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 Jeannet**)
- 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., gîgue, temps d'exécution, communication par buffer)
– **extension hybride** (mélange continu/discret)
– Comment **simuler efficacement des milliers de processus** synchrones crées/tués dynamiquement ?
– Programmation/compilation pour **le calcul intensif** (e.g., vidéo) : comment générer du code parallèle d'efficacité garantie ?

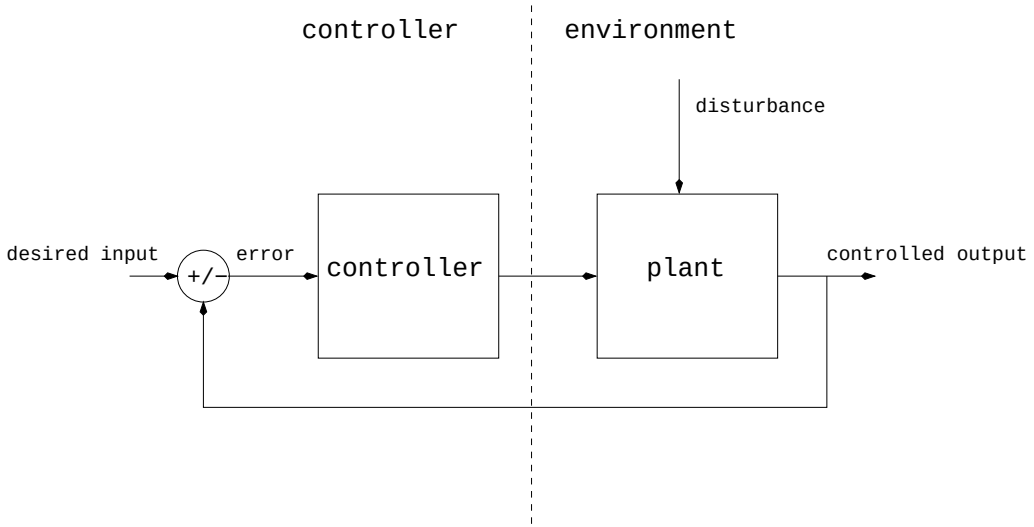
Systèmes embarqués temps-réel

Des systèmes réactifs en interaction permanente avec leur environnement.

– avec un **environnement physique** (e.g., commandes de vol, contrôle-moteur)
– ou **d'autres systèmes 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 systèmes fonctionnent en ***boucle fermée***



Comment programmer/simuler de tels systèmes, en se concentrant d'abord sur la **fonction mathématique** indépendemment 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 (**shéma/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)

