Lustre est discret, mais il continue! [1]

Marc Pouzet

UPMC/DI ENS/Inria Paris

Marc.Pouzet@ens.fr

4 juin 2018

Journée en l'honneur de Nicolas Halbwachs
Grenoble

[1] L'idée du titre est due à Sylvan Dissoubray (ANSYS).

Programmer et contrôler un système réactif?

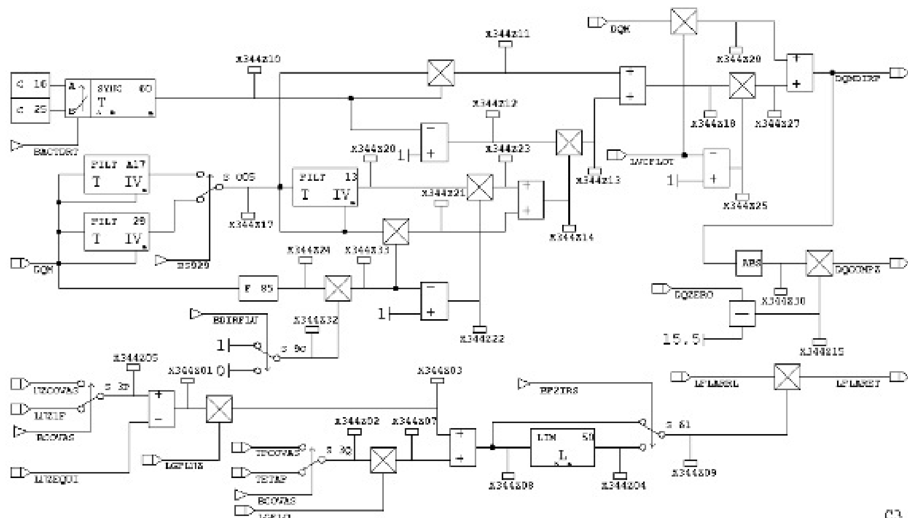
- commande de vol: $\approx$ 1,5MLOC
- logiciel critique
- "dynamique" = "trop tard"
- exécution cyclique

Écrire du code Assembleur/C/C++/JavaScript/Python/OCaml/Coq à la main?

Et le comparer alors a quoi?

Quelle est la spec?

SAO (Spécification Assistée par Ordinateur) — Airbus 80's



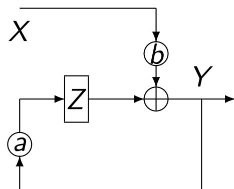
Des dessins très précis...

Les automaticiens/traiteurs de signaux décrivaient les systèmes de contrôle/commande avec des mathématiques très précises avant même l'arrivée de calculateurs.

Equations de suites, transformée en Z, etc.

Exemple: un filtre linéaire

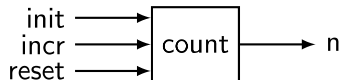
$$Y_0 = bX_0 \;,\; \forall n \; Y_{n+1} = aY_n + bX_{n+1}$$



...mais non exécutables! Écrire du code et se convaincre qu'il est juste?

Comment rendre ces mathématiques exécutables?

Quelque part à Grenoble... le langage Lustre (1984)




```
node COUNT (init, incr: int; reset: bool)
   returns (n: int);
let
   n = init ->
      if reset then init else pre(n) + incr;
tel;
```

Programmer en écrivant des équations de suites

Un système discret: une fonction de suites; les suites sont synchrones.

X	1	2	1	4	5	6	...
Y	2	4	2	1	1	2	...
$X+Y$	3	6	3	5	6	8	...
pre X	*nil*	1	2	1	4	5	...
Y -> X	2	2	1	4	5	6	...

L'équation $Z = X + Y$ signifie $\forall n. Z_n = X_n + Y_n$.

Le temps est logique: les entrées X et Y arrivent "en même temps"; la sortie Z est produite "en même temps"

Euh... c'est temps réel?

