A Synchronous-based Code Generator For Explicit Hybrid Systems Languages

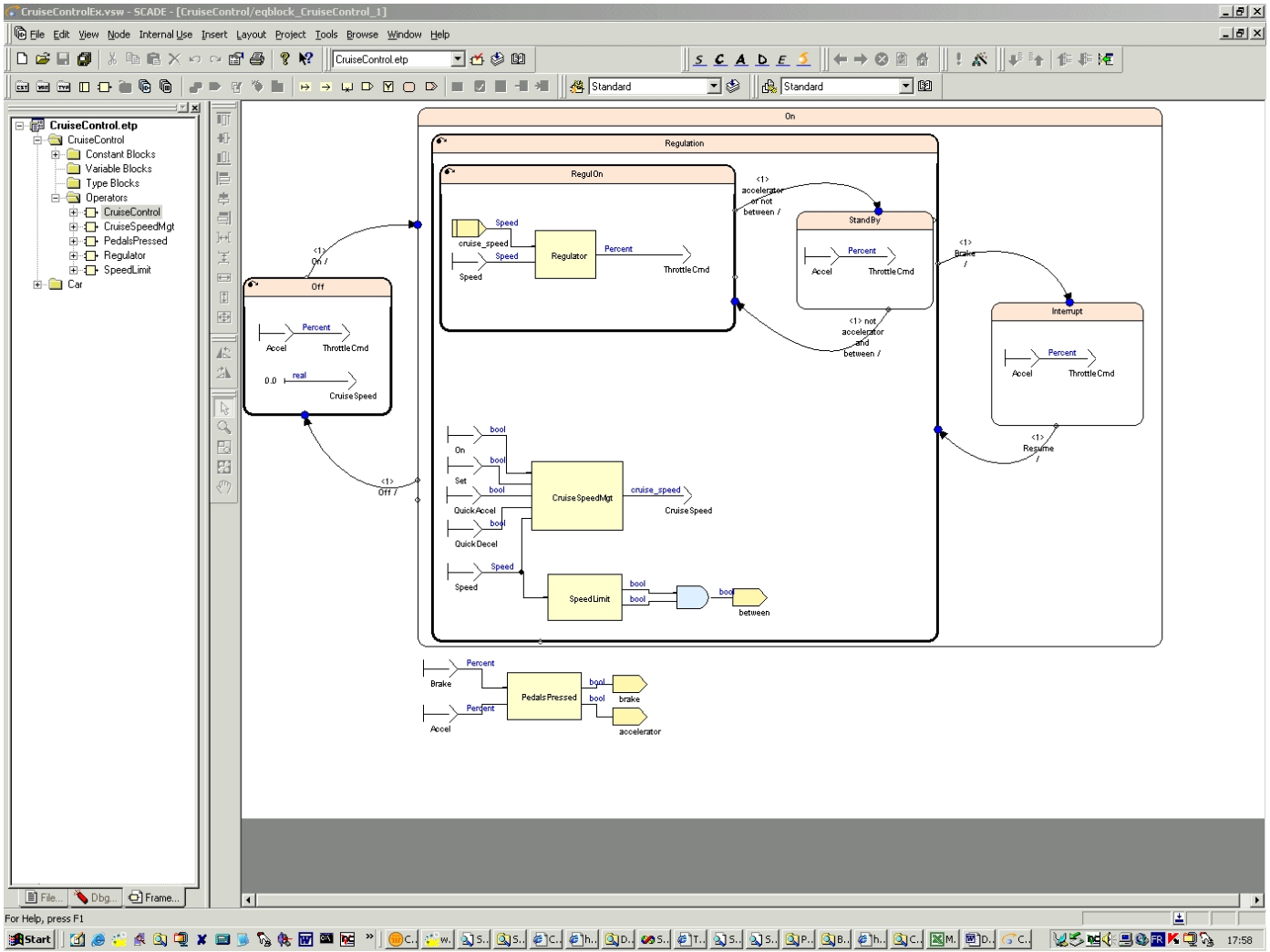
Timothy Bourke[1] Jean-Louis Colaço[2] Bruno Pagano[2]
Cédric Pasteur[2] Marc Pouzet[3,1]

1. INRIA Paris-Rocquencourt
2. Esterel-Technologies/ANSYS, Toulouse
3. DI, École normale supérieure, Paris

CC'2015
London, ETAPS
April 17, 2015

Synchronous Block Diagram Languages: SCADE

- Widely used for critical control software development;
- E.g., avionic (Airbus, Ambraier, Comac, SAFRAN), trains (Ansaldo).

