Synchronous Objects with Scheduling Policies

Introducing safe shared memory in Lustre [a]

Marc Pouzet

Univ. Paris-Sud 11

Paris

March 2009, 10th

[a] Joint work with Paul Caspi, Jean-Louis Colaço, Léonard Gérard and Pascal Raymond

Motivations

- address the **modular programming** of synchronous systems with **modes**
- allowing to **separate** their **specification** from their **implementation** and their **instantiation** with a particular controller

Existing solutions are either unsafe or two restrictive to allow for a truly modular design

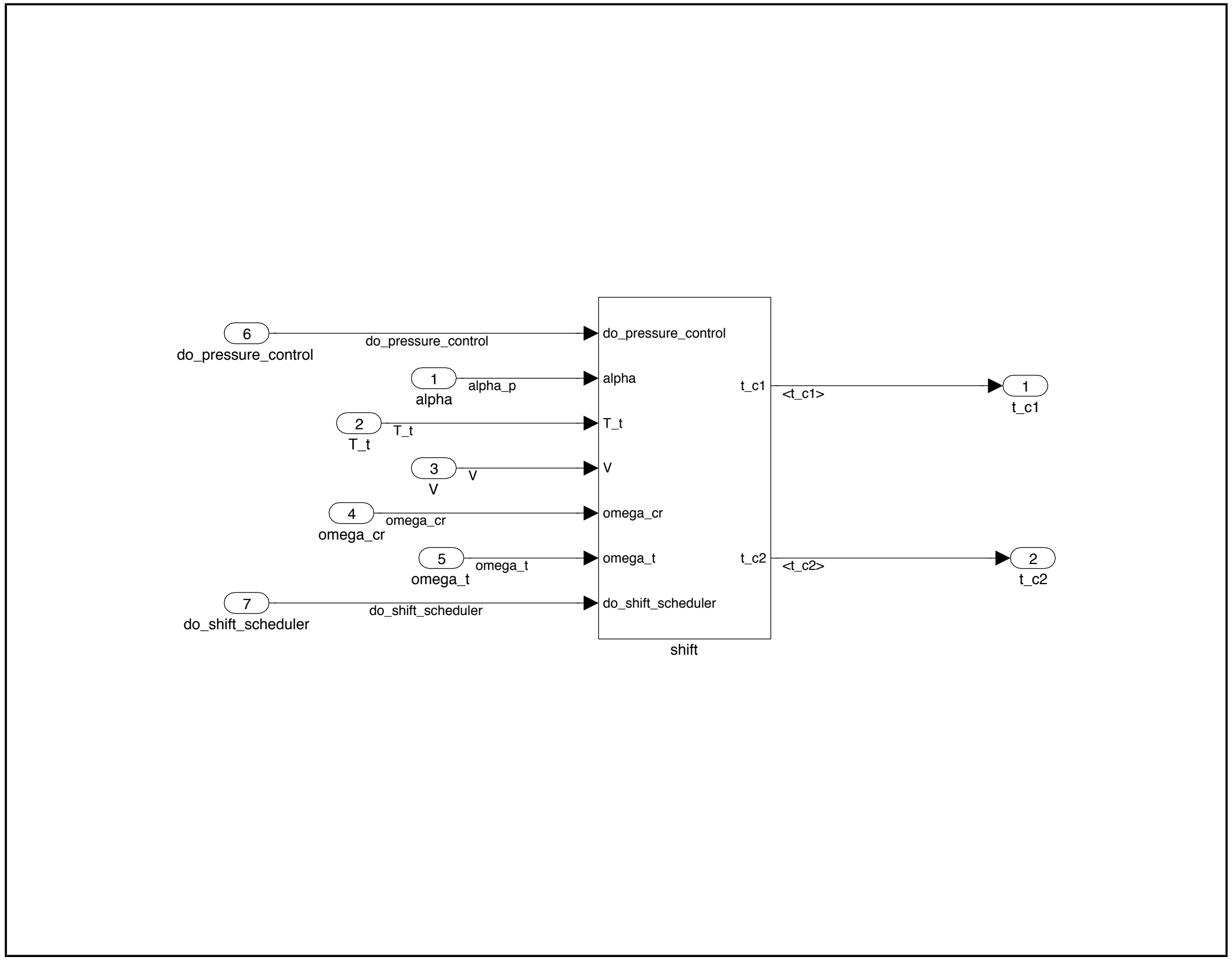
- in synchronous data-flow, shared states must be transmitted explicitly
- the Simulink solution is more modular but rely on unsafe read/write to shared variables

Proposal: We make an analogy between modes and object orientation and propose to organize a design in term of classes and objects:

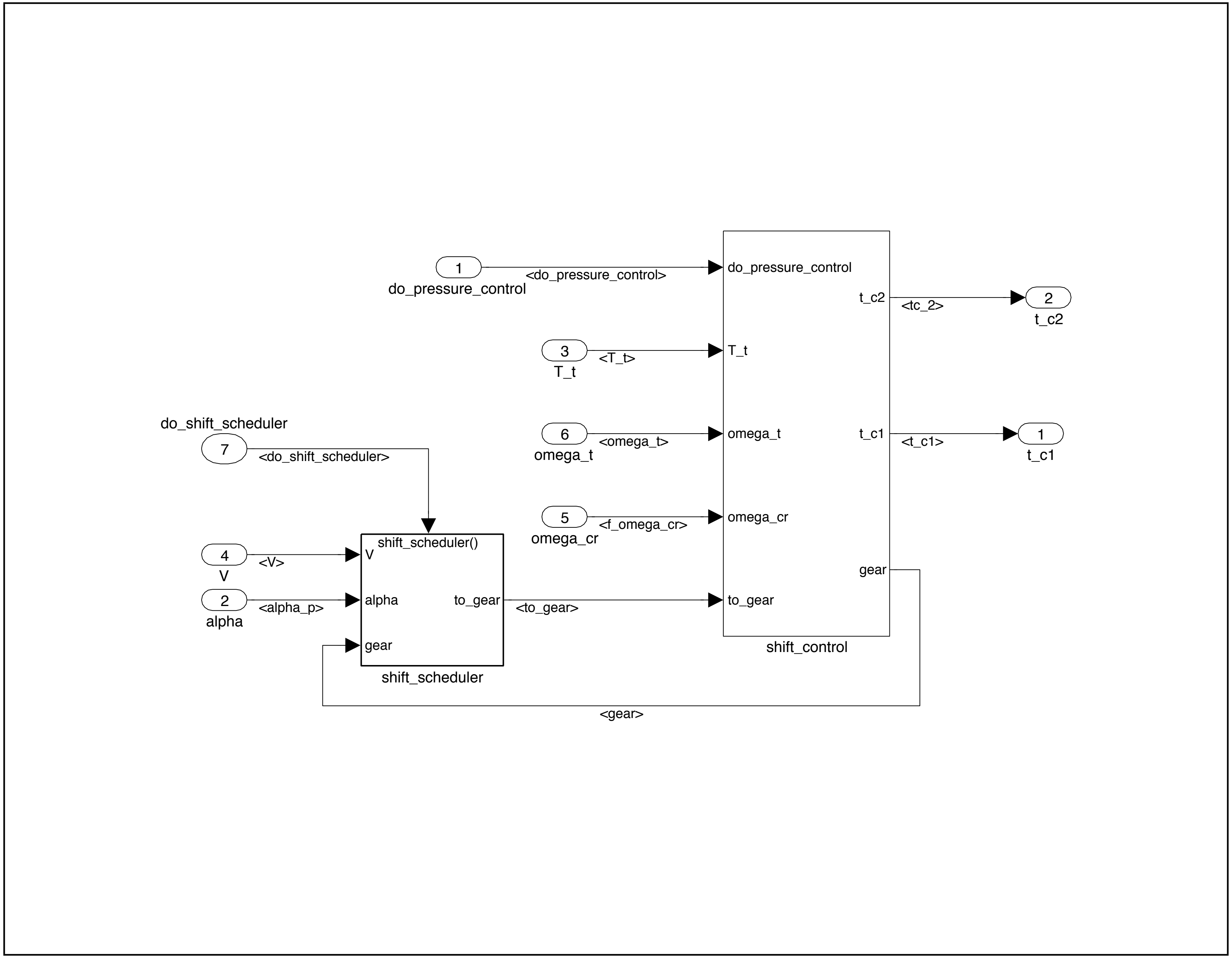
- shared variables play the part of attribute
- subsystems defining the behavior of each mode correspond to methods
- provide modular means to guaranty the absence of conflict (e.g., critical races)