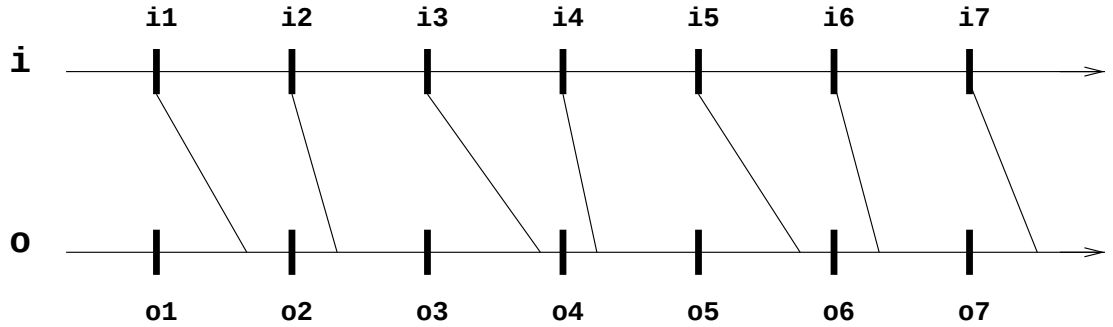
Limitations

- **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 **fonctionalité** 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 :

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

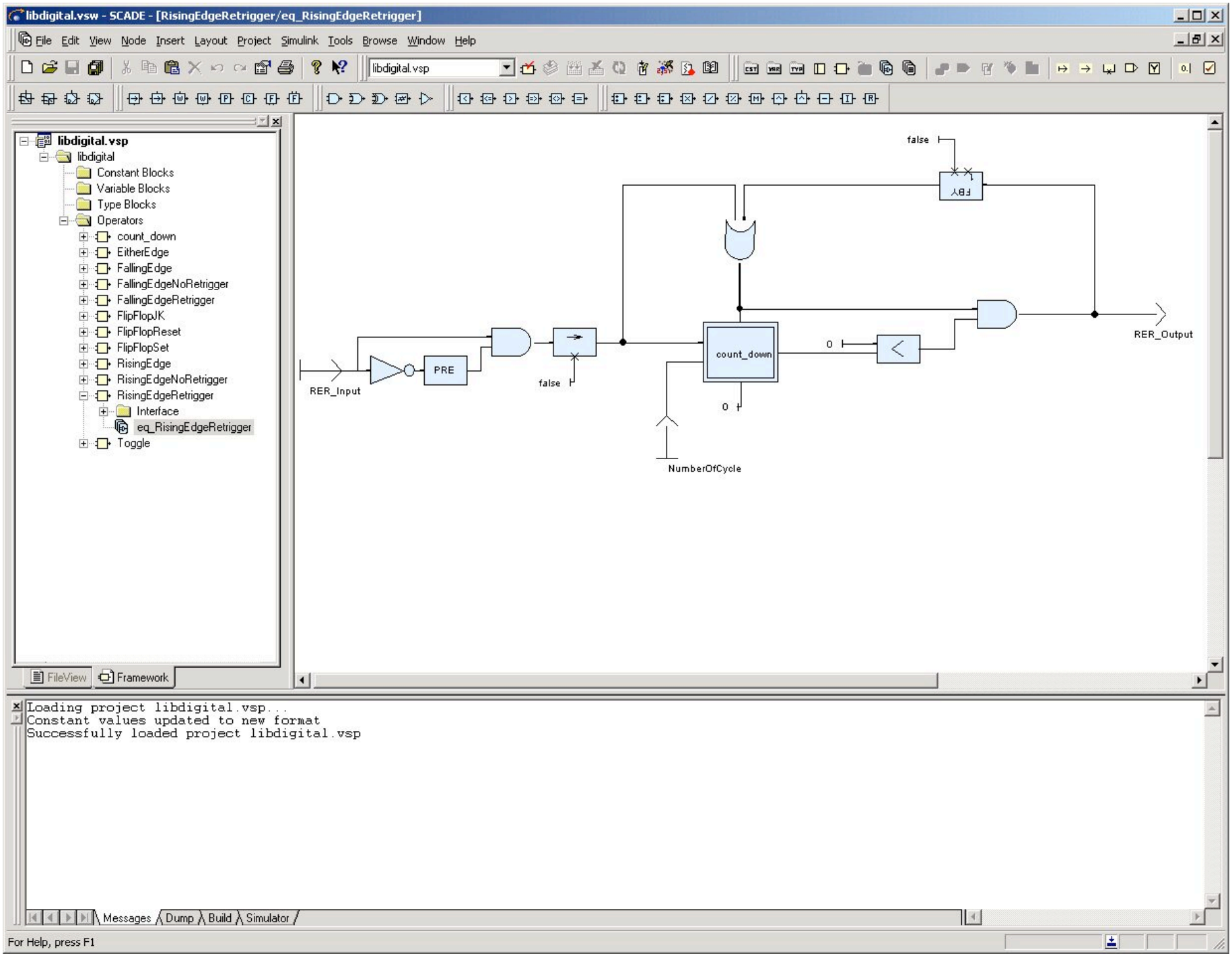
est programmé en écrivant :

```
Y = (0 -> a * 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 recoit 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 chaine 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

Deux exemples : (1) calcul d’horloge par typage et (2) mélange automates/data-flow

Calcul d'horloges par typage

Comment mélanger des rythmes différents de manière sûre ?

– systèmes **échantionnés** (logiciel), **multi-horloge** (hardware)
– que signifie la communication entre deux composants qui ne **partagent pas une échelle de temps commune** ?