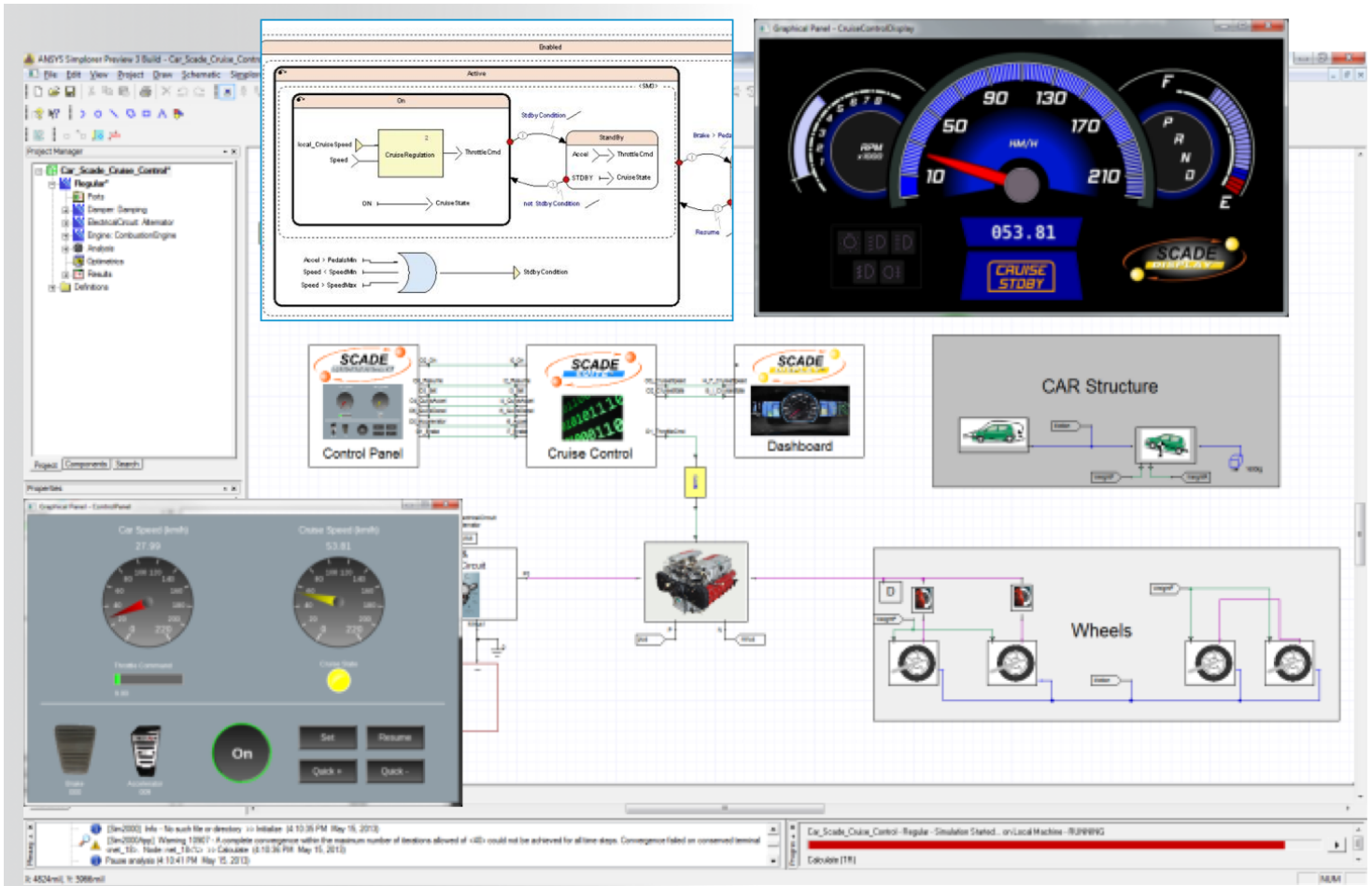


But modern systems need more. . .