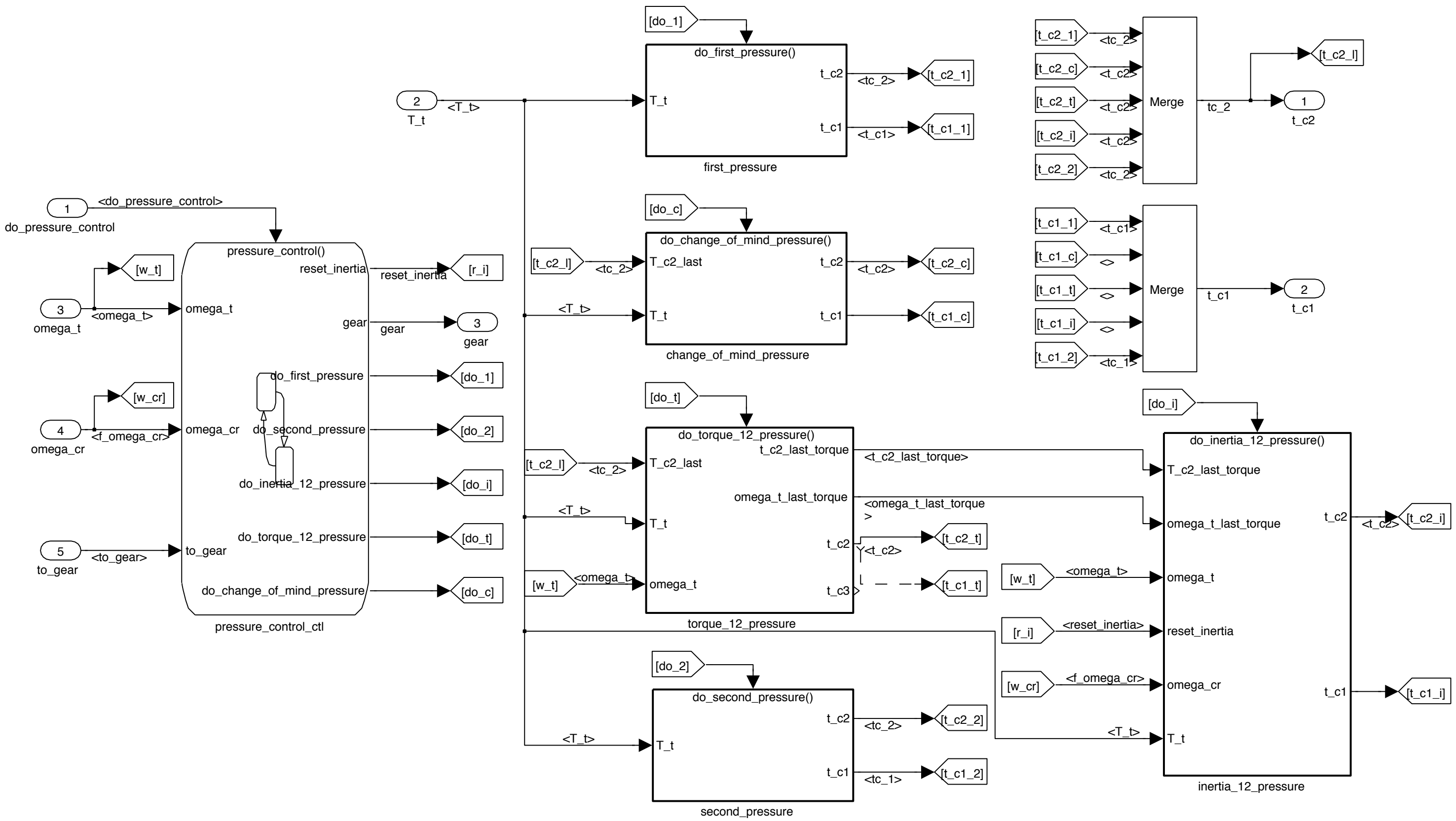
Example: an automotive power-train modeling [a]

- a set of modes described as data-flow block (in Simulink)
- activated through hierarchical automata (in Stateflow)



[a] http://amp.ece.cmu.edu/eceseminar/2000/Spring/slides/Butts


1
do_pressure_control
<do_pressure_control>
do_pressure_control
t_c2
<tc_2>
2
t_c2
3
T_t
<T_t>
T_t
do_shift_scheduler
7
<do_shift_scheduler>
6
omega_t
<omega_t>
omega_t
t_c1
<t_c1>
1
t_c1
5
omega_cr
<f_omega_cr>
omega_cr
shift_scheduler()
4
V
<V>
V
gear
2
alpha
<alpha_p>
alpha
to_gear
<to_gear>
to_gear
gear
shift_control
shift_scheduler
<gear>


do_pressure_control
pressure_control()
reset_inertia
gear
do_first_pressure
do_second_pressure
do_inertia_12_pressure
do_torque_12_pressure
do_change_of_mind_pressure
omega_t
omega_cr
to_gear
pressure_control_ctl
do_first_pressure()
first_pressure
do_change_of_mind_pressure()
change_of_mind_pressure
do_torque_12_pressure()
torque_12_pressure
do_second_pressure()
second_pressure
do_inertia_12_pressure()
inertia_12_pressure
Merge
Merge
t_c2
t_c1

Observation

Interest of the approach

- control laws are described as data-flow systems whereas their activation is defined as an hierarchical automaton
- both styles (data-flow + control-flow (automata)) live together and can be combined

Weaknesses

- the control structure is completely hidden in boolean variables
  - E.g., nothing states that `do_1` and `do_2` are exclusive
  - exclusive flows have to be merged; concurrent writes are not statically checked
- too much wires in this diagram!
  - the current and last value managed explicitly (e.g., `t_c2_c` and `t_c2_l`)
  - otherwise, use the “Read/Write” blocks but this may lead to critical races

Other tools/languages

This design methodology can be followed with many other tools combining two different languages/notations: SCADE 5+SSM, PtolemyII, etc.

Still, we need a **better/more integrated solution** inside a unique language

Mode automata (SCADE 6) [Maraninchi et all, ESOP'98; Colaço et all, EMSOFT'05 and '06]

Provide a programming construct to describe systems with modes

- modes communicate through shared variables (`last o`)
- ensure the absence of data-race by simple means

```
let node updown(y) returns (o)
  last o = 0 in
  automaton
  | Up -> do o = last o + y until (o = 4) then Down Done
  | Down -> do o = last o - y until (o = -4) then Up Done
  end

val updown : int => int
```

Mode Automata

- data-flow equations and hierarchical automata can be **mixed arbitrarily**
- the resulting language is **compiled into a subset language**, mainly Lustre with clocks
- this programming construct is **integrated to SCADE 6**

Still, **mode-automata do not allow for a truly modular** programming of modes allowing to separate:

- the specification of modes
- its instantiation with a precise control automaton

The actual solution is to program modes in a purely functional manner:

- **explicitly communicate values between states** (add extra wires)
- this reduces **modularity** and leads to **poor generated code**

In this sense, the Simulink solution using imperative shared variables is more modular

Mode Automata and Structured Design

Mode-automata are not only related to questions of mixing discrete and continuous dynamics but also to questions of modular design.

Suppose that two teams must develop a system with two modes, `up` and `down`...

