A Synchronous Look at the Simulink Standard Library

Timothy Bourke[1,2] François Carcenac[4] Jean-Louis Colaço[4]

Bruno Pagano[4] Cédric Pasteur[4] Marc Pouzet[3,2,1]

1. Inria Paris
2. DI, École normale supérieure
3. Univ. Pierre et Marie Curie
4. ANSYS/Esterel-Technologies

EMSOFT – Seoul – October 2017

Paper at: `http://www.di.ens.fr/~pouzet/bib/emsoft17.pdf`

A hybrid system = control software + physical model

Mixed signals (discrete + continuous)

A trend in building safe and complex embedded software

The "Model Based Design" motto

Write an executable deterministic model in a mathematical language used as:

A reference semantics independent of any implementation.

A basis for simulation, testing, formal verification.

Compiled into executable code, sequential or parallel

with semantics preservation all along the chain.

A way to achieve correct-by-construction software

Domain Specific Languages (DSL)

Directly write the mathematical models of the control scientist/engineer.

Difference and stream equations,

hierarchical finite state machines,

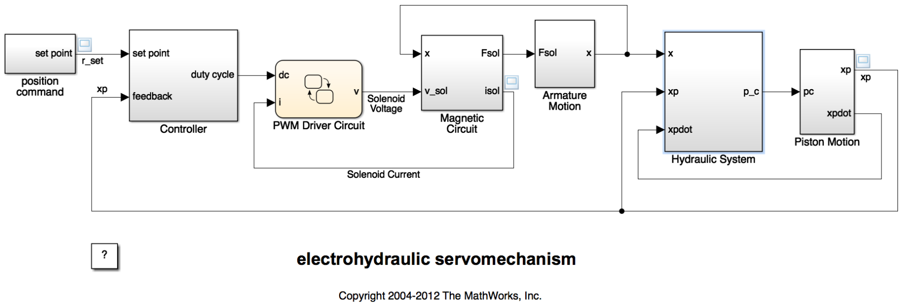
differential equations (ODEs),

composed with deterministic synchronous parallelism,

$+$ imperative programming constructs for the algorithmic part.

A representative of this trend is Simulink

Simulink Model [1]



The model is used for simulation (off-line/on-line), automatic testing, formal verification and code generation.

The compiler has a central role.

[1] Image taken from the standard distribution of Simulink

Study it from a PL perspective

Which models make sense?

Which should be statically rejected?

How to ensure determinacy?

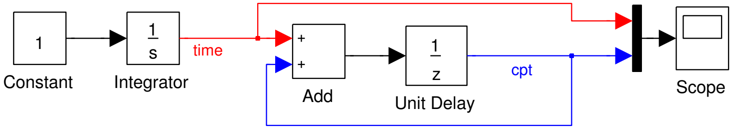
How to ensure that compilation is correct?

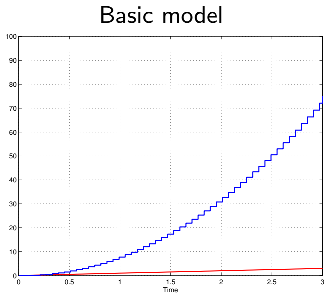
Some models mix discrete logical time and continuous time in an undisciplined manner

They are wrongly typed.

Typing Issues

Dubious compositions of discrete and continuous time: statically reject?

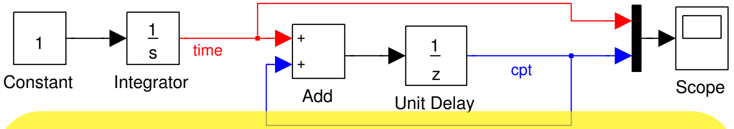




- The value of cpt depends on the steps chosen by the solver

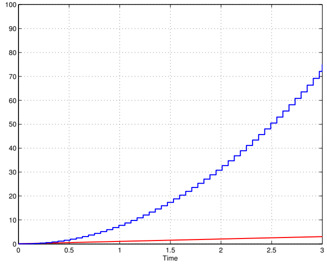
Typing Issues

Dubious compositions of discrete and continuous time: statically reject?



discrete time is that of the simulation engine

Basic model



- The value of cpt depends on the steps chosen by the solver

Wrongly typed models

Design type systems to statically reject bizarre models.