The Current Practice of Hybrid Systems Modeling

Embedded software interacts with physical devices.

The whole system has to be modeled: the controller and the plant.[1]



[1] Image by Esterel-Technologies/ANSYS.

Current Practice and Objective

Current Practice

- Simulink, Modelica used to **model**, rarely to **implement** critical soft.
- Software must be reimplemented in SCADE or imperative code.
- Interconnect tools (Simulink+Modelica+SCADE+Simplorer+...)
- Interchange format for co-simulation: S-functions, FMU/FMI

Objective and Approach

- **Increase the confidence** in what is simulated
- Use SCADE both to simulate and implement
- Synchronous code for both the controller and the plant
- Reuse the existing compiler infrastructure
- Run with an off-the-shelf numerical solver (e.g., SUNDIALS)

Hybrid System Modelers

Simulink / FMI	Simplorer / Modelica
Ordinary differential equation	Differential algebraic equation
$\dot{y} = f(y, t)$	$f(y, \dot{y}, t) = 0$
Explicit	Implicit
Causal	Acausal

Hybrid System Modelers

Simulink / FMI / **Zélus** / **Scade Hybrid**	Simplorer / Modelica
Ordinary differential equation $\dot{y} = f(y, t)$	Differential algebraic equation $f(y, \dot{y}, t) = 0$
Explicit Causal	Implicit Acausal

Background: [Benveniste et al., 2010 - 2014]

"Build a hybrid modeler on synchronous language principles"

Milestones

- Do as if time was global and discrete [JCSS'12]
- `Lustre` with ODEs [LCTES'11]
- Hierarchical automata, both discrete and hybrid [EMSOFT'11]
- Causality analysis [HSCC'14]

This was experimented in the language Zélus [HCSS'13]

The validation on an industrial compiler remained to be done.

SCADE Hybrid (summer 2014)

- Prototype based on KCG 6.4 (last release)
- SCADE Hybrid = full SCADE + ODEs
- Generates FMI 1.0 model-exchange FMUs with Simplorer

Synchronous languages in a slide

- Compose stream functions; basic values are streams.
- Operation apply pointwise + unit delay (fby) + automata.

```
(* computes [x(n) + y(n) + 1] at every instant [n] *)
fun add (x,y) = x + y + 1

(* returns [true] when the number of [t] has reached [bound] *)
node after (bound, t) = (c = bound) where
  rec c = 0 fby (min(tick, bound))
  and tick = if t then c + 1 else c
```

The counter can be instantiated twice in a two state automaton,

```
node blink (n, m, t) = x where
 automaton
 | On → do x = true   until (after(n, t)) then Off
 | Off → do x = false  until (after(m, t)) then On
```

From it, a synchronous compiler produces **sequential loop-free code** that compute a single **step** of the system.

A Simple Hybrid System

Yet, time was discrete. Now, a simple heat controller. [2]

```
(* a model of the heater defined by an ODE with two modes *)
hybrid heater(active) = temp where
  rec der temp = if active then c -. k *. temp else -. k *. temp init temp0

(* an hysteresis controller for a heater *)
hybrid hysteresis_controller(temp) = active where
  rec automaton
      | Idle -> do active = false until (up(t_min -. temp)) then Active
      | Active -> do active = true until (up(temp -. t_max)) then Idle

(* The controller and the plant are put parallel *)
hybrid main() = temp where
  rec active = hysteresis_controller(temp)
  and temp = heater(active)
```

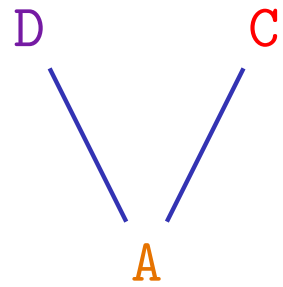
Three syntactic novelties: keyword **hybrid**, **der** and **up**.

[2] Hybrid version of N. Halbwachs's example in Lustre at Collège de France, Jan.10.

