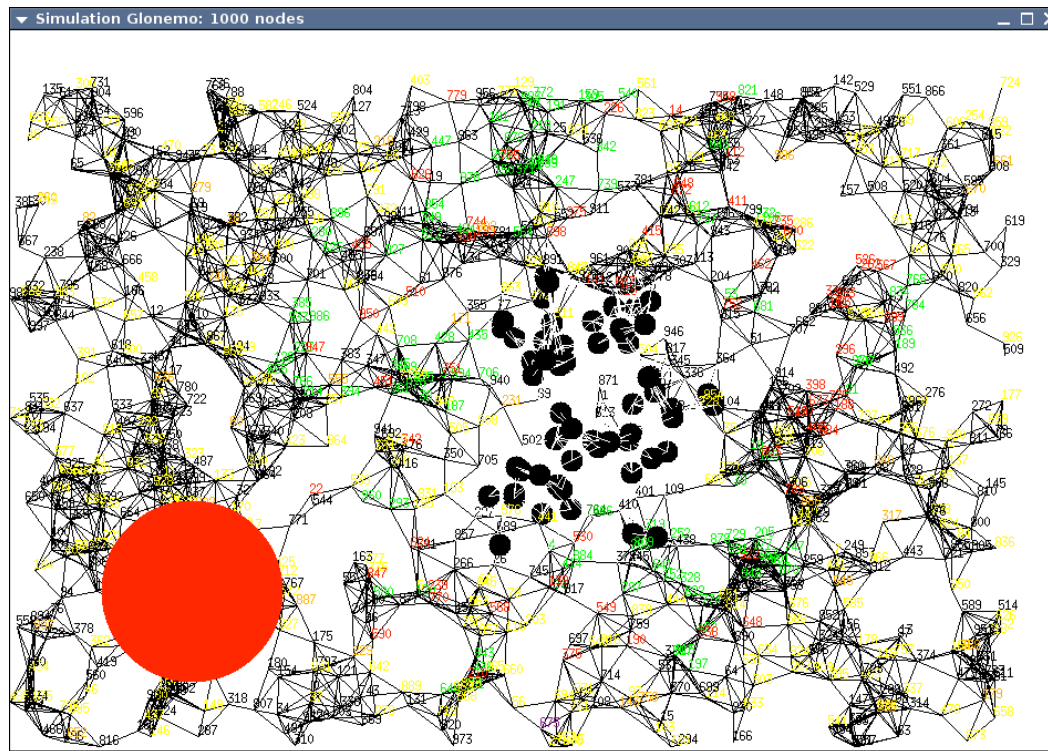
- *simulation massive de systèmes réactifs créés/détruits dynamiquement*
- *N-synchronisme pour modéliser/programmer des systèmes à asynchronisme borné (e.g., communication par buffer, gigue)*
- génération de code parallèle d'efficacité garantie à partir d'un programme synchrone
- vérification modulaire de programmes synchrones
- mélange du temps continu/temps discret (en somme, définir un sous-ensemble de Simulink "propre")



# Systemes avec création dynamique

## Simulation de réseaux de capteurs (VERIMAG et FT, depuis 2007)

- le système combine du temps réel et du dynamique
- prendre en compte tous les aspects de la simulation : les noeuds, l'interaction entre eux et l'environnement, l'ajout dynamique éventuel, l'interface (affichage, métriques, etc)...



Exemple : simulation de la consommation d'énergie dans un réseau de capteurs

# Le langage ReactiveML (Louis Mandel) — (DÉMO)

- une extension de Ocaml basée sur le modèle de Boussinot
  - les programmes sont causaux par construction ; la réaction à l'absence est décallée d'un instant
- les processus peuvent être créés/détruits dynamiquement

```
(* a piece of RML code *)
```

```
let rec process add new_node =  
  await new_node(pos) in  
  run (add new_node) || run (node pos)
```

- **interface avec des langages de modélisation de l'environnement** (Lucky)
- un **oplevel**, lui-même programmé en RML : le même langage sert à la fois pour programmer le système et contrôler la simulation !
- simulation de plusieurs **milliers de processus** :
  - ordonnancement dynamique des processus
  - problème dur : éviter l'attente active