– **rejeter statiquement** les erreurs de synchronisation

x	$=$	x_0	$*$	$*$	$*$	x_1	$*$	x_2	$*$	$*$	x_3	x_4	$*$	x_5	$\ldots$
y	$=$	$*$	y_0	y_1	y_2	$*$	y_3	$*$	$*$	y_4	$*$	y_5	$*$	$*$	$\ldots$

– exprimer l'information **d'horloge** sous la forme d'un **type** qui donne les instants ou une valeur est produite
– la vérification des horloges devient un problème de typage
– propriété de sûreté garantie à la compilation : les programmes bien typés (du point de vue des horloges) s'exécutent de manière synchrone

Extensions : comment mélanger automates et data-flow ?

Systèmes dominés "données" : typiquement, à partir de systèmes continus échantillonnés

– schémas de type Simulink ou SCADE

Systèmes 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"

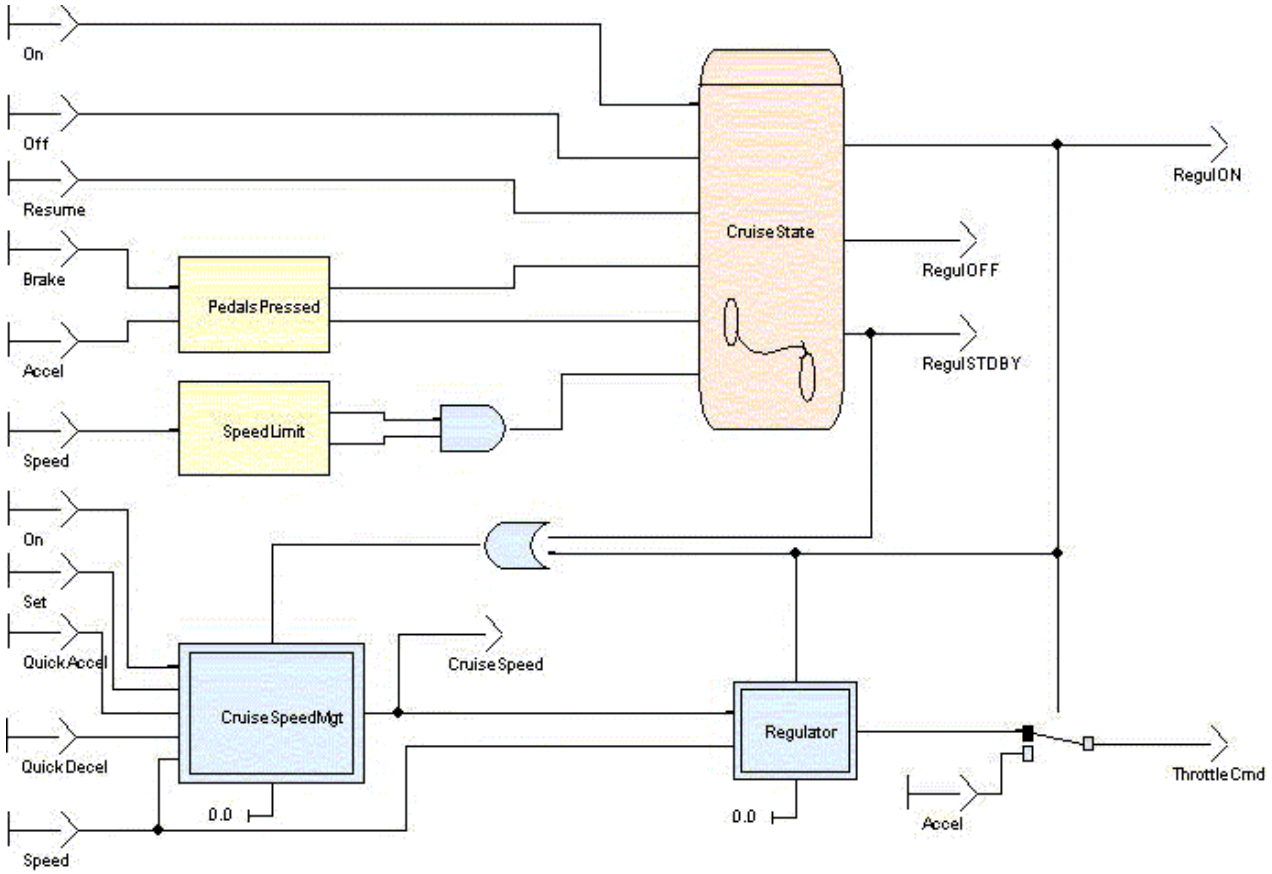
– chacun des mode est défini par une loi de commande exprimée sous forme d'équations data-flow

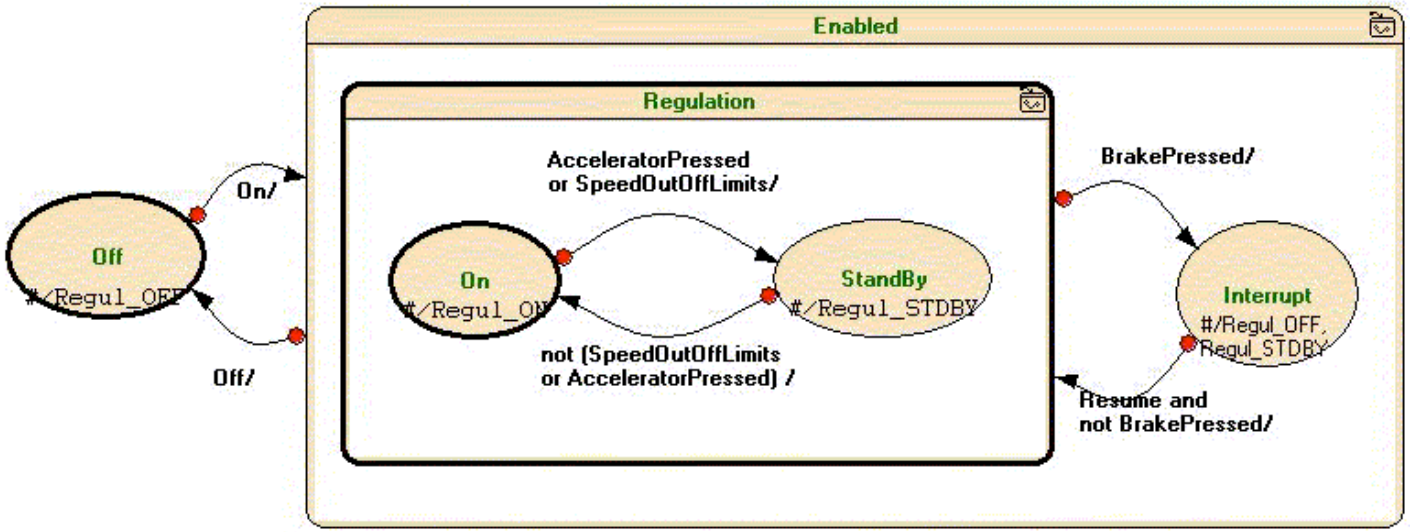
– la partie contrôle est décrite par un automate hiérarchique

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

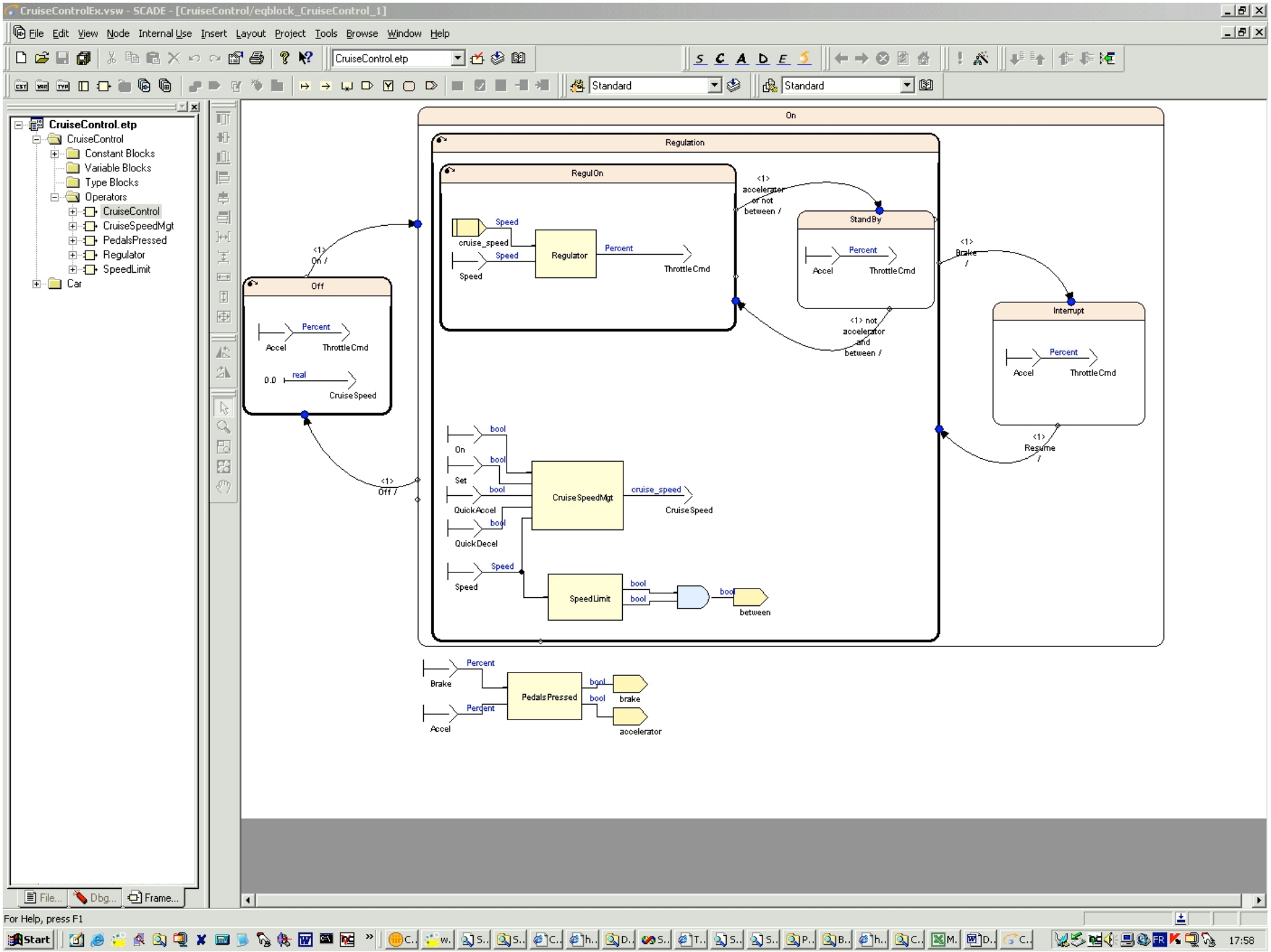
Un peu *ad-hoc* ; pas de sémantique précise donnée à l'ensemble

Contrôle de vitesse en SCADE+SSM (V4)





Scade 6 (2008)


CruiseControlEx.vsw - SCADE - [CruiseControl/eqblock_CruiseControl_1]
File Edit View Node Internal Use Insert Layout Project Tools Browse Window Help
CruiseControl.etp
Standard
Standard
CruiseControl.etp
CruiseControl
Constant Blocks
Variable Blocks
Type Blocks
Operators
CruiseControl
CruiseSpeedMgt
PedalsPressed
Regulator
SpeedLimit
Car
On
Regulation
RegulOn
Speed
cruise_speed
Speed
Regulator
Percent
ThrottleCmd
<1> accelerator or not between /
StandBy
Percent
Accel
ThrottleCmd
<1> Brake /
Interrupt
Percent
Accel
ThrottleCmd
<1> Resume /
<1> not accelerator and between /
<1> On /
Off
Percent
Accel
ThrottleCmd
0.0
real
CruiseSpeed
<1> Off /
bool
On
Set
QuickAccel
QuickDecel
CruiseSpeedMgt
cruise_speed
CruiseSpeed
Speed
Speed
SpeedLimit
bool
bool
between
Percent
Brake
Percent
Accel
PedalsPressed
bool
bool
brake
accelerator
File...
Dbg...
Frame...