Can we formally ensure a property like:

“Well typed programs cannot go wrong” (Robin Milner) ?

What is a wrong model/program?

To study those questions, define a minimalistic language,

consider only constructs for which the semantics is precisely defined,

together with typing constraints to ensure safety properties.

Build a Hybrid Modeler on Synch. Language Principles

Milestones

- An ideal semantics based on *non standard analysis* *[JCSS'12]*
- `Lustre` with ODEs; typing discrete/continuous *[LCTES'11]*
- Hierarchical automata, both discrete and hybrid *[EMSOFT'11]*
- Causality analysis *[HSCC'14]*; Sequential code generation *[CC'15]*

Implemented in Zélus *[HCSS'13]*

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

Simulate with an off-the-shelf solver: SUNDIALS CVODE from LLNL

SCADE Hybrid = SCADE + ODEs/Xcrossings

- Prototype based on KCG 6.4 (now 6.6) at ANSYS/Esterel-Tech.

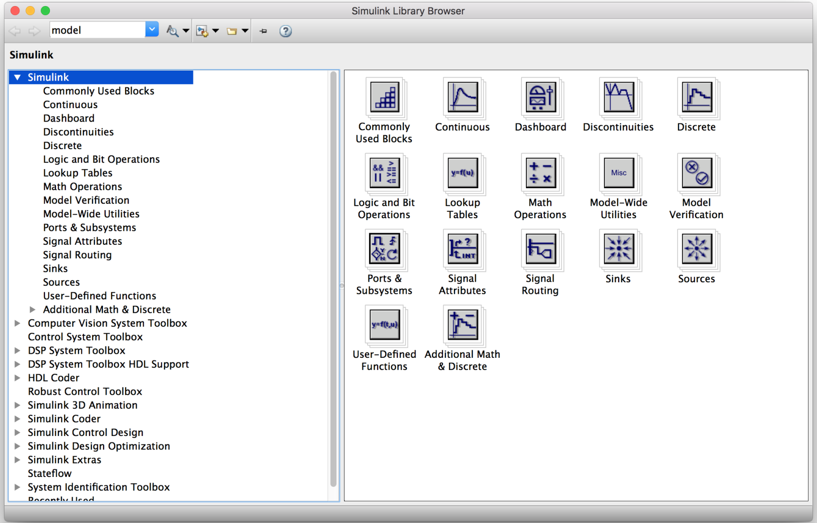
Yet, is that enough to program a comprehensive library

of discrete/continuous-time control blocks,

e.g., those of Simulink, so that

the program is the formal specification?

The Simulink Standard Library



- Simulink
  - Commonly Used Blocks
  - Continuous
  - Dashboard
  - Discontinuities
  - Discrete
  - Logic and Bit Operations
  - Lookup Tables
  - Math Operations
  - Model Verification
  - Model-Wide Utilities
  - Ports & Subsystems
  - Signal Attributes
  - Signal Routing
  - Sinks
  - Sources
  - User-Defined Functions
  - Additional Math & Discrete
- Computer Vision System Toolbox
- Control System Toolbox
- DSP System Toolbox
- DSP System Toolbox HDL Support
- HDL Coder
- Robust Control Toolbox
- Simulink 3D Animation
- Simulink Coder
- Simulink Control Design
- Simulink Design Optimization
- Simulink Extras
- Stateflow
- System Identification Toolbox
- Recently Used

A comprehensive set of blocks, some being the composition of simpler ones, some being directly implemented in C;

described through an informal documentation.