Raisonner en pire cas: vérifier que le code généré produit la sortie avant l'arrivée de l'entrée suivante.

Exemple: le contrôleur de chaudière [2]

Modèle de l'environnement

- u est la commande. $u = true$ (chauffe); $u = false$ (ne chauffe pas)
- α, β, c sont des paramêtres; ext est la température extérieure.
- La vitesse $temp'$ de la température $temp$ évolue selon:

$$temp' \;=\; \alpha(c - temp) \text{ si } u \quad \beta(ext - temp) \text{ sinon}$$

On discrétise (avec un pas h)

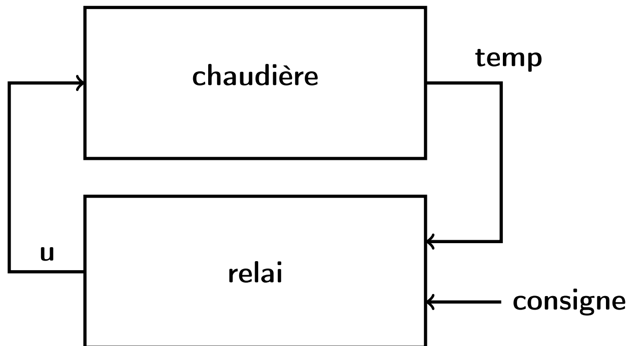
$$temp' \text{ est approximé par la différence } (temp_{n+1} - temp_n)/h$$

Contrôleur discret (relai)

$$\begin{array}{rcl} u_n & = & true \text{ si } temp_n < bas \quad false \text{ si } temp_n > haut \\ u_n & = & false \text{ si } n = 0 \text{ sinon } u_{n-1} \end{array}$$

[2] Exemple présenté par Nicolas Halbwachs au CdF (2010).

Boucle



Démo

L'idée géniale de Lustre

Programmer en écrivant directement des équations mathématiques

Les analyser/transformer/simuler/tester/vérifier

Les traduire automatiquement vers du code exécutable

SCADE: Safety Critical Application Dev. Env. (Verilog, 95)

Au même moment...

À Rennes: le langage Signal (1987)

- Même approche que Lustre mais un peu plus expressif.
- Un système = un ensemble de contraintes sur des suites.

À Nice: le langage Esterel (1985)

- Style impératif plus proche de l'informatique.
- Interruption, suspension d'une tâche, mise en parallèle.
- Comment rendre cela déterministe?

La même approche synchrone:
(1) raisonner idéalement; (2) calculer le WCET du code généré.

C'est bien plus simple.

Mais certain programmes conduisent à des monstres...

comment les rejeter?

Différente échelles de temps

Synchroniser des processus lents et rapide?

X	1	2	3	4	5	...
half	*true*	*false*	*true*	*false*	*true*	...
X *when half*	1		3		5	...
$X + (X$ *when half*$)$	2		5		8	...

```
let half = true -> not (pre half);
    o = x + (x when half);
tel
```

Définit la suite: $\forall n \in \mathbb{N}. o_n = x_n + x_{2n}$

- ne peut être implémentée à mémoire (buffer) bornée;
- le rejeter statiquement: on peut le faire par typage.

Analyser les causalités entre signaux

Des programmes ont zéro solutions (*deadlock*) ou trop (*non déterminisme*)

En Lustre/Signal

- `x = y + 1 and y = x + 2`
- `y = x and x = y`

En Esterel

- `present S else emit S`
- `present S1 then emit S2 || present S2 else emit S1`

Découverte

La "bonne" notion de causalité est celle de l'électricité.

Si on "cable" le programme synchrone, les sorties sont-elles stables?

Coincide avec ce qui est démontrable en logique constructive.

Les choix de Lustre

Toujours favoriser la simplicité.

Suites et fonctions de suite.

Style data-flow pur. Vérification des horloges = vérification de types.

Causalité = toute boucle doit traverser un délai.

Générer du code en boucle simple et en automates.

Un peu après, quelque part entre Grenoble et Jussieu...