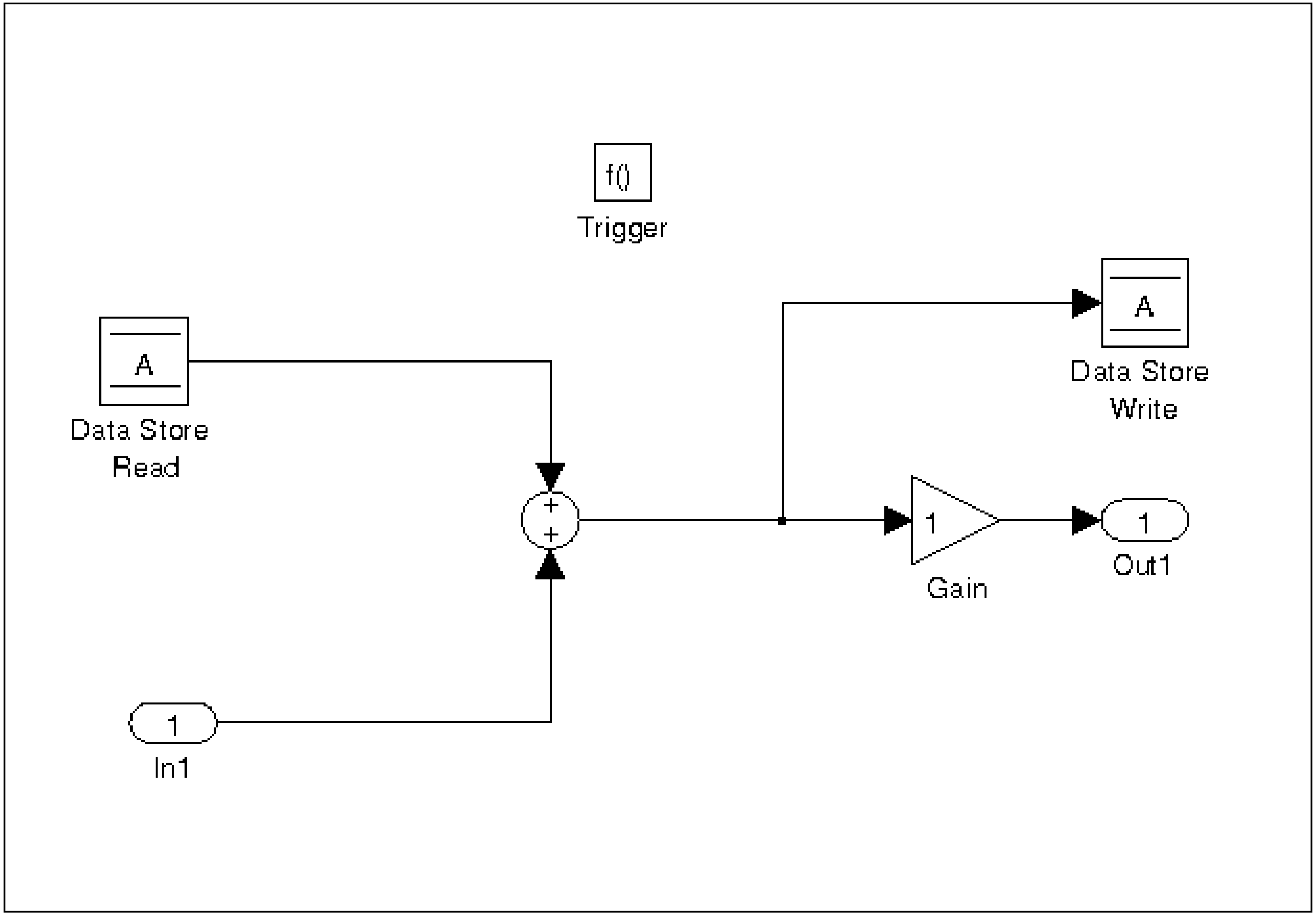
Software architecture design

- define the functional requirements of the two main modes
- define the interface between them, i.e., *the shared state variables the modes have to exchange*
- together with their name, types, ranges, precision, timing characteristics, etc.

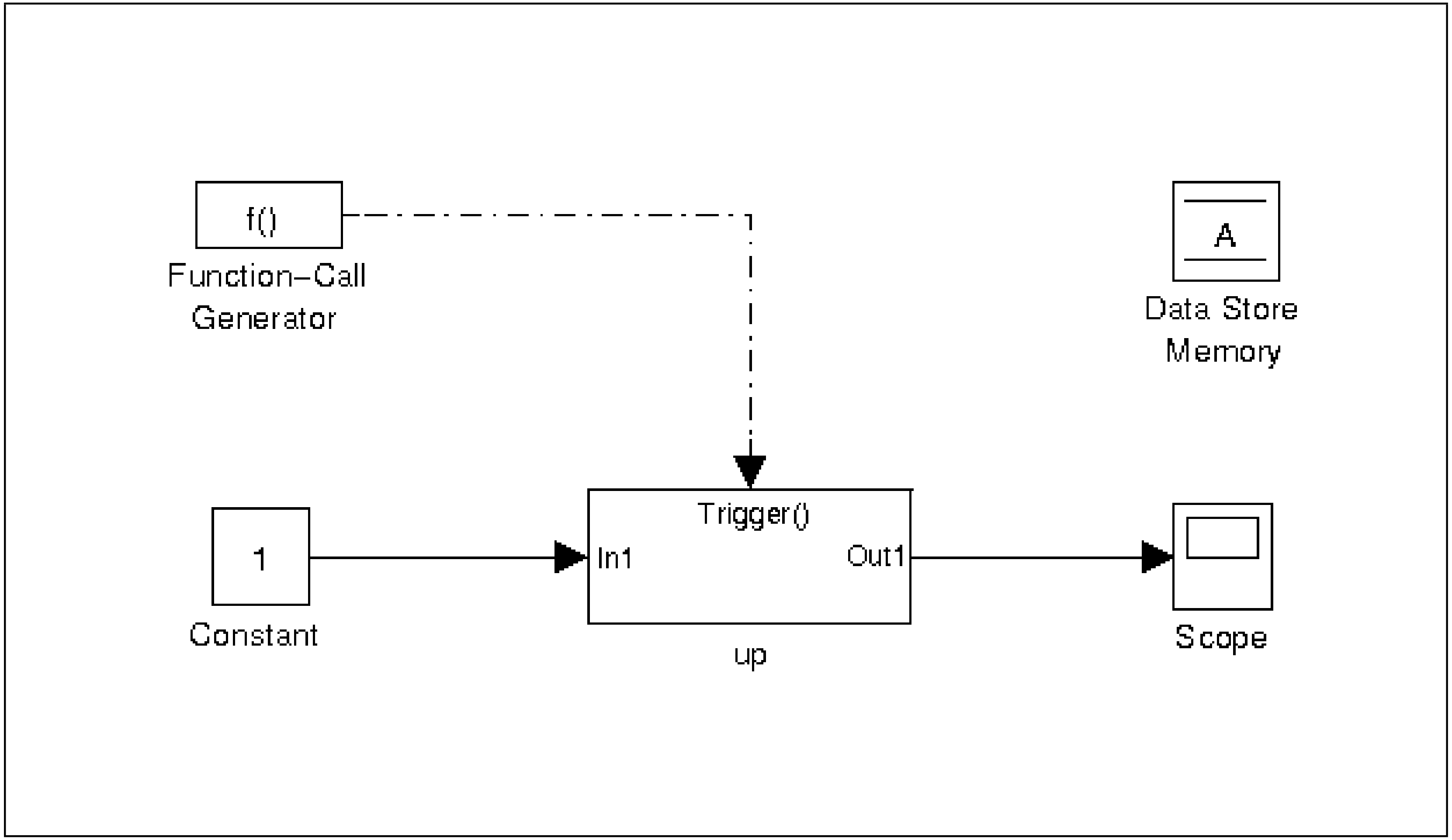
Mode design

- each team can build its own mode, modeling it, simulating it, testing it
- in Simulink, this will be done thanks to the “Data Read/Data Write” and “Data” blocks.