An experiment with `Zélus` and `SCADE Hybrid`.

Combinational Blocks

E.g., Math operations, Logic and bit operations, look-up tables.

Essentially Lustre or SCADE: data-flow equations + external functions.

```
let fun half(a, b) = (s, co)
  where
   rec s = if a then not b else b
   and co = a & b

let fun adder(c, a, b) = (s, co)
  where
   rec (s1, c1) = half(a, b)
   and (s, c2) = half(c, s1)
   and co = c1 or c2

val half : bool * bool -A-> bool * bool
val adder : bool * bool * bool -A-> bool * bool
```

The type $t_1 \xrightarrow{A} t_2$ for f means that $f(x)$ is executed at every instant.

Other are written similarly.

Combinatorial Blocks: Lookup tables

Typically programmed in the host language (e.g., C, Matlab).

Can we express that the size of the array is statically fixed?

A function f with type $t_1 \xrightarrow{S} t_2$ means that $f(x)$ must be a static value.

```
val lut1D : (l: int) -S-> float[l] -S-> float -A-> float

val lut2D : (l1: int) -S-> (l2: int)
                      -S-> float[l1][l2]
                      -S-> float * float -A-> float
```

Arrays and Loops

The for loop is borrowed from the SISAL [2] language.

```
let sum(l)(x, y) = z where
  rec
    forall i in 0 .. l - 1, xi in x, yi in y, zi out z
      do
       zi = xi + yi
      done

val sum : (l:int) -S-> int[l] * int[l] -A-> int[l]
```

The equation zi $=$ xi $+$ yi means for all $i \in [0..l-1]$:

$$z(i) \quad = \quad x(i) + y(i)$$

That is for all $i \in [0..l-1]$, for all $n \in \mathbb{N}$:

$$z(i)_n \quad = \quad x(i)_n + y(i)_n$$

[2] "Streams and Iteration in a Single Assignment Language", by McGraw et al.

Accummulator

```
let scalar(l)(x, y) = acc where
  rec forall i in 0 .. l - 1, xi in x, yi in y
        do
          acc = (xi * yi) + last acc
        initialize
          last acc = 0.0
        done

val scalar : (l: int) -S-> float array[l] * float array[l]
                      -A-> float
```

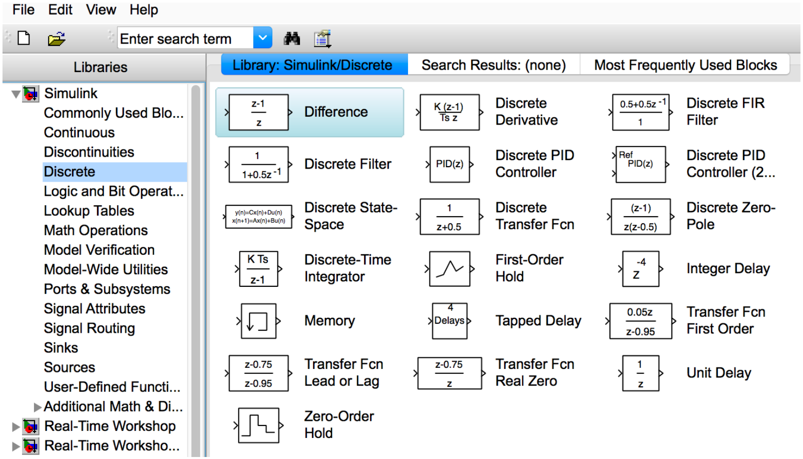
The equation acc $=$ (xi $*.$ yi) $+.$ **last** acc stands for:

$$\begin{array}{lcl} acc(i) & = & (x(i) * y(i)) + acc(i-1) \text{ with } i \in [0..l-1] \\ acc(-1) & = & 0 \end{array}$$

and so, for all $n \in \mathbb{N}$ and $i \in [0..l-1]$:

$$\begin{array}{lcl} acc(i)_n & = & (x(i)_n * y(i)_n) + acc(i-1)_n \\ acc(-1)(n) & = & 0 \end{array}$$

Discrete-time Blocks



Simulink
- Commonly Used Blo...
- Continuous
- Discontinuities
- Discrete
- Logic and Bit Operat...
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems
- Signal Attributes
- Signal Routing
- Sinks
- Sources
- User-Defined Functi...
- Additional Math & Di...