Lucid Synchrone et ReactiveML

Une idée de Paul Caspi, en 1994, à Grenoble.

> *"Marc, observe bien, on peut écrire des programmes Lustre récursifs en quelques lignes de LazyML!"*

Très expressif: ordre supérieur, inférence des types, récursivité, etc.

mais les "monstres" sont toujours la.

du typage, du typage, du typage... et adapter la compilation.

Lucid Synchrone (95-05)

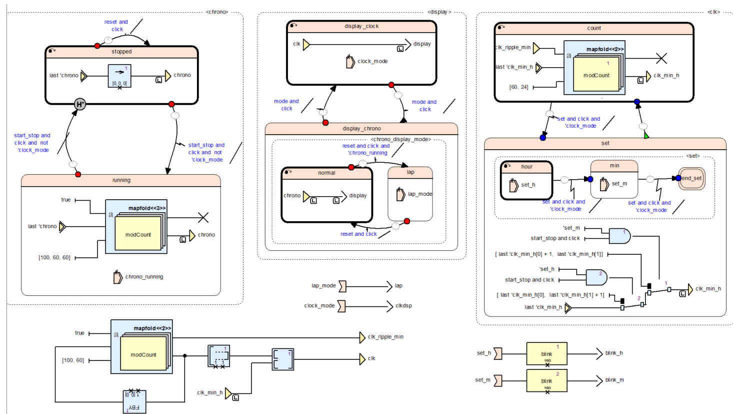
Construire un langage synchrone fonctionnel avec des traits de ML

ReactiveML (05-15)

Du parallélisme synchrone dans un langage à la ML (OCaml)

SCADE 6 (2008-): la grande unification

Un nouveau langage et un nouveau compilateur, en OCaml
(et plein d'idées de Lucid Synchrone dedans)



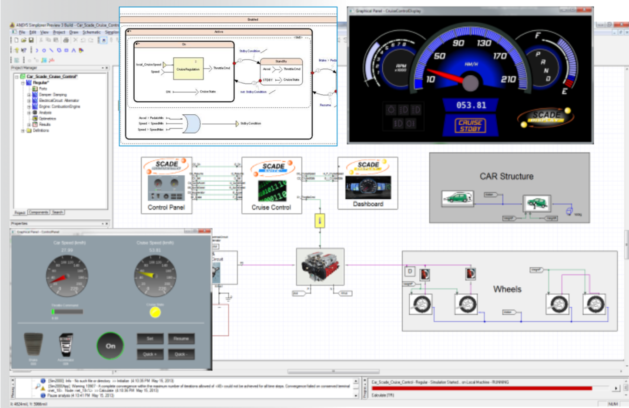
Lustre décrit des modèles à temps discret.

Quid de modèle hybrides combinant temps discret/temps continu?

Un exemple de système hybride

Modéliser l'ensemble du système: le contrôleur et son environnement[3]

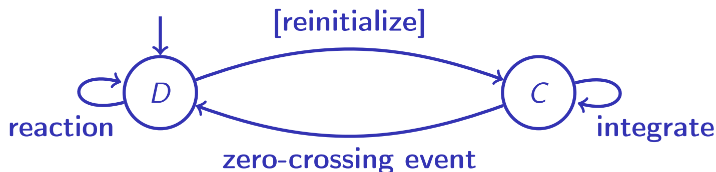
Modèle hybride: temps discret et continu



[3]Image de ANSYS/Esterel-Technologies

Si on programmait tout ça à la main?

Alterner pas discrets et intégration numérique



$$\sigma', y' = next_\sigma(t, y) \qquad upz = g_\sigma(t, y) \qquad \dot{y} = f_\sigma(t, y)$$

Pour un langage de haut niveau, le compilateur doit produire:

- $next_\sigma$ rassemble les changements discrets.
- g_σ définit les signaux de zero-crossing.
- f_σ est la fonction à intégrer.

Garantir que f et g sont sans effets de bord? continus?