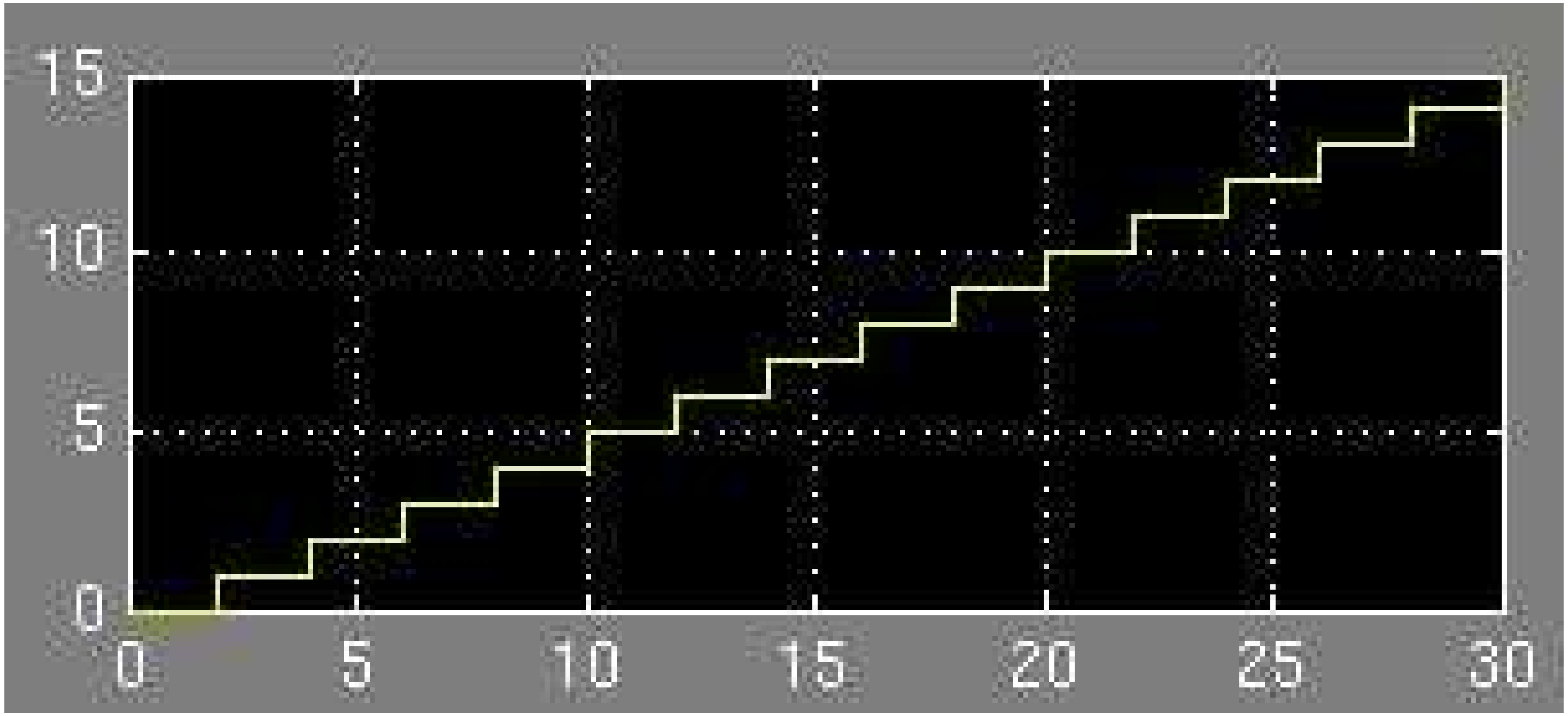
The up mode


f()
Trigger
A
Data Store
Read
A
Data Store
Write
+
+
1
Gain
1
Out1
1
In1

The up mode test harness

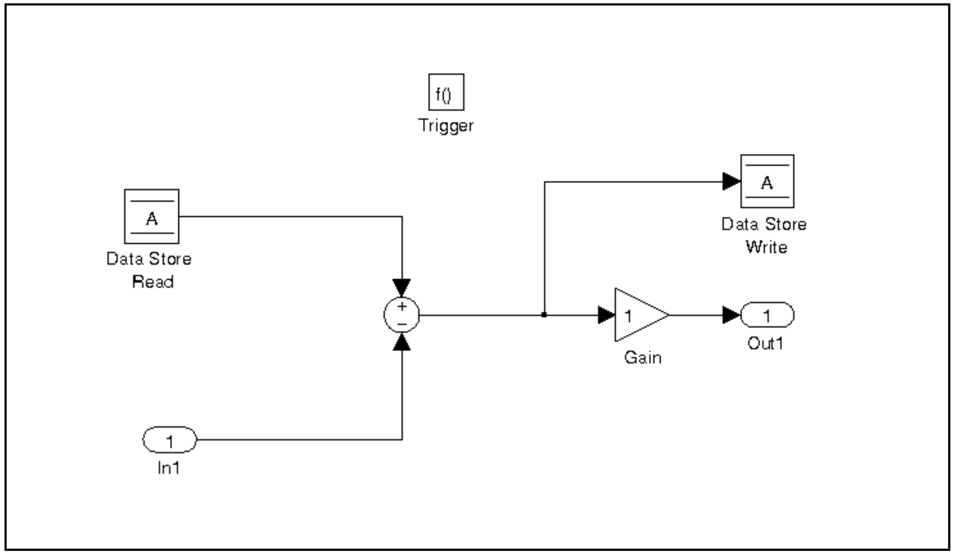


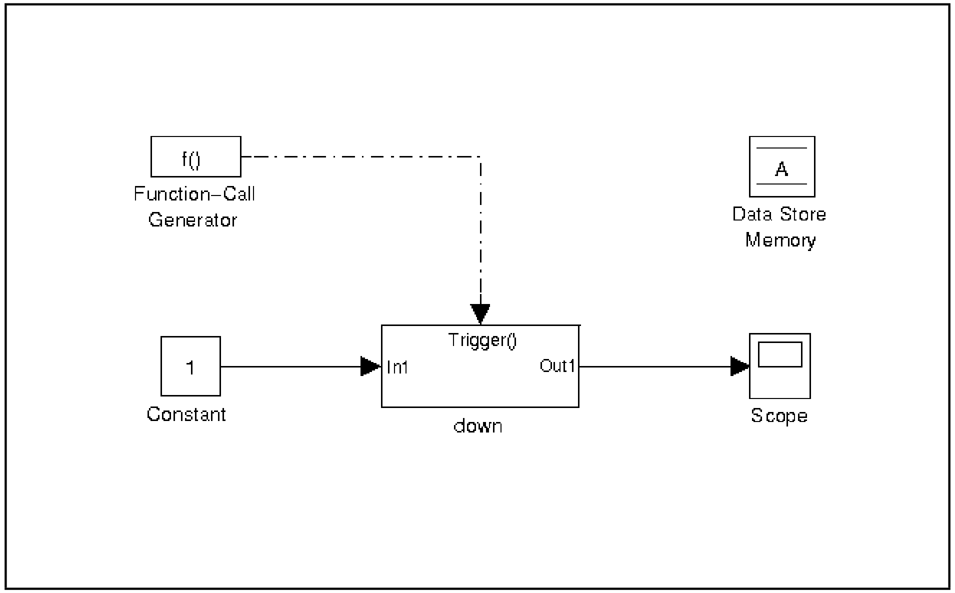
The up mode test result

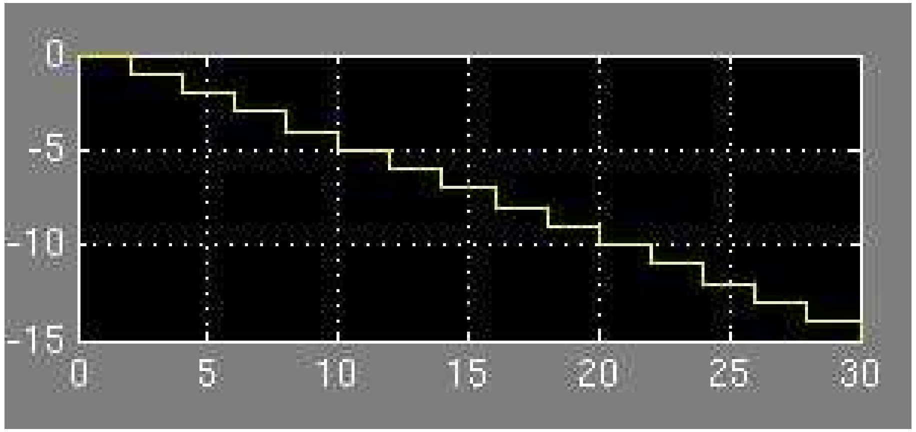


The dowm mode

In a similar way:





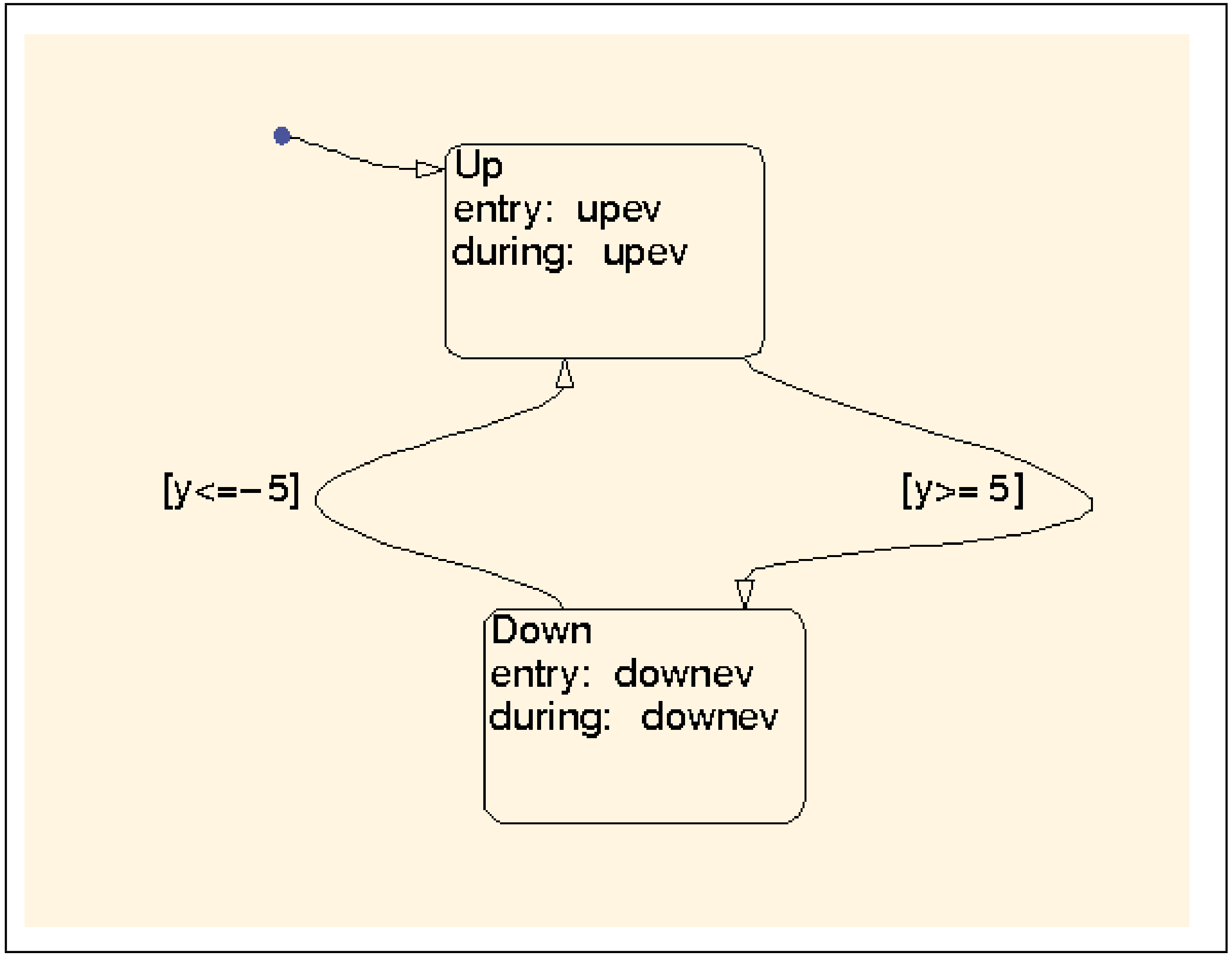


Overall Design

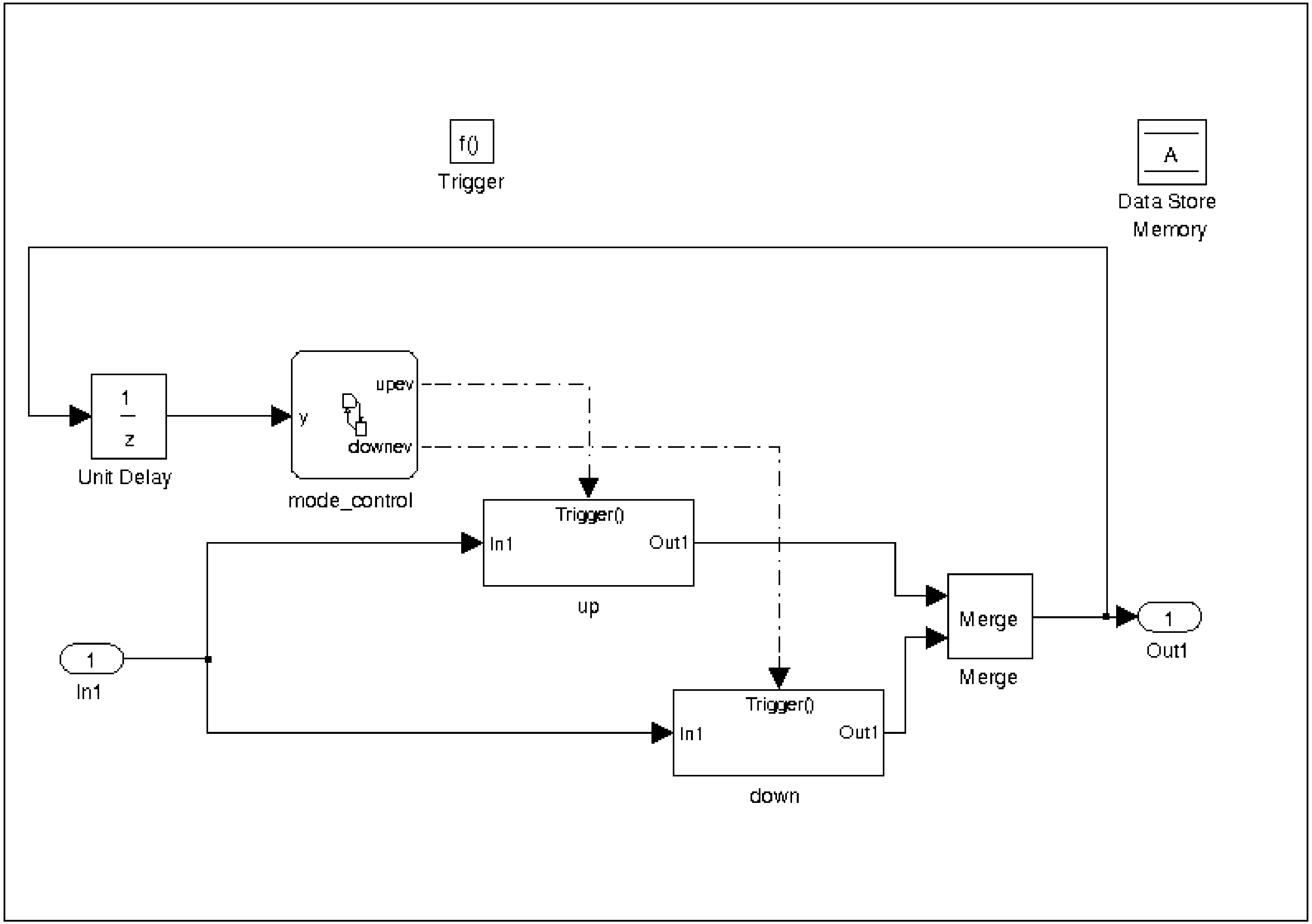
In parallel, the architecture team can start studying the global system model dealing with the transition logic.