Real-Time Workshop

Real-Time Worksho...

Block	Name	Block	Name	Block	Name
$\frac{z-1}{z}$	Difference	$\frac{K\,(z-1)}{Ts\;z}$	Discrete Derivative	$\frac{0.5+0.5z^{-1}}{1}$	Discrete FIR Filter
$\frac{1}{1+0.5z^{-1}}$	Discrete Filter	PID(z)	Discrete PID Controller	Ref PID(z)	Discrete PID Controller (2...
y(n)=Cx(n)+Du(n) x(n+1)=Ax(n)+Bu(n)	Discrete State-Space	$\frac{1}{z+0.5}$	Discrete Transfer Fcn	$\frac{(z-1)}{z(z-0.5)}$	Discrete Zero-Pole
$\frac{K\,Ts}{z-1}$	Discrete-Time Integrator		First-Order Hold	z^{-4}	Integer Delay
	Memory	4 Delays	Tapped Delay	$\frac{0.05z}{z-0.95}$	Transfer Fcn First Order
$\frac{z-0.75}{z-0.95}$	Transfer Fcn Lead or Lag	$\frac{z-0.75}{z}$	Transfer Fcn Real Zero	$\frac{1}{z}$	Unit Delay
	Zero-Order Hold				

Unit Delay (synchronous register)

1. $\forall i \in \mathbb{N}^*.(\mathtt{pre}(x))_i = x_{i-1}$ and $(\mathtt{pre}(x))_0 = \mathit{nil}$.
2. $\forall i \in \mathbb{N}^*.(x \,\mathtt{fby}\, y)_i = y_{i-1}$ and $(x \,\mathtt{fby}\, y)_0 = x_0$
3. $\forall i \in \mathbb{N}^*.(x \,\mathtt{->}\, y)_i = y_i$ and $(x \,\mathtt{->}\, y)_0 = x_0$

```
(* difference *)
let node diff(u) = o where
  rec o = u -. (u fby u)

val diff : float -D-> float
```

$$\begin{array}{llll} \forall i \in \mathbb{N}.\ o_i & = & (u - (u \,\mathtt{fby}\, u))_i & \\ & = & u_i - u_{i-1} & \text{if } i \geq 1 \\ & = & u_0 - u_0 = 0 & \text{otherwise} \end{array}$$

The type $t_1 \xrightarrow{D} t_2$ for f meands that $f(x)$ is discrete-time, that is, transforms sequences into sequences.

State space representation

The discrete-space representation is defined by:

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n) \end{aligned}$$

```
let node discrete_state_space(l)(m)(r)(x0)(a)(b)(c)(d)(u) = y
  where
  rec x = const(l)(x0) fby sum(l)(mvproduct(l)(l)(a,x),
                                  mvproduct(l)(m)(b,u))
  and y = sum(r)(mvproduct(r)(l)(c,x),
                 mvproduct(r)(m)(d,u))

let node discrete_state_space_0(l)(m)(r)(x0)(a)(b)(c)(d)(u) = y
  where
  rec x = const(l)(x0) fby sum(l)(mvproduct(l)(l)(a,x),
                                  mvproduct(l)(m)(b,u))
  and y = mvproduct(r)(l)(c,x)
```

A difficulty

According to the Simulink documentation, if $D = 0$, output y does not depend on input u.

The causality analysis of `Zélus` is unable to express this.