From Discrete to Hybrid

The type language [LCTES'11]

$$
\begin{array}{lcl}
bt & ::= & \texttt{float} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{zero} \mid \cdots \\
\sigma & ::= & bt \times \ldots \times bt \xrightarrow{k} bt \times \ldots \times bt \\
k & ::= & \texttt{D} \mid \texttt{C} \mid \texttt{A}
\end{array}
$$



Function Definition: `fun f(x1,...) = (y1,...)`

- **Combinatorial functions** (`A`); usable anywhere.

Node Definition: `node f(x1,...) = (y1,...)`

- **Discrete-time constructs** (`D`) of SCADE/Lustre: `pre`, `->`, `fby`.

Hybrid Definition: `hybrid f(x1,...) = (y1,...)`

- **Continuous-time constructs** (`C`): `der x = ...`, `up`, `down`, etc.

Mixing continuous/discrete parts

Zero-crossing events

- They correspond to event indicators/state events in FMI
- Detected by the solver when a given signal crosses zero

Design choices

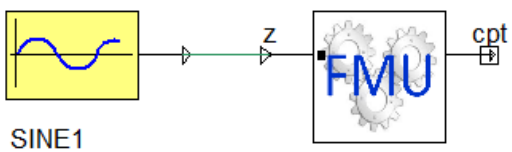
- A discrete computation can only be triggered by a zero-crossing
- Discrete state only changes at a zero-crossing event
- A continuous state can be reset at a zero-crossing event

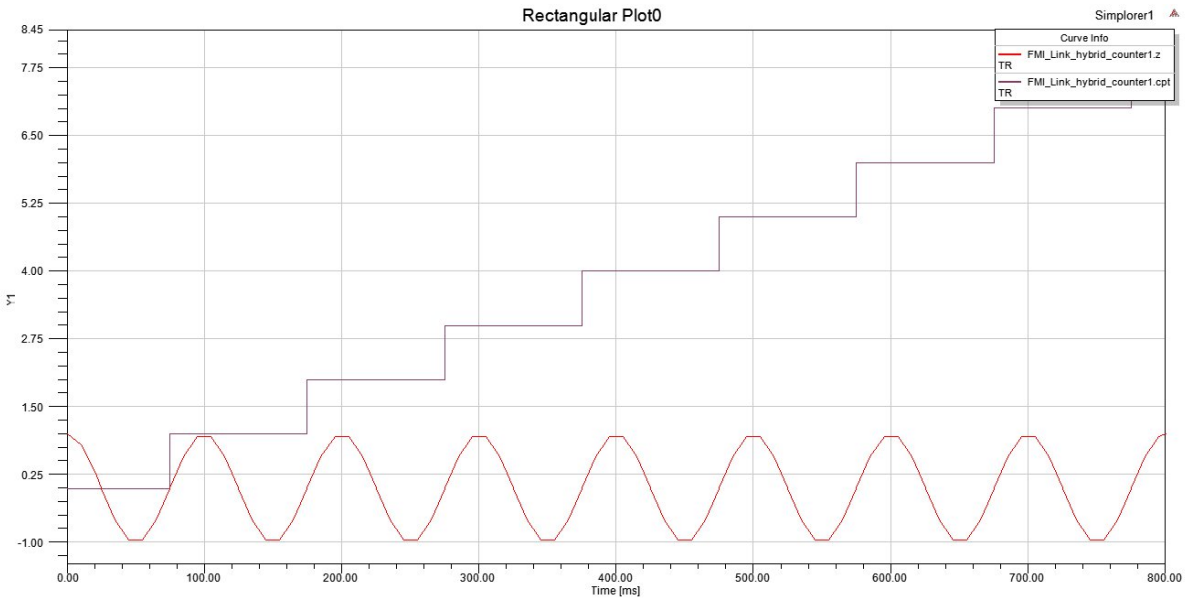
Example

```
node counter() = cpt where
  rec cpt = 1 → pre cpt + 1

hybrid hybrid_counter() = cpt where
 rec cpt = present up(z) → counter() init 0
 and z = sinus()
```

Output with SCADE Hybrid + Simplorer





How to communicate between continuous and discrete time?