Intensive use of Stateflow at this stage with global variable blocks of Simulink

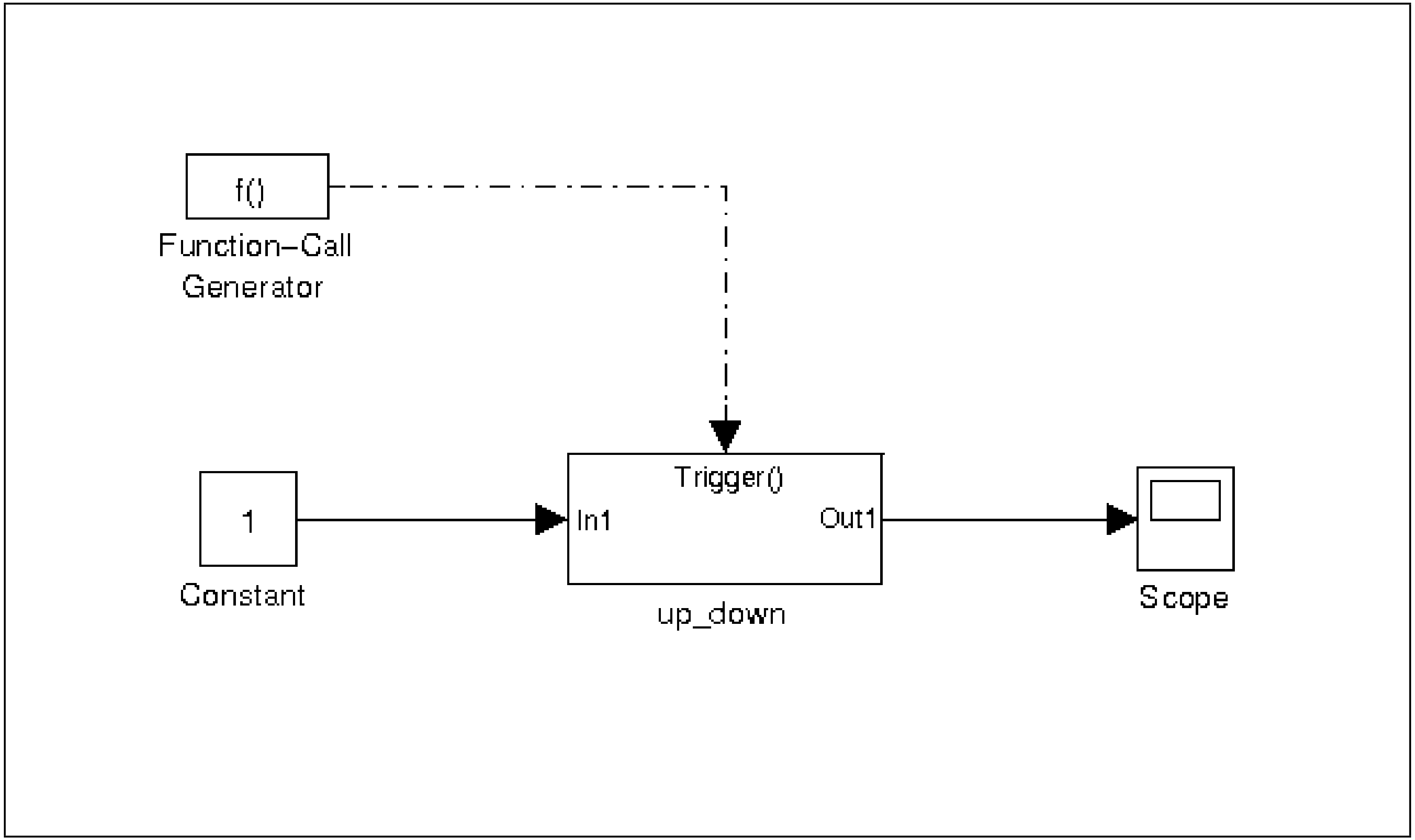
For example...



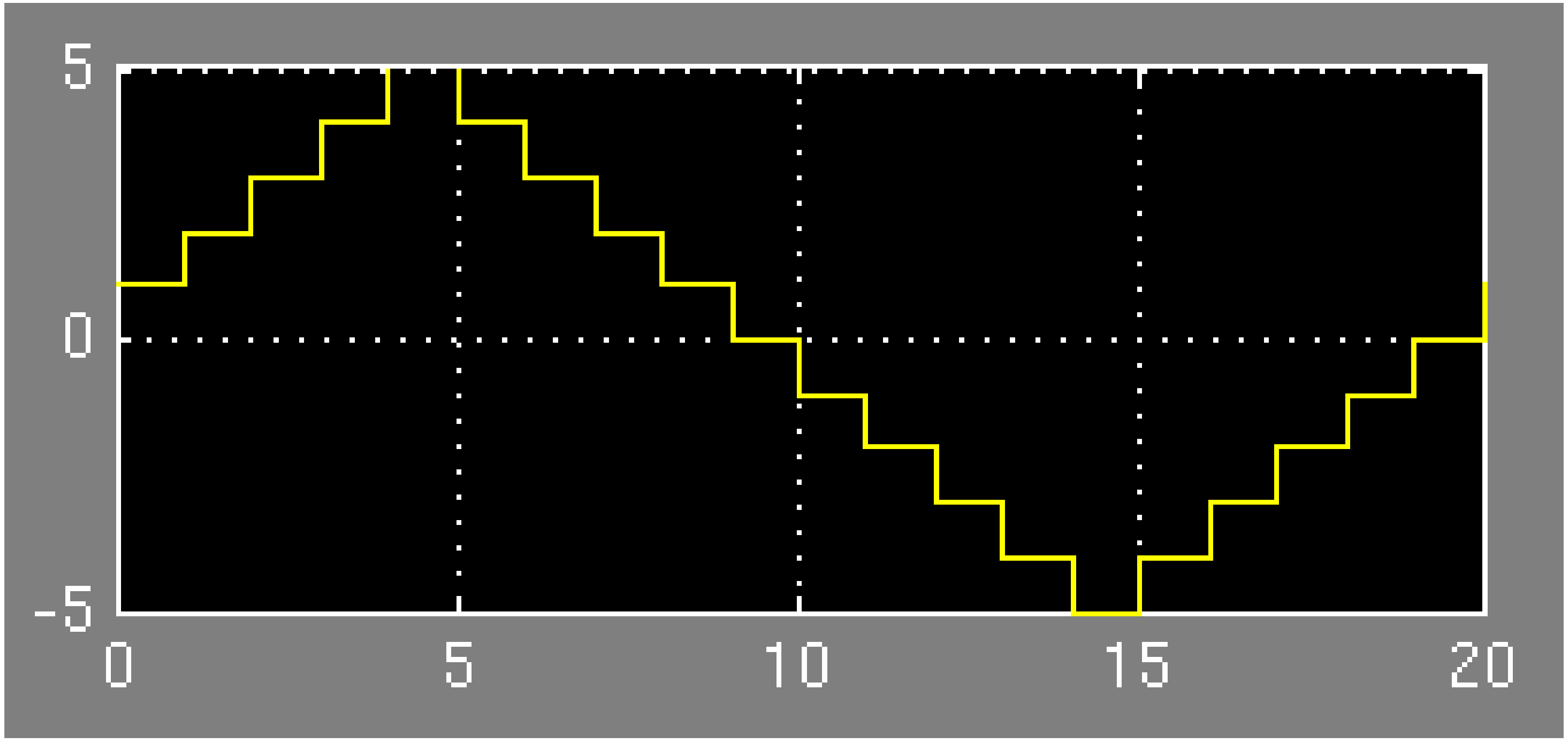
Integration Phase (the updown subsystem)