What solution? write two functions? do multi-stage (specialise the function at compile-time)? make a more expressive causality analysis?

Discrete-time blocks: the Integrator

E.g., forward/backward Euler, Trapezoidal, with state port, saturation.

```
let node forward_euler(t)(k, x0, u) = output where
  rec output = x0 fby (output +. (k *. t) *. u)

let node backward_euler(t)(k, x0, u) = output where
  rec output = x0 -> pre output +. (k *. t) *. u
```

The compiler computes type but also causality signatures:

They express how inputs and output depend on each other.

```
val forward_euler :
  {'a < 'b, 'c, 'd}. 'b -> 'c * 'a * 'd -> 'a
```

That is, `output` is available before `k`, `x0` and `u` are read.

```
val backward_euler : {}. 'a -> 'a * 'a * 'a -> 'a
```

Discrete-time PID

Transfer function:

$$C_{par}(z) = P + Ia(z) + D(\frac{N}{1 + Nb(z)})$$

Suppose int is the integration function; filter is the filtering function:

```
val int : float -S-> float * float * float -D-> float
val filter : float -S-> float -S-> float * float -D-> float
```

```
(* PID controller
 * p is the proportional gain; i the integral gain;
 * d the derivative gain;  n the filter coefficient *)

 let node pid_par(h)(n)(p, i, d, u) = c where
   rec c_p = p *. u
   and i_p = int(h)(i, 0.0, u)
   and c_d = filter(n)(h)(d, u)
   and c = c_p +. i_p +. c_d
```

When there is no filtering, the definition of filter is the derivative:

```
let node filter(n)(h)(k, u) = derivative(h)(k, u)
```

Otherwise, approximate it using a linear low pass filter:

```
  (* Apply a low pass filter on the input *)
  (* (see Astrom & Murray's book, 2008). *)
  let node filter(n)(h)(k, u) = udot where
    rec udot = n *. (k *. u -. f)
    and f = int(h)(n, 0.0, udot)
```

A Discrete-time PID as a higher-order function

```
let node generic_pid(int)(filter)(h)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run (int h)(i, 0.0, u)
  and c_d = run (filter h)(d, u)
  and c = c_p +. i_p +. c_d

let node pid_forward_no_filter(h)(p, i, d, u) =
  generic_pid(forward_euler)(derivative)(h)(p, i, d, u)

let node pid_backward_no_filter(h)(p, i, d, u) =
  generic_pid(backward_euler)(derivative)(h)(p, i, d, u)
```

The type for `generic_pid` is:

```
val generic_pid :
  ('a -S-> 'b * float * float -D-> float) -S->
    ('a -S-> 'c * float -D-> float) -S->
      'a -S-> float * 'b * 'c * float -D-> float
```

Conclusion: discrete-time blocks

Other blocks can be programmed in a similar manner.

A discrete-time version of the “discontinuous blocks” too.

Exercice all features of the language: data-flow equations, hierarchical automata, arrays, higher-order.

The program is very close to the mathematical specification.

A comprehensive library has been developed in SCADE in 2017.

Discrete-time blocks

This is not that surprising.

Since the work of Caspi et al. [3], several tools automatically translate a subset of Simulink discrete-time blocks into `Lustre`.

But they are mostly designed for model checking, targetting "flat" Lustre.

Efficiency/readability/modularity of the code has not been considered.

This experiment raises interesting PL questions:

Simulink blocks come in various forms with overloading of operations.