DEMO

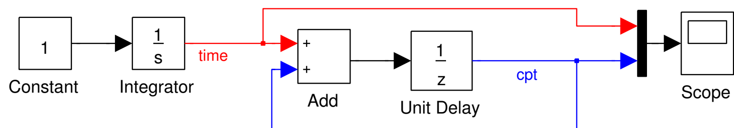
Zélus

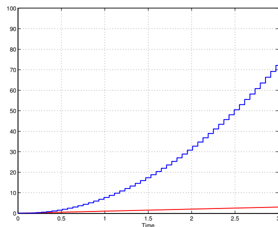
Zélus = Lustre + ODEs + zero crossings

Où sont les monstres?

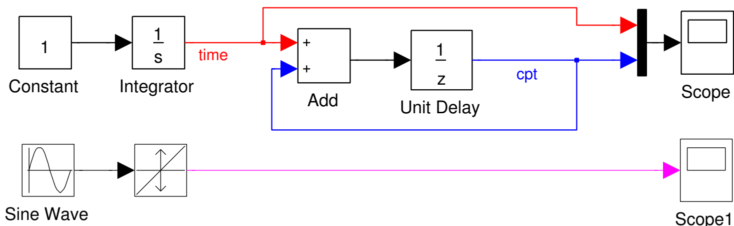
Certains modèles mélangent le temps discret logique et le temps continu de manière ambigue ou fausse

Que fait Simulink?

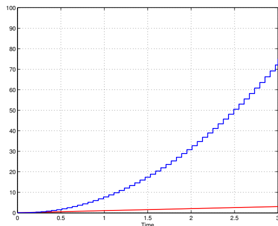




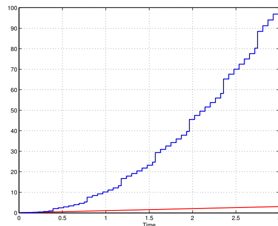
Que fait Simulink?



Basic model



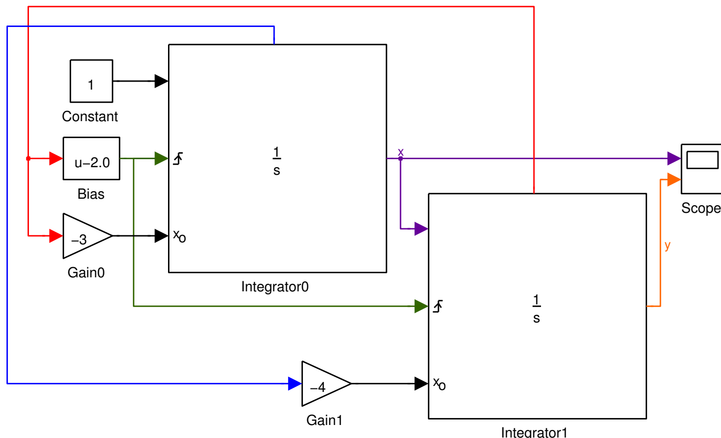
with Sine Wave



Le temps discret n'est pas le temps logique de Lustre.

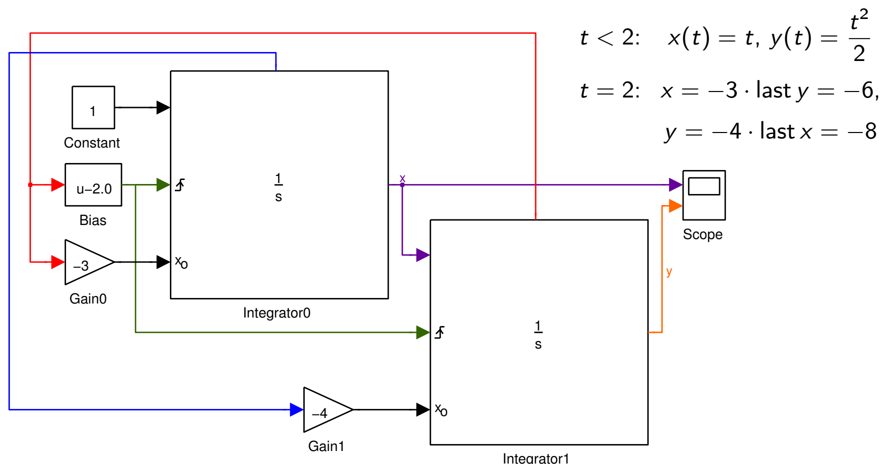
C'est celui de la mécanique de simulation.