E.g., the bouncing ball

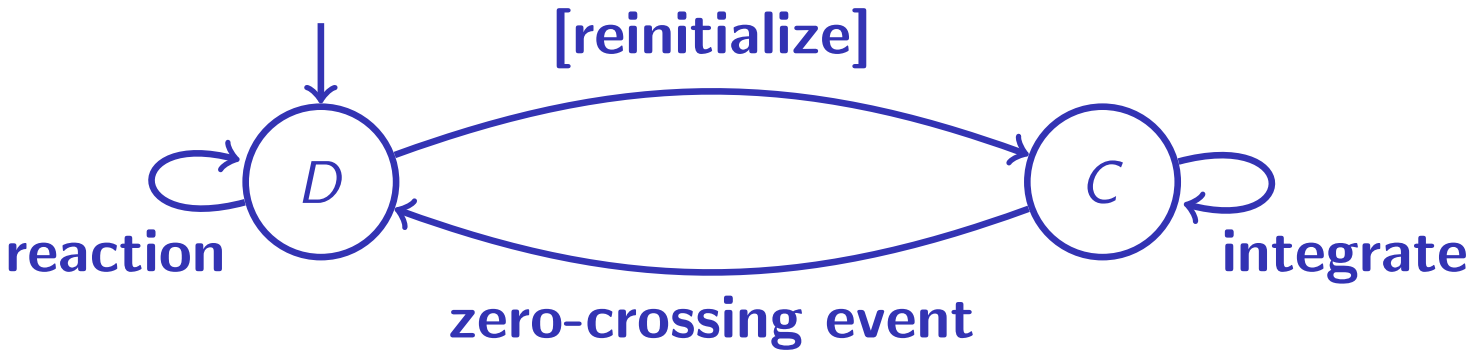
```
hybrid ball(y0) = y where
 rec der y = y_v init y0
 and der y_v = −. g init 0.0 reset z → 0.8 ∗. last y_v
 and z = up(−. y)
```

- Replacing last y_v by y_v would lead to a deadlock.
- In SCADE and Zélus, last y_v is the previous value of y_v.
- It coincides with the **left limit** of y_v when y_v is left continuous.

Internals

The Simulation Engine of Hybrid Systems

Alternate discrete steps and integration steps



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

Properties of the three functions

- $next_\sigma$ gathers all discrete changes.
- g_σ defines signals for zero-crossing detection.
- f_σ is the function to integrate.

Compilation

The Compiler has to produce:

1. Inititialization function *init* to define $y(0)$ and $\sigma(0)$.
2. Functions f and g.
3. Function *next*.

The Runtime System

1. Program the simulation loop, using a black-box solver (e.g., SUNDIALS CVODE);
2. Or rely on an existing infrastructure.

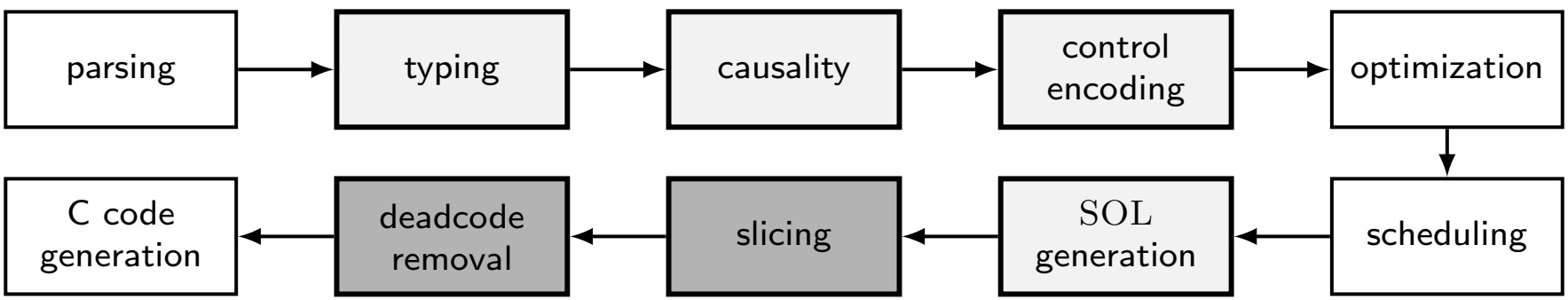
`Zélus` follows (1); `SCADE Hybrid` follows (2), targetting Simplorer FMIs.

Compiler Architecture

Two implementations: `Zélus` and KCG 6.4 (Release 2014) of SCADE.