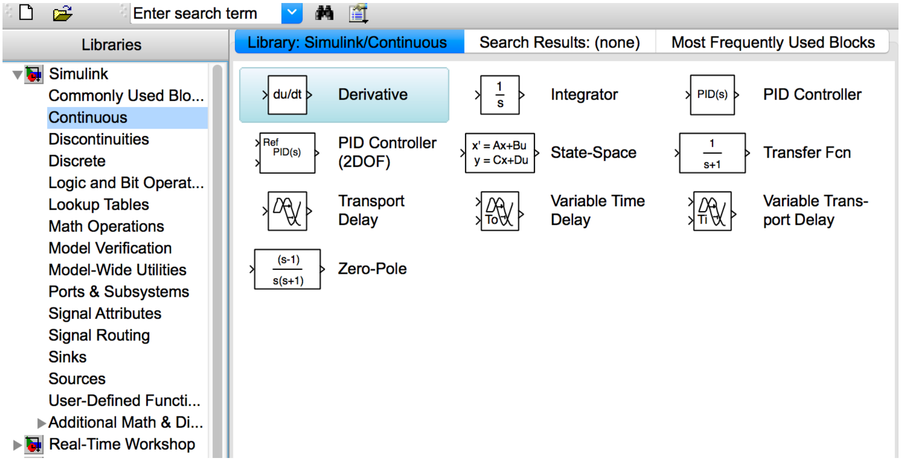
The strong typing discipline we impose is painful.

All static verifications are done on the function definition.

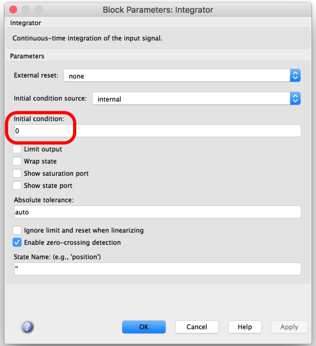
Should we do specialisation/macro-expansion before static typing?

[3] P. Caspi and A. Curic and A. Maignan and C. Sofronis and S. Tripakis. Translating Discrete-Time Simulink to Lustre, TECS'05

Continuous-time Blocks



The integrator block



Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: none

Initial condition source: internal

Initial condition:

0

- [ ] Limit output
- [ ] Wrap state
- [ ] Show saturation port
- [ ] Show state port

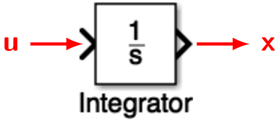
Absolute tolerance:

auto

- [ ] Ignore limit and reset when linearizing
- [x] Enable zero-crossing detection

State Name: (e.g., 'position')

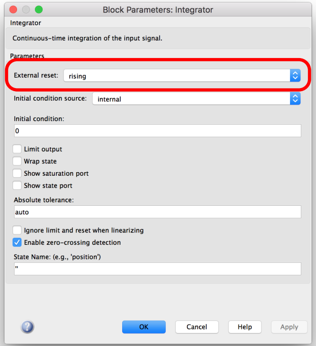
''



Basic form

```
let hybrid int(x0, u) = x where
  der x = u init x0

  val int :
    float * float -C-> float

  (* when [u] is an array *)
  let hybrid vint(n)(x0, u) = x where
    forall i in 0 .. (n - 1),
        x0i in x0, ui in u,
        xi out x do
        der xi = ui init x0i
    done

  val vint :
    (n_6:int) -S-> float[n_6] * float[n_6]
              -C-> float[n_6]
```

The integrator block


Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: rising

Initial condition source: internal

Initial condition:

0

- [ ] Limit output
- [ ] Wrap state
- [ ] Show saturation port
- [ ] Show state port

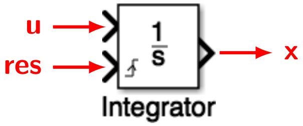
Absolute tolerance:

auto

- [ ] Ignore limit and reset when linearizing
- [x] Enable zero-crossing detection

State Name: (e.g., 'position')

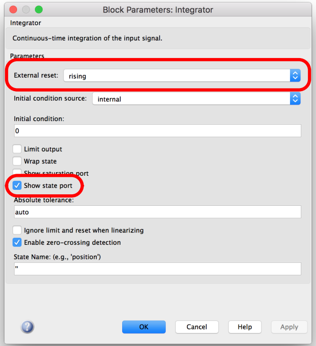
''