## Du synchrone au *N*-synchrone

**Problème** : traiter une classe plus large de systèmes synchrones

- communication par FIFO bornés
- prenant en compte la gigue dans les systèmes de calcul vidéo
- modéliser les temps d'exécution et les contraintes d'ordonnancement
- donner plus de liberté au compilateur/optimizeur

**Travaux reliés** : systèmes insensibles à la latence (Carloni, De Simone, etc.), circuits élastiques (Cortadella), SDF (Ed. Lee)  
calcul réseau (Boudec, etc.), calcul temps réel (Thiele)

**Thèse de Florence Plateau** : [www.lri.fr/~plateau](http://www.lri.fr/~plateau), Jan. 2010.

## Un exemple typique : le réducteur d'échelle (downscaler)

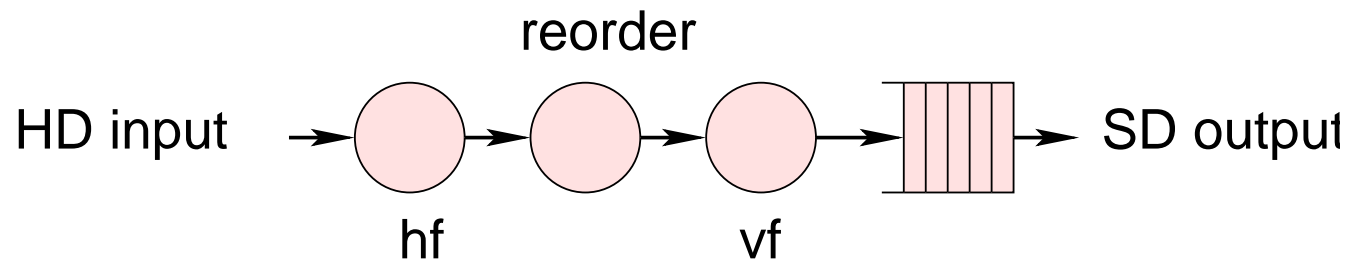
image haute définition (HD) → définition standard (SD)

1920 × 1080 pixels

720 × 480

**filtre horizontal** : nombre de pixels par ligne (1920 pixels à 720 pixels)

**filtre vertical** : nombre de lignes de 1080 à 480



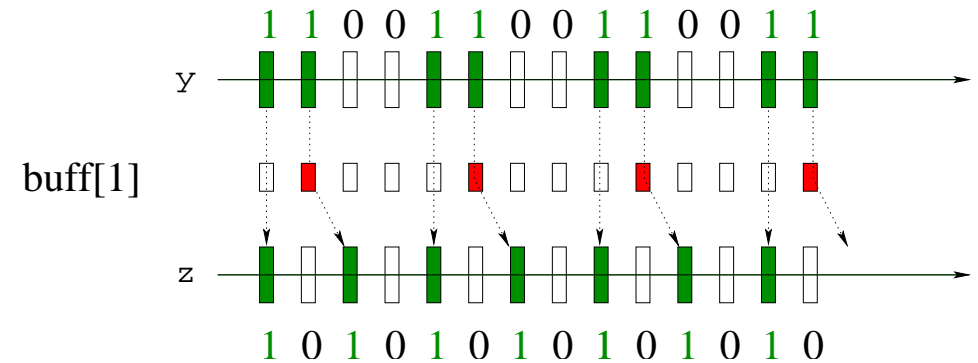
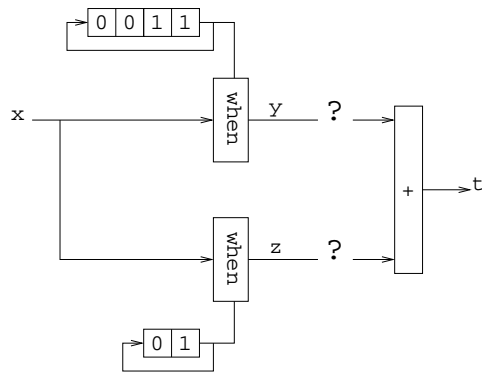
### Contraintes de temps réel

l'entrée et la sortie : : 30Hz.