KCG 6.4 of SCADE

- Generates FMI 1.0 model-exchange FMUs for Simplorer.
- Only 5% of the compiler modified. Small changes in:
  - static analysis (typing, causality).
  - automata translation; code generation.
  - FMU generation (XML description, wrapper).
- FMU integration loop: about 1000 LoC.



A SCADE-like Input Language

Essentially SCADE with three syntax extensions (in red).

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

A Clocked Data-flow Internal Language

The internal language is extended with three extra operations.
Translation based on Colaco et al. [EMSOFT'05].

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

Clocked Equations Put in Normal Form

Name the result of every stateful operation. Separate into syntactic categories.

- *se*: strict expressions
- *de*: delayed expressions
- *ce*: controlled expressions.

Equation $lx = \texttt{integr}(x', x)$ defines lx to be the continuous state variable; possibly reset with x.

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

Well Scheduled Form

Equations are statically scheduled.

$Read(a)$: set of variables read by a.

Given $C = (x_i = a_i)_{x_i \in I}$, a valid schedule is a one-to-one function

$$Schedule(.) : I \to \{1 \dots |I|\}$$

such that, for all $x_i \in I, x_j \in Read(a_i) \cap I$:

1. if a_i is strict, $Schedule(x_j) < Schedule(x_i)$ and
2. if a_i is delayed, $Schedule(x_i) \leq Schedule(x_j)$.

From the data-dependence point-of-view, $\texttt{integr}(ca_1, ca_2)$ and $\texttt{up}(ca)$ break instantaneous loops.

A Sequential Object Language (SOL)

- Translation into an intermediate imperative language [Colaco et al., LCTES'08]
- Instead of producing two methods `step` and `reset`, produce more.
- Mark memory variables with a kind *m*

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

State Variables

Discrete State Variables (sort *Discrete*)

- Read with `state`(x);
- modified with `state`$(x) \leftarrow c$

Zero-crossing State Variables (sort *Zero*)

- A pair with two fields.
- The field `state`$(x).zin$ is a boolean, true when a zero-crossing on x has been detected, false otherwise.
- The field `state`$(x).zout$ is the value for which a zero-crossing must be detected.

Continuous State Variables (sort *Cont*)

- `state`$(x).der$ is its instantaneous derivative;
- `state`$(x).pos$ its value

Example: translation of the bouncing ball

```
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 }
```

Finally

1. Translate as usual to produce a function `step`.
2. For hybrid nodes, **copy-and-paste** the step method.
3. Either into a `cont` method activated during the continuous mode, or two extra methods `derivatives` and `crossings`.
4. Apply the following:
   - During the continuous mode (method `cont`), all zero-crossings (variables of type *zero*, e.g., `state`$(x).zin$) are surely false. All zero-crossing outputs (`state`$(x).zout \leftarrow \ldots$) are useless.
   - During the discrete step (method `step`), all derivative changes (`state`$(x).der \leftarrow \ldots$) are useless.
   - Remove dead-code by calling an existing pass.
5. That's all!

Examples (both `Zélus` and `SCADE`) at: `zelus.di.ens.fr/cc2015`

Conclusion

Two full scale experiments

- The `Zélus` academic langage and compiler.
- The industrial **KCG 6.4** (Release 2014) code generator of SCADE.
- For KCG, **less than 5%** of extra LOC, in all.
- The extension is **fully conservative** w.r.t existing SCADE.

Lessons

- The existing compiler infrastructure of SCADE, based on successive rewritting, helped a lot.
- Synchronous languages principles are useful to build a real hybrid systems modeling language.

Yet, doing the same for ODEs + constraints (DAEs) is far less clear.

Compiler

Zélus is a synchronous language extended with Ordinary Differential Equations (ODEs) to model systems with complex interaction between discrete-time and continuous-time dynamics. It shares the basic principles of Lustre with features from Lucid Synchrone (type inference, hierarchical automata, and signals). The compiler is written

Research

Zélus is used to experiment with new techniques for building hybrid modelers like Simulink/Stateflow and Modelica on top of a synchronous language. The language exploits novel techniques for defining the semantics of hybrid modelers, it provides dedicated type systems to ensure the absence of discontinuities during integration and the