With reset port

```
let hybrid reset_int(x0, res, u) = x
where
  reset
    der x = u init x0
  every res
```

The integrator block


Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: rising

Initial condition source: internal

Initial condition:

0

☐ Limit output

☐ Wrap state

☐ Show saturation port

☑ Show state port

Absolute tolerance:

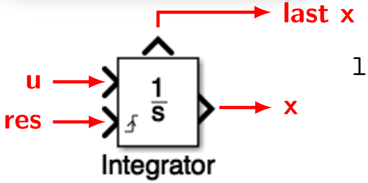
auto

☐ Ignore limit and reset when linearizing

☑ Enable zero-crossing detection

State Name: (e.g., 'position')

''

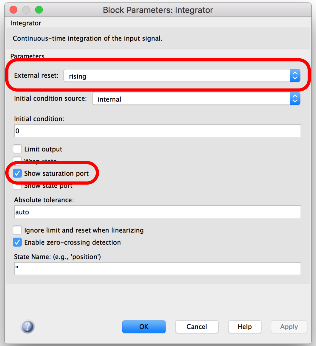


With reset and state ports

```
let hybrid reset_int(x0, res, u)
    = (x, last x)
where
  reset
    der x = u init x0
  every res
```

`last x` is the left limit of `x` *[HSCC'14]*

The integrator block



Block Parameters: Integrator

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: rising

Initial condition source: internal

Initial condition:

0

- [ ] Limit output
- [ ] Wrap state
- [x] Show saturation port
- [ ] Show state port

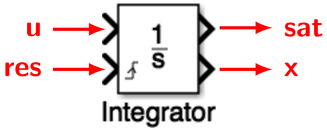
Absolute tolerance:

auto

- [ ] Ignore limit and reset when linearizing
- [x] Enable zero-crossing detection

State Name: (e.g., 'position')

''



With reset and saturation ports

```
let hybrid limit_int
  (k, y0, upper, lower, r, u)
  = (y, sat)
where
  rec
   reset
    init y = y0
    and automaton ... end
   every r
```

```
let hybrid limit_int(y0, upper, lower, r, u) = (y, sat)
  where rec reset
    init y = y0
   and automaton
       | BetweenState ->
           (* regular mode. Integrate the signal *)
           do der y = u and sat = Between
           unless up(y -. upper) then UpperState
           else up(-. (y -. lower)) then LowerState
       | UpperState ->
           (* when the input [u] is negative *)
           do y = upper and sat = Upper
           unless up(-. u) then BetweenState
       | LowerState ->
           (* when the input [u] is positive *)
           do y = lower and sat = Lower
           unless up(u) then BetweenState
       end
 every r
```

up(.) detects a zero-crossing.

Continuous-time PID

```
let hybrid pid_par(int)(filter)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run int(0.0, i *. u)
  and c_d = run filter(d *. u)
  and c = c_p +. i_p +. c_d

let hybrid pid(n)(p, i, d, u) =
  pid_par(Cint.int)(Cint.filter(n))(p, i, d, u)

let hybrid filter(n)(int)(k, u) = udot where
   rec udot = n *. (u -. f)
   and f = run int (0.0, k *. udot)
```

Types and causalities signatures are computed automatically.

State space representation

The continuous-time state-space representation is now:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