Causality issue: the Simulink state port



The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.
–Simulink Reference (2-685)

Causality issue: the Simulink state port



$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

$$y = -4 \cdot \text{last}\, x = -8$$

The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.
–Simulink Reference (2-685)

Causality issue: the Simulink state port

$$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$$

$$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$$

$$y = -4 \cdot \text{last}\, x = -8$$

1

Constant

u–2.0

Bias

–3

Gain0

$\frac{1}{s}$

x_0

Integrator0

x

Scope

y

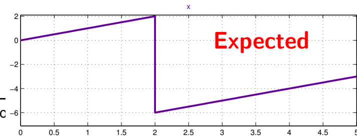
$\frac{1}{s}$

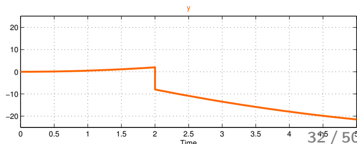
–4

Gain1

x_0

Integrato

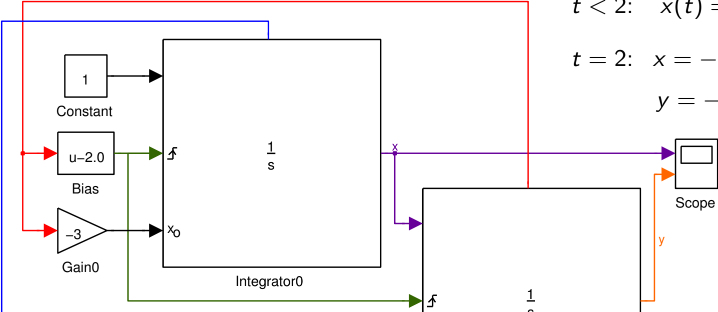




The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.

–Simulink Reference (2-685)

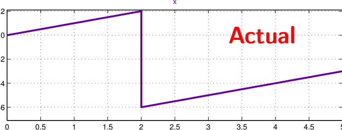
Causality issue: the Simulink state port

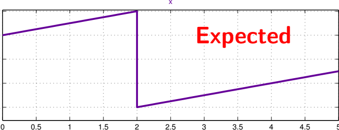


$t < 2: \quad x(t) = t,\ y(t) = \frac{t^2}{2}$

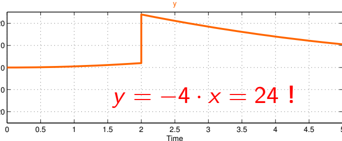
$t = 2: \quad x = -3 \cdot \text{last}\, y = -6,$

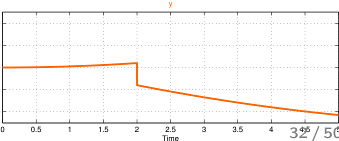
$y = -4 \cdot \text{last}\, x = -8$





The output of the st
the block's standard
case. If the block is r
put of the state port
at the block's standa
reset.
–Simulink Reference





Les mêmes en Zélus

```
let hybrid wrong() = (time, cpt) where
  rec
    der time = 1.0 init 0.0
  and
    cpt = 0.0 fby (time +. cpt)

>    cpt = 0.0 fby (time +. cpt)
>          ^^^^^^^^^^^^^^^^^^^^^
Type error: this is a discrete expression
and is expected to be continuous.


let hybrid causal() = (x, y) where
  rec
    der x = 1.0 init 0.0 reset z -> -3.0 *. last y
  and
    der y = x init 0.0 reset z -> -4. *. last x
  and
    z = up(last y -. 2.0)
```

En bref

Ecrire des modèles synchrones (Lustre)

combinés avec des équations à temps continu (ODEs).