les pixels HD entrée arrivent à :  $30 \times 1920 \times 1080 = 62,208,000 Hz$

ls pixels SD en sortie à  $30 \times 720 \times 480 = 10,368,000 Hz$  (6 fois moins vite)

# Synchronisme relaché



## Composition synchrone

- les flots doivent être synchrones lorsqu'ils sont composés
- l'ajout d'un buffer (à la main) est difficile et source d'erreurs

## Composition *N*-synchrone

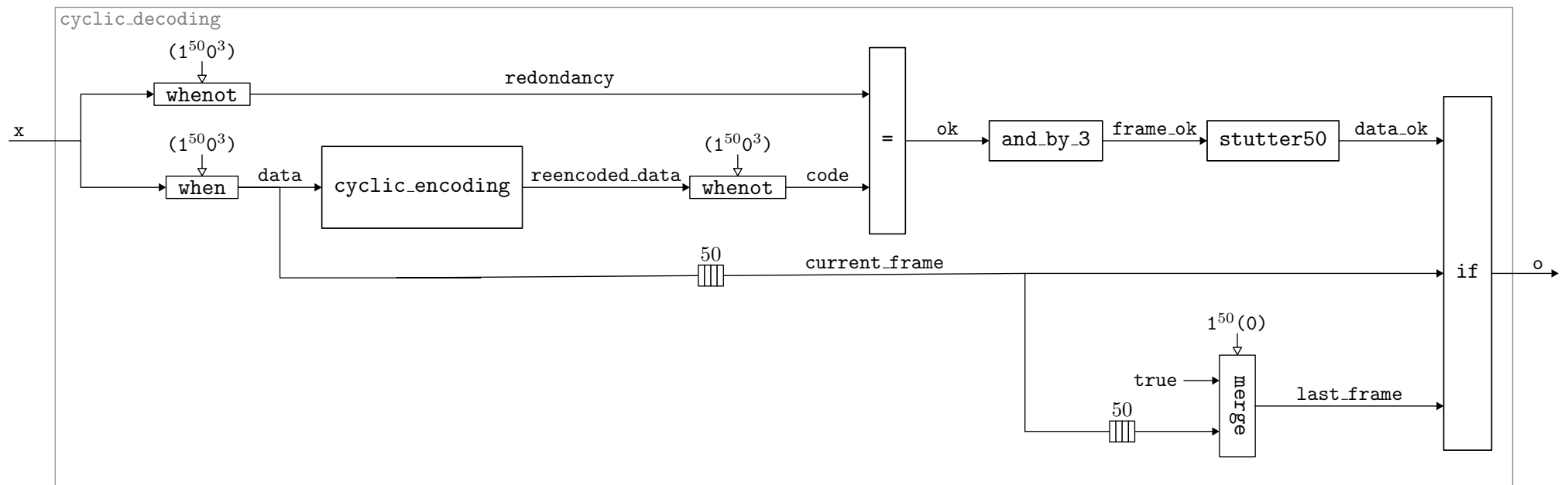
- la communication à travers des buffers dont la taille est garantie par le compilateur
- si  $w_1 <: w_2$  alors un buffer est ajouté, e.g.,  $(1100) <: (10)$

## Langage LucyN (Mandel, Plateau ; disponible depuis Jan. 2010)

- une extension de Lustre fondée sur le *N*-synchrone
- calcul automatique (durant le typage) de la taille des buffers

## Exemple : l'encodeur GSM — (DÉMO)

- le programmer écrit simplement `buffer(x)` lorsqu'un buffer peut être utilisé
- sa taille est calculée automatiquement



## Conclusion/Travaux en cours

- méler scheduling statique/dynamique pour augmenter (encore) l'efficacité de la simulation des systèmes de grande taille
- théorie du  $N$ -synchrone ; répartition de code
- génération de code en boucle et parallèle
- vérification modulaire
- extension hybride de SCADE/Lustre