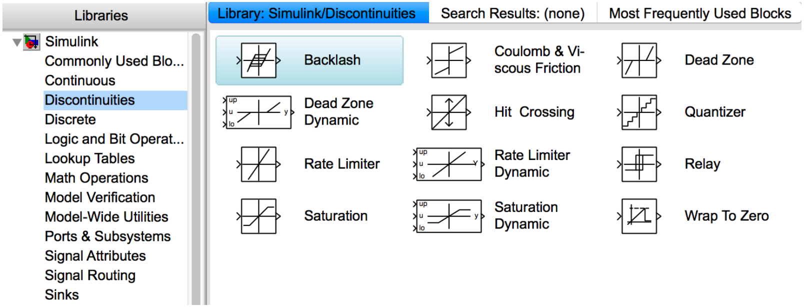
```
open Arrays
open Cint

let hybrid state_space(n)(m)(r)(x0)(a)(b)(c)(d)(u) = y where
  rec
      x = vint(n)(const(n)(x0), sum(n)(mvproduct(n)(n)(a, x),
                                         mvproduct(n)(m)(b, u)))
  and
      y = sum(r)(mvproduct(r)(n)(c, x), mvproduct(r)(m)(d, u))
```

The structure is the same as for the discrete-time version.

The previous remark about `d = 0` stay the same.

Other continuous-time blocks programmed similarly (see paper).

Discontinuous Blocks



Discontinuous blocks

Some blocks are relatively easy to program (see paper).

E.g., sign, coulomb friction, quantization, saturation, relay, comparison, dead-zone

Several use the zero-crossing detection (function `up(.)`).

```
(* Relay *)
  let hybrid relay(son, so , von, vo , u) = r where
  rec automaton
      | On -> do r = von unless up(so -. u) then Off
      | Off -> do r = vo  unless up(u -. son) then On
      end
```

`up(x)` detects when `x` goes from strictly negative to strictly positive.

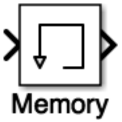
The language also provides periodic timers, that correspond to a particular form of zero-crossing (but compiled without it).

All discontinuous changes must be aligned on a zero-crossing or a timer.

The type system statically ensures that continuous-time signals are continuous during integration.

The compiler forbid writting some of the standard Simulink blocks when applied to continuous-time inputs.

Troublesome blocks in continuous time



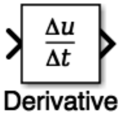
- What is the ‘previous’ value of a continuous time input?

Transport Delay

- The ideal definition is:

$$\begin{aligned} \mathit{delay}(\tau)(x)(t) &= x(t-\tau) && \text{if } t \geq \tau \\ &= 0 && \text{otherwise} \end{aligned}$$

- How to implement it?



- no symbolic differentiation is computed.
- what should be the output?

They explicitely rely on the major step of the simulation engine.

This makes models very fragile.

Their use in continuous time model is warned in the Simulink documentation (for good reasons).

In `Zélus` and `SCADE Hybrid`, they are wrongly typed.

Yet, it is possible to define a discrete time version of them, explicitely passing a signal telling when a discrete step is performed.

E.g., the memory block.

```
let hybrid memory(x0, z, x) =
  present z -> x0 fby x init x0

val memory : 'a * zero * 'a -C-> 'a
```

The memory block is often used to break algebraic loops. For that, `Zélus` and `SCADE Hybrid` provide the safer construct `last x`.

Conclusion

Most blocks can be programmed in a purely functional manner.

This gives a mathematically precise specification of the blocks that is compiled into sequential code.

Yet, some blocks cannot because they mix discrete and continuous time in an unprincipled manner.

An experiment with both `Zélus`, `SCADE`/`SCADE Hybrid`. [4]

For SCADE, blocks can be used and adapted according to the designer's needs.

The ability to mix discrete/continuous make possible to test/simulate a SCADE model of the software with its physical environment.

[4] Code available on the web page associated to the paper.

Open questions

Discrete/continuous

For `up(x)`, the compiler ensures that `x` is continuous during integration.

Yet, it is not able to impose that `x` must be C^1.

The type discipline impacts the way models are written.

It also inpacts the performance with possibly more zero-crossings.

Yet, zero crossings can be shared: `let x = up(e) in f(x) + g(x)`

Overloading

The type system is not powerful enough to express overloaded operators.

Do we need dynamic dispatch or can we stick to a stongly typed discipline (e.g., type classes)?

Zélus

A synchronous language with ODEs

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