Integration Phase (the updown test harness)



Integration Phase (the updown test result)



Interest of the Approach

- it is modular: teams can work quite independently
- problems are dealt with at the right level: problems at each mode are treated at the mode level and global problems are treated at the integration level.
- clearer: global variables allow avoiding complex wiring (and corresponding wiring error)

Question

- this last point is the classical weakness of purely functional programming
- it could be simulated in a purely functional way (with monads-like constructs)
- this would not give good target code

Drawbacks

- those of Simulink/Stateflow first: imprecise semantics, termination problems, weak typing and absence of static checks
- ensure the modes are exclusive in time; otherwise, the semantics of the “Read” and “Write” Data block can become as chaotic as to depend upon the lexicographic order of the subsystems they are included in
- the “Data” block corresponds to the declaration of a shared variable with a “dynamic binding” semantics
  - in the `updown` system, the fact that “A” reads in “up” match “A” writes in “down” is discovered when the global model “updown” is constructed.
  - this is not necessarily a bad principle but...
  - most modern functional language stick to static binding for safety reasons (remember that block-diagram language are functional first-order languages)
  - Shared variables become global variables: Simulink is not that modular

Some Proposal for Improvement

We have thus identified two main drawbacks related to:

- the exclusivity of mode activation,
- the dynamic binding of shared variable names

Proposal

- a system is an object where shared variables stand for instance variables and modes are methods
- equip with a mean to specify the valid use of an object ensuring the absence of concurrent writes
- build upon control-structures (e.g., [EMSOFT'05, EMSOFT'06])
- source-to-source compilation into an object-based imperative code

Examples

```
let f x =
  object
    last o = x

    when up(y) returns (o) where
     o = last o + y

    when down(y) returns (o) where
     o = last o - y

  with up # down
  end
```

val f :
   int -> < up: int => int; down: int => int with up # down >

`f` is essentially a parameterized class. When evaluated, it returns an object with two methods

The synthesized type gives names of methods, their type and reminds the **scheduling policy** of the object. Every instance will have to follow it.

```
val f : int -> < up: int => int;
                 down int => int
                 with up # down >
```

- the notation `up # down` states that `up` and `down` are exclusive, i.e., they should not appear both in a synchronous reaction
- `up # down` is a scheduling policy which define what is a valid synchronous reaction
- it defines a finite set of valid scheduling

Instantiation

```
(* instantiation *)
let node g(x) returns (w) where
  new o = f (x + 2) in
  automaton
    | Up -> do v = o.up(y) until (v = 5) then Down done
    | Down -> do v = o.down(y) until (v = -5) then Up done
  end
```

val g : int => int

- `new o = f(x+2)` instantiates the object. `o` has a local scope
- the two modes are executed exclusively
- this verification is simple and syntax-directed

Observation Methods

Separate the code that modifies the state form the code that observe it.

```
let move x y =
  object
    last nx = x
    last ny = y

    when movex(x) returns () where
      nx = last nx + x

    when movey(y) returns () where
      ny = last ny + y

    when show() returns (nx, ny)

  with ((movex || movey) # {}) < show
  end
```

$P_1 \mid\mid P_2$ is the shuffle operator; $P_1 < P_2$ for the sequence; $\{\}$ is the empty schedule

(Re)-building Lustre primitives: `pre`, ->

```
let ipre(x) =
  object
    last nx = x
    when get() returns (last nx) where
    when set(y) returns () where
      do nx = y done
  with get < set
  end

val ipre : 'a -> < get: unit => 'a; set: 'a => unit with get < set >

let pre =
  object
    last nx
    when get() returns (last nx) where
    when set(y) returns () where
      do nx = y done
  with get < set end
val pre : 'a -> < get: unit => 'a; set: 'a => unit with get < set >
```

```
let (->) =
  object
    last init = true
    when get(x,y) returns (o) where
      var o in
      do o = if last init then x else y
      and init = false
      done
  with get end

let node fby(x,y) returns (x -> pre(y))
```

That is:

```
let (fby) =
  object
   new p = pre
   new o = (->)
   when get(x) returns (o.get(x,p.get()))
   when set(y) returns (p.set(y))
  with get < set
  end
```

What is minimal?

A Lustre node is a particular case of a **synchronous object**

```
let node counter(x,y) returns (z) where
  var y in
  do z = 0 -> pre z + y + cpt done
...

r = counter(x1,x2)
```

is equivalent to

```
let counter =
  object
   when step (x,y) returns (z) where
     var y in
     do z = 0 -> pre z + y done
   with step
   end
...
new m = counter in ... r = m.step(x1,x2)
```

Verification: correct use of an object

In the current implementation, the calling context should respect the scheduling policy specified by the programmer, i.e., it should be included in the set of declared schedules

```
let f x =
  object
    last o1 = x
    last o2 = x

    when one(y) returns (o) where
     o1 = last o1 + y

    when two(y) returns (o) where
     o2 = last o2 + y

  with up || down
  end
```

The policy `one||two` policy says that the e two methods `one` and `two` must be called in parallel.

Verification: correct use of an object

We can call `one` and `two` in a context where the two processes are run in parallel.

```
let node main1 x returns o where
  var o in
  new m = f (x+1) in
  do o = m.one(x) + m.two(x) done

let node main1 x returns o where
  var o in
  new m = f (x+1) in
  do o = m.two(m.one(x)) done
```

The calling context defines a scheduling which must be included in the set of possible schedules.

- `m.one || m.two` projected on `m` gives `one || two`
- `m.two < m.one` gives `two < one` included in `one || two`

Verification: soundness of a definition

The policy defines a set of schedules: we check that all of them are coherent

```
let f x =
  object
    last o = x

    when up(y) returns (o) where
     do o = last o + y done

    when down(y) returns (o) where
     do o = last o - y done

  with up < down end
```

is statically rejected

- Let $S = [(\texttt{last}\, o <\downarrow o)/up, (\texttt{last}\, o <\downarrow o)/down]$
- $S(up < down) = (\texttt{last}\, o < \downarrow o) < (\texttt{last}\, o < \downarrow o)$ is not satisfiable because $\downarrow o < \downarrow o$ is not

Higher-order

Write a component which take a component as an argument and infer the **most general** scheduling policy.

```
let node g h x returns (w)
  new o = h(x+2) in
  do automaton
   | Up -> do v = o.up(y) until (v = 5) then Down done
   | Down -> do v = o.down(y) until (v = -5) then Up done
   end
  and
   w = o.show(x)
  done

val g : (int -> < up: int => int; down: int => int;
                  show: int => 'a, ... with (up#down)||show) => 'a
```

Since we do not know the dependences between methods of `o`, we infer the strongest constraint.

Higher-order and implicit dependences

We can also syntactically force an execution order

```
let node g h x returns (w)
  new o = h(x+2) in
  do automaton
   | Up -> do v = o.up(y) until (v = 5) then Down done
   | Down -> do v = o.down(y) until (v = -5) then Up done
   end
  in
   w = o.show(x)
  done

val g : (int -> < up: int => int; down: int => int;
                  show: int => 'a, ... with (up#down)<show) => 'a
```

Here, the context says that `show` is necessarily done after.

Now we instantiate the previous code with an actual object.

```
let myf m = object
              when up(x) returns (v) ...
              when down(x) returns (v) ...
              when show() returns (...) ...
            with (up#down)<show
            end
...

(* instantiation *)
let node main x returns (w) where
  do o = f myf x done
```

This program is statically rejected in case (1) because `(up#down)||down` is not included in `(up#down)<show`. It is accepted in case (2).

Scheduling policies and Scheduling Constraints

Policies:

- a scheduling policy is an expression telling what is a valid reaction
- the activation of a mode corresponds to the definition of a clock name; the policy is a boolean property

$$P \quad ::= \quad m \mid \epsilon \mid P \mid\mid P \mid P < P \mid P \# P$$

Constraints:

- Add shared variables and named methods to policies

$$C \quad ::= \quad o.m \mid \epsilon \mid C \mid\mid C \mid C < C \mid C \# C \mid \downarrow x \mid \uparrow x \mid \texttt{last}\, x$$

Soundness of Policies and Constraints

A stands for an action (e.g., method call, read, write).