Rejeter statiquement les monstres.

Compiler vers du code séquentiel.

Utiliser un solveur d'ODEs sur étagère.

Principaux résultats

T. Bourke, A. Benveniste, B. Caillaud depuis 2010.
J.-L. Colaco, B. Pagano, C. Pasteur (ANSYS/Esterel-Tech), depuis 2013.

- Une sémantique idéale basée sur *l'analyse non standard* *[JCSS'12]*
- Lustre avec ODEs *[LCTES'11]*
- Automates hiérarchiques, temps discret et temps continu *[EMSOFT'11]*
- Analyse de causalité *[HSCC'14]*
- Génération de code séquentiel *[CC'15]*
- Librairie Simulink (discret, continu, hybride) *[EMSOFT'17]*

Implémenté dans Zélus *[HCSS'13]*

`http://zelus.di.ens.fr`

Solveur pas variable (SUNDIALS CVODE).

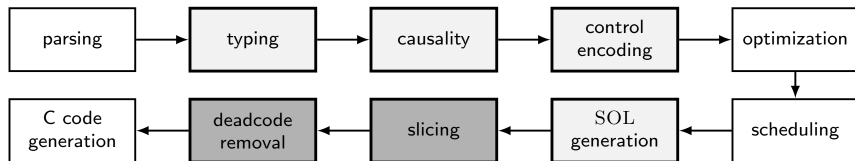
Comment ça marche? (les tripes du compilateur)

Architecture du compilateur

Deux implémentations: Zélus et KCG 6.4 (Release 2014).

SCADE Hybrid (basé sur KCG 6.4)

- Genère du FMI 1.0 (FMU) pour Simplorer.
- Seulement 5% de code en plus ou modifié. Petits changements dans:
  - analyse statiques (typage, causalité).
  - élimination des automates; génération de code.
  - Génération de FMUs (description XML, wrapper).
- boucle de simulation: environ 1000 LoC.



Un noyau à la SCADE

Essentiellement SCADE avec quelques extensions syntaxiques (en rouge).

$$
\begin{array}{lll}
d & ::= & \texttt{const}\ x = e \mid k\ f(pi) = pi\ \texttt{where}\ E \mid d; d \\
k & ::= & \texttt{fun} \mid \texttt{node} \mid \texttt{hybrid} \\
e & ::= & x \mid v \mid op(e, ..., e) \mid v\ \texttt{fby}\ e \mid \texttt{last}\ x \mid f(e, ..., e) \mid \texttt{up}(e) \\
p & ::= & x \mid (x, ..., x) \\
pi & ::= & xi \mid xi, ..., xi \\
xi & ::= & x \mid x\ \texttt{last}\ e \mid x\ \texttt{default}\ e \\
E & ::= & p = e \mid \texttt{der}\ x = e \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ E\ \texttt{else}\ E \\
 & & \mid \texttt{reset}\ E\ \texttt{every}\ e \\
 & & \mid \texttt{local}\ pi\ \texttt{in}\ E \mid \texttt{do}\ E\ \texttt{and} \ldots E\ \texttt{done}
\end{array}
$$

Un langage intermédiaire data-flow

Le langage est étendu avec trois opérations nouvelles. Traduction basée sur Colaco et al. [EMSOFT'05].