For Help, press F1
Start
17:58

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 :

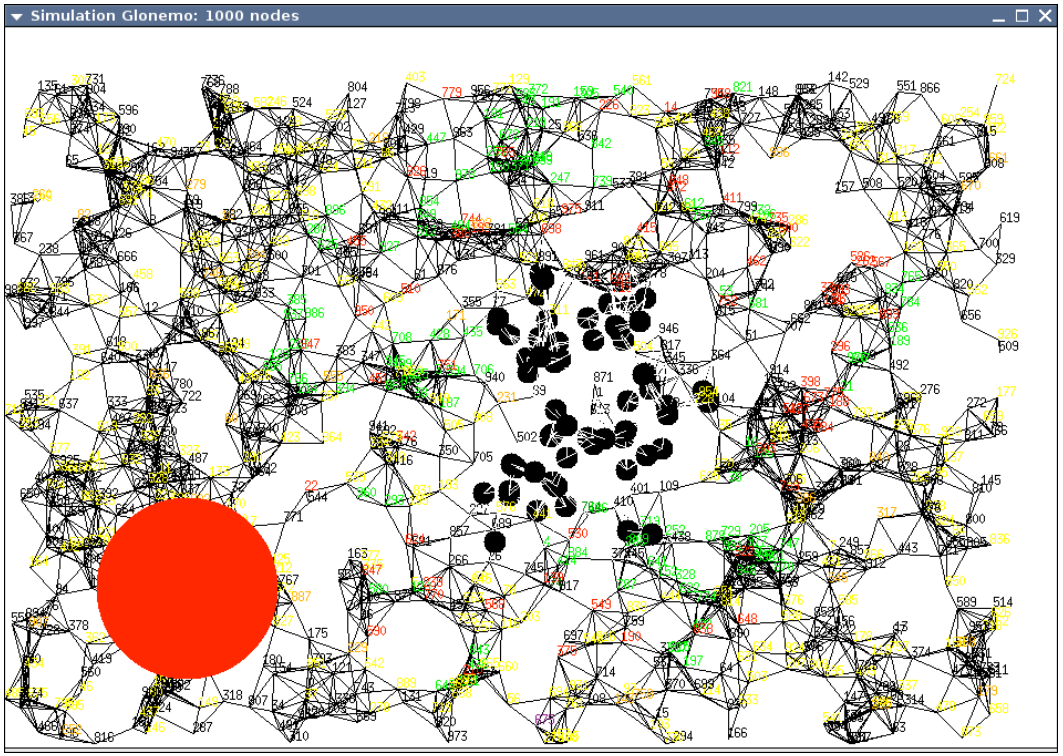
- *simulation massive de systèmes réactifs créés/détruits dynamiquement*
- *N-synchronisme pour modéliser/programmer des systèmes à asynchronisme borné (e.g., communication par buffer, gîgue)*
- génération de code parallèle d'efficacité garantie à partir d'un programme synchrone
- vérification modulaire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

Systèmes 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 **toplevel**, lui-même programmé en RML : le même langage sert à la fois pour programmer le système et controler 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 gîgue dans les systèmes de calcul vidéo