Parallel composition as a shuffle operator. This lead to two interesting normal forms:

Constraints as sets of schedules:

$$\begin{array}{lll} t & ::= & A \mid A < t \\ C & ::= & \underset{i}{\#}\, t_i \end{array}$$

- Equality, inclusion, intersection simple to compute
- mostly unreadable and algorithmically expensive

Constraints as disjunctions of parallel/sequential schedules:

$$\begin{array}{lll} t & ::= & t < t \mid t \mid\mid t \mid A \\ C & ::= & \underset{i}{\#}\, t_i \end{array}$$

- less explosive; this is used for checking that a constraint is causal

Causality

```
let node f(x) returns (o) where
  var o1 in
  var o2 in
  do o1 = o2 + 1
  and o2 = o1 + 2 done

file "t1.ls", line 7-10, characters 2-63:

Type error: the following constraint is not causal.
^o2 < o1 || ^o1 < o2
```

A constraint is causal when every schedule is causal. This can be computed efficiently on the weak normal form. We check the absence of cycles.

```
let node f(x) returns (o2) where
  var o1 in
  last o2 = 0 in
  do o1 = last o2 + 1
  and o2 = o1 + 2 done
```

Soundness and Correction

Soundness: C is sound iff for every variable x in C, its normal form does not contain $(\downarrow x < \downarrow x)$ nor $(\downarrow x < \uparrow x)$ nor $(\downarrow x < \uparrow \texttt{last}\, x)$.

Relating policies : inclusion between normal forms

Restriction: If C is a constraint, $C|_o$ is the projection of C on o. It returns a policy where only method calls $o.m$ have been kept.

Correct Use of an Object: If P_c is the declared policy of o and $C|_o$ is the actual policy of o. It has to respect P_c, that is, $C|_o \subseteq P_c$.

The Type Language

$$
\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1, ..., \alpha_n . \forall \rho_1, ..., \rho_m . t \\
t & ::= & t \rightarrow t \mid t \times t \mid \alpha \mid c(t, ..., t) \mid r \\
r & ::= & \emptyset \mid m : t, r \mid r \;\texttt{with}\; P \mid \rho
\end{array}
$$

A typing environment H is defined in the following way:

$$
H \quad ::= \quad \emptyset \mid H + x : \sigma \mid H + \texttt{last}\, x : t \mid H + \texttt{new}\, o : t
$$

The Type Judgment:

$$
H, C \vdash e : t
$$

Under typing environment H and scheduling constraints C, e is of type t.

The Type System

Syntax-directed construction of constraints, e.g.,:

$$\frac{H, C_1 \vdash e_1 : t_1 \quad H, C_2 \vdash e_2 : t_2}{H, C_1 \mid\mid C_2 \vdash (e_1, e_2) : t_1 \times t_2} \qquad \frac{H, C_1 \vdash e_2 : t_1 \to t_2 \quad H, C_2 \vdash e_1 : t_1}{H, C_1 \mid\mid C_2 \vdash e_2\ e_1 : t_2}$$

$$\frac{H, \epsilon \vdash e : \{r \texttt{ with } C|_o\} \quad H + \texttt{\textbf{new}}\, o : \{r \texttt{ with } C|_o\}, C \vdash d : H_0}{H, C_o \vdash \texttt{new}\, o = e \texttt{ in } d : H_0}$$

$$\frac{\begin{array}{c} H \vdash \mathit{fields} : H_0 \quad H, s(P) \vdash \mathit{objs} : H_1 \\ H + H_0 + H_1 + \texttt{self} : \{r \texttt{ with } P\} \vdash \mathit{modes} : r, s \quad \mathit{Sound}(s(P)) \end{array}}{H, \epsilon \vdash \langle \mathit{fields}\ \mathit{objs}\ \mathit{modes} \texttt{ with } P \rangle : \{r \texttt{ with } P\}}$$

The Type System:

- it is based on row types as introduced by Rémy & Vouillon for Objective ML [TPOS'97]
- we extend row types with policies

The comma operator (,) is the concatenation for method names and act as the exclusion operator ($\#$) on scheduling policies. Rules are:

$$
\begin{array}{lcl}
(r \;\texttt{with}\; P_1) \;\texttt{with}\; P_2 & = & (r \;\texttt{with}\; P_2) \;\texttt{with}\; P_1 \\
(r \;\texttt{with}\; P_1) \;\texttt{with}\; P_2 & = & r \;\texttt{with}\; P_1 \# P_2 \\
m_1 : t_1, m_2 : t_2, r & = & m_2 : t_2, m_1 : t_1, r
\end{array}
$$

Using above properties, a row type can be normalized into:
$\{m_1 : t_1, ..., m_n : t_n \;\texttt{with}\; P; \rho\}$ or $\{m_1 : t_1, ..., m_n : t_n \;\texttt{with}\; P\}$.

The unification algorithm of Rémy & Vouillon is modified accordingly.

Example

```
let node g(f)(x) returns (r) where
  new o1 = f (x) in
  new o2 = f (x+1) in
  var r in
  if (x = 0) then do r = o1.m(x) + o2.m(x) done
  else do r = o1.n(x) + o1.m(x) done
```

```
val g :
   (int -> < m: int => int; n: int => int ...
             with {} # m # (m || n) >) -> int => int
```

Constraints:

$$H', (o_1.m \mid\mid o_2.m) \vdash o_1.m(x) + o_2.m(x) : \texttt{int}$$

and

$$H', (o_1.n \mid\mid o_1.m) \vdash o_1.n(x) + o_1.m(x) : \texttt{int}$$

then

$$C = (o_1.m \mid\mid o_2.m) \;\#\; (o_1.n \mid\mid o_1.m)$$

Discussion

Still, we cannot develop and test modes separately without adding extra wires.

An alternative to objects was to simply add (restricted forms) of references + an effect type system. E.g.,:

```
let node up(last x)(y) returns () where
  do x = last x + y done
```

- `up` takes a reference (a left-value) and modifies it
- a type-system with effect (e.g., policy constraints) to check the coherency. E.g.:

$$\forall \mathtt{last}\, x : \mathtt{int}.\mathtt{int} \Rightarrow \mathtt{unit}\ \mathtt{with}\ \mathtt{last}\, x < \downarrow x$$

- heavy, types hard to read; type inference is complicated in case of higher-order and not conservative

Extensions: inheritance

Still, we cannot develop and test modes separately.

```
class virtual cup =
  object
    virtual last x : int
    when up(y) returns (x) where
      do x = last x + y done
  end
```

```
class type cdown =
  object
    virtual last x : int
    when down(y) returns (x) where
      do x = last x - y done
  end
```

- join the two with multiple inheritance
- what should be the minimal policies given to each class?

```
class updown x0 =
  object
    last x = x0
    inherit up
    inherit down end
```

- The minimal constraint for `cup` is that `up(y)` reads and writes `x`

Related Works

- Reactive modules [Alur & Henzinger, FMSD'99]: more expressive (synchronous/asynchronous)
- Interface automata [De Alfaro & Henzinger, ESEC/FSE'01]: more expressive signatures (dynanic properties as automata) but need model-checking techniques
- 42 model [Maraninchi et all, GPCE'07]: general model of a component (synchronous/asynchronous), more general scheduling policies
- a synchronous object generalises the notion of a Lustre node
- the idea of separating the specification from the possible implementations is present in Signal; this is hidden in clocks; needs the full power of the clock calculus

Conclusion and Future Works

- a completely new implementation with these ideas (10000 LOC, Ocaml)
- the target language is imperative and object-based
- automata, signals and flows, (static) higher-order,
- compilation through source-to-source program transformation.
- provide more constructs (as macros) for describing policies
- avoid the declaration of policies
- introduce class and inheritance

References

[1] Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. Synchronous Objects with Scheduling Policies: Introducing safe shared memory in lustre. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2009.

[2] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.