$$
\begin{array}{rcl}
d & ::= & \texttt{const}\, x = c \mid k\, f(p) = a\, \texttt{where}\, C \mid d; d \\
k & ::= & \texttt{fun} \mid \texttt{node} \mid \texttt{hybrid} \\
C & ::= & (x_i \ = \ a_i)_{x_i \in I} \text{ with } \forall i \neq j. x_i \neq x_j \\
a & ::= & e^{ck} \\
e & ::= & x \mid v \mid op(a, ..., a) \mid v \ \texttt{fby}\ a \mid \texttt{pre}(a) \\
 & & \mid f(a, ..., a) \\
 & & \mid \texttt{merge}(a, a, a) \mid a \ \texttt{when}\ a \\
 & & \mid \texttt{integr}(a, a) \mid \texttt{up}(a) \\
p & ::= & x \mid (x, ..., x) \\
ck & ::= & \texttt{base} \mid ck \ \texttt{on}\ a
\end{array}
$$

Mise en forme normale des équations

Nommer le résultat de chaque opération à mémoire. Les séparer en trois catégories.

- *se*: expressions strictes
- *de*: délais
- *ce*: expressions gardées par une condition (horloge)

L'équation $lx = \texttt{integr}(x', x)$ définie lx en tant que variable d'état continue; possiblement réinitialisée en x.

$$
\begin{array}{lcl}
eq & ::= & x = ce^{ck} \mid x = f(sa, ..., sa)^{ck} \mid x = de^{ck} \\
sa & ::= & se^{ck} \\
ca & ::= & ce^{ck} \\
se & ::= & x \mid v \mid op(sa, ..., sa) \mid sa \texttt{ when } sa \\
ce & ::= & se \mid \texttt{merge}(sa, ca, ca) \mid ca \texttt{ when } sa \\
de & ::= & \texttt{pre}(ca) \mid v \texttt{ fby } ca \mid \texttt{integr}(ca, ca) \mid \texttt{up}(ca)
\end{array}
$$

Ordonnancement statique des équations

$Read(a)$: variables lues par a.

Etant donné $C = (x_i = a_i)_{x_i \in I}$, un ordonnancement est une bijection:

$$Schedule(.) : I \rightarrow \{1 \ldots |I|\}$$

telle que, pour tout $x_i \in I, x_j \in Read(a_i) \cap I$:

1. si a_i est stricte, $Schedule(x_j) < Schedule(x_i)$ et
2. si a_i est un délai, $Schedule(x_i) \leq Schedule(x_j)$.

Du point-de-vue des dépendances, `integr`(ca_1, ca_2) and `up`(ca) coupe une boucle instantanée.

Un langage objet interne (SOL)

- Traduction du source vers ce langage intermédiaire [Colaco et al., LCTES'08]
- Au lieu de produire deux méthodes `step` and `reset`, en produire plus.
- Marquer les mémoires avec une sorte m

$$
\begin{array}{lll}
md & ::= & \mid \texttt{const}\, x = c \\
 & & \mid \texttt{const}\, f = \texttt{class}\langle M,\ I,\ (method_i(p_i) = e_i\, \texttt{where}\, S_i)_{i \in [1..n]} \rangle \\
M & ::= & [x : m[= v]; ...; x : m[= v]] \\
I & ::= & [o : f; ...; o : f] \\
m & ::= & Discrete \mid Zero \mid Cont \\
e & ::= & v \mid lv \mid op(e, ..., e) \mid o.method(e, ..., e) \\
S & ::= & () \mid lv \leftarrow e \mid S \,;\, S \mid \texttt{var}\, x, ..., x\, \texttt{in}\, S \mid \texttt{if}\, c\, \texttt{then}\, S\, \texttt{else}\, S \\
R, L & ::= & S; ...; S \\
lv & ::= & x \mid lv.field \mid \texttt{state}\,(x)
\end{array}
$$

Variables d'état

Variables d'état discrètes (sorte *Discrete*)

- Lue avec `state` (x);
- modifiée avec `state` $(x) \leftarrow c$

Variables de zero-crossing (sorte *Zero*)

- Une paire avec deux champs.
- Le champ `state` $(x).zin$ est un booléen, vrai lorsqu x a un zero-crossing, faux sinon.
- Le champ `state` $(x).zout$ est la valeur à observer.