– modéliser les temps d'exécution et les contraintes d'ordonnancement
– donner plus de liberté au compilateur/optimiseur

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`, Jan. 2010.

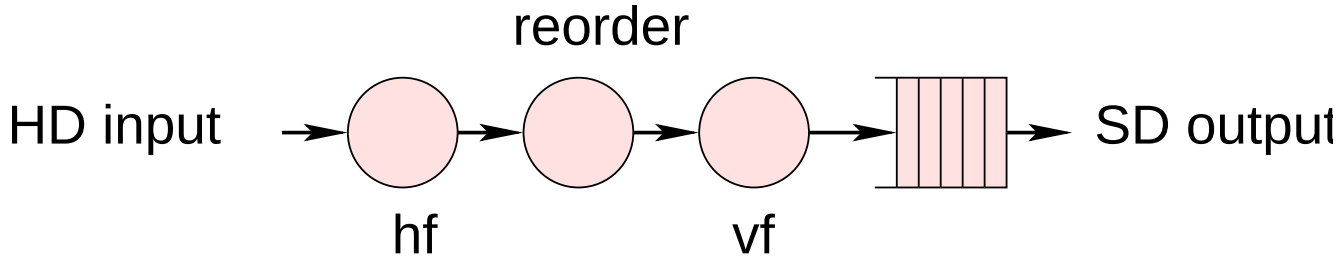
Un exemple typique : le réducteur d'échelle (downscaler)

image haute définition (HD) $\longrightarrow$ 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



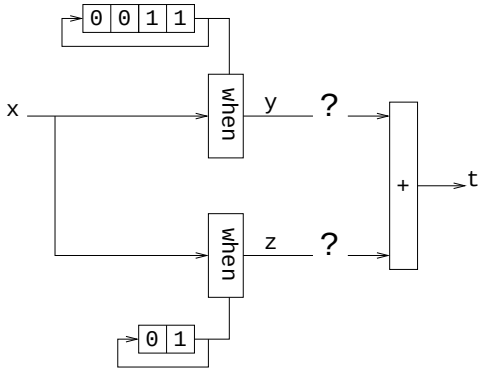
Contraintes de temps réel

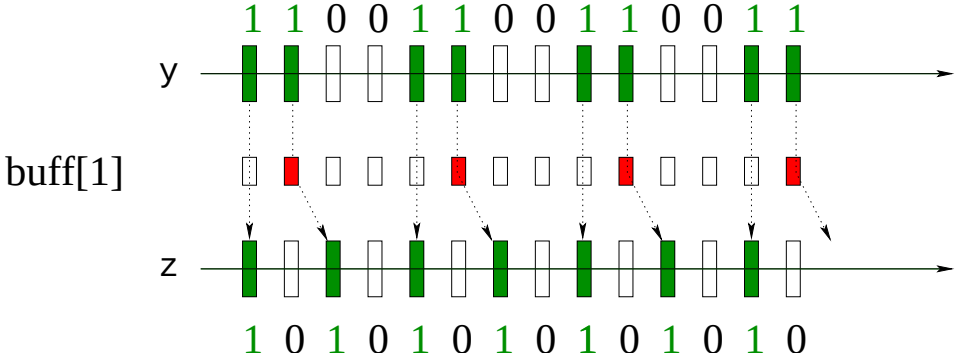
l'entrée et la sortie : : 30Hz.

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., $(1100) <: (10)$

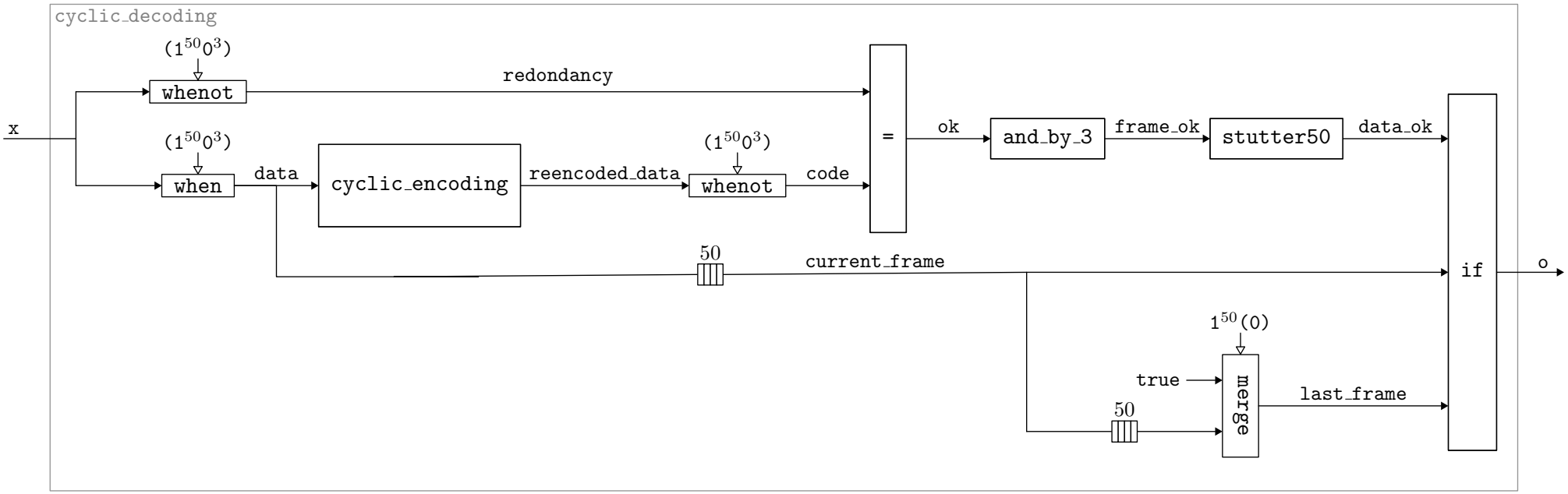
Langage LucyN (Mandel, Plateau ; disponible depuis Jan. 2010)

– 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