Variables d'état continues (sorte *Cont*)

- `state` $(x).der$ est la dérivée instantanée;
- `state` $(x).pos$ sa valeur

Exemple: la balle qui rebondit

```
let bouncing = machine(continuous) {
  memories disc init_25 : bool = true;
           zero result_17 : bool = false;
           cont y_v_15 : float = 0.; cont y_14 : float = 0.

  method reset  =
    init_25 <- true; y_v_15.pos <- 0.

  method step time_23 y0_9 =
    (if init_25 then (y_14.pos <- y0_9; ()) else ());
    init_25 <- false;
    result_17.zout <- (~-.) y_14.pos;
    if result_17.zin
     then (y_v_15.pos <- ( *. ) 0.8 y_v_15.pos);
    y_14.der <- y_v_15.pos;
    y_v_15.der <- (~-.) g; y_14.pos }
```

Enfin

1. Générer, comme d'habitude, une fonction `step`.
2. Pour les noeuds hybrides, **copy-et-coller** la méthode `step`.
3. Produire, une méthode `cont` activée pendant l'intégration, ou deux méthodes `derivatives` et `crossings`.
4. Simplifier ainsi:
   - Pendant l'intégration (method `cont`), les variables de zero-crossing (variables de type *zero*, e.g., `state` $(x).zin$) sont fausses. Les sorties des zero-crossings (`state` $(x).zout \leftarrow \ldots$) sont inutiles.
   - Pendant le pas discret (method `step`), les dérivées (`state` $(x).der \leftarrow \ldots$) sont inutiles.
   - Eliminer le code mort.
5. C'est tout !

Exemples (en Zélus et SCADE) ici: `zelus.di.ens.fr/cc2015`

Exemple: la balle qui rebondit

```
let bouncing = machine(continuous) {
  memories disc init_25 : bool = true;
           zero result_17 : bool = false;
           cont y_v_15 : float = 0.; cont y_14 : float = 0.
  method reset  =
    init_25 <- true; y_v_15.pos <- 0.
  method step time_23 y0_9 =
    (if init_25 then (y_14.pos <- y0_9; ()) else ());
    init_25 <- false;
    if result_17.zin
     then (y_v_15.pos <- ( *. ) 0.8 y_v_15.pos);
    y_14.pos
  method cont time_23 y0_9 =
    result_17.zout <- (~-.) y_14.pos;
    y_14.der <- y_v_15.pos;
    y_v_15.der <- (~-.) g }
```

Conclusion

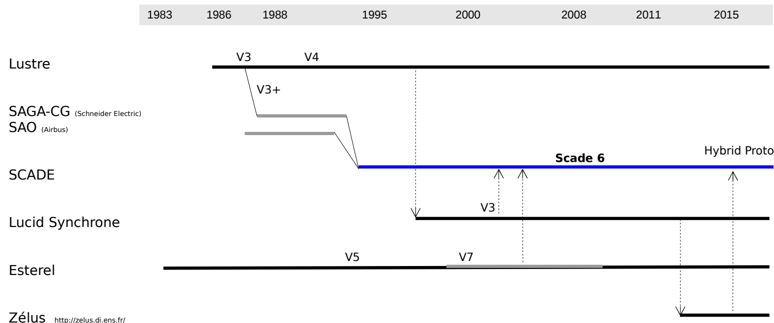
Deux expériences

- Le langage **Zélus** et son compilateur.
- Un prototype industriel SCADE Hybrid (**KCG 6.7**)
- Pour KCG, **moins de 5%** de LOC en plus.
- Extension conservative w.r.t SCADE.
- Utiliser le même code pour la simulation et le code embarqué.
- Un nouveau produit par ANSYS.

Quid d'une librarie?

- Comme en Lustre, faire en sorte que "le programme est la spec"?
- Une librarie standard de blocs discrets/continus *[EMSOFT'17]*.
- Mais fait l'hypothèse d'un solveur idéal.
- Une sémantique paramétrée par celle d'une solveur particulier?

Timeline



Et le plus important pour la fin

Merci Nicolas!