Synchronous Objects with Scheduling Policies

Introducing safe shared memory in Lustre [*]

Paul Caspi
VERIMAG
Grenoble
Paul.Caspi@imag.fr

Jean-Louis Colaço
Prover Technology
Toulouse
Jean-louis.Colaco@prover.com

Léonard Gérard
LRI, Univ. Paris-Sud 11, INRIA
Orsay
Leonard.Gerard@lri.fr

Marc Pouzet
LRI, Univ. Paris-Sud 11, INRIA
Orsay
Marc.Pouzet@lri.fr

Pascal Raymond
VERIMAG
Grenoble
Pascal.Raymond@imag.fr

Abstract

This paper addresses the problem of designing and implementing complex control systems for real-time embedded software. Typical applications involve different control laws corresponding to different phases or *modes*, e.g., take-off, full flight and landing in a fly-by-wire control system. On one hand, existing methods such as the combination of Simulink/Stateflow provide powerful but unsafe mechanisms by means of imperative updates of shared variables. On the other hand, synchronous languages and tools such as Esterel or SCADE/Lustre are too restrictive and forbid to fully separate the specification of modes from their actual instantiation with a particular control automaton.

In this paper, we introduce a conservative extension of a synchronous data-flow language close to Lustre, in order to be able to define systems with modes in a more modular way, while insuring the absence of data-races. We show that such a system can be viewed as an *object* where modes are *methods* acting on a shared memory. The object is associated to a *scheduling policy* which specifies the ways methods can be called to build a valid synchronous reaction. We show that the verification of the proper use of an object reduces to a type inference problem using *row types* introduced by Wand, Rémy and Vouillon. We define the semantics of the extended synchronous language and the type system. The proposed extension has been implemented and we illustrate its use through several examples.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.2 [*Language Classifications*]: Data-flow languages; D.3.4 [*Processors*]: Code generation, Compilers

[*] This work was partially supported by the INRIA research project Synchronics.

General Terms Algorithms, Languages, Theory, Verification

Keywords Real-time systems; Synchronous languages; Block-diagrams; Compilation; Semantics; Type systems

1. Introduction

This paper addresses the problem of designing complex control systems. Typical applications involve several control laws or *modes*, composed together to form the final application. In these applications, each mode comes from a continuous or discrete control law and is naturally described by means of data-flow equations [1]. Conversely, the control activation of each law shall be described by a hierarchical automata *a la StateCharts* [10]. This clear separation explains the growing adoption of model-based design tools such as the combination of Simulink and Stateflow, Simulink being used to describe the data-flow part whereas Stateflow is used for the control-flow part. It is also a good software engineering discipline, especially in allowing, before the integration phase, to independently develop and test each mode as well as the activation automaton. In order to ease the communication between modes and avoid cumbersome wiring, Simulink/Stateflow only provides low-level mechanisms by means of imperative variables — the so-called *Read* and *Write* blocks — shared among the modes. This may lead to data-races which are not statically detected. Moreover, these concurrent accesses can appear in an order as chaotic as depending upon the lexical order of the subsystems they are included in (see [5]). Then an unadvised modification in the design can result in a drastic behavior deviation. To top this, shared variables are subject to *dynamic binding* which has its specific drawbacks. Conversely, synchronous data-flow languages such as Esterel or Lustre [2] forbid such concurrent accesses but they do not allow for a truly modular description of modes which separate each mode's specification from the actual instantiation with a particular control automaton: they essentially amount to programming modes in a purely functional manner forcing to pass explicitly the current state between each mode. This extra wiring is error-prone, it reduces the modularity/readability as well as the efficiency of the generated code.

Mode-automata [12, 7] have been proposed to arbitrarily mix data-flow equations and automata while ensuring the absence of

[1] The ford gear-shift study (`amp.ece.cmu.edu/eceseminar/2000/Spring/slides/Butts/S00_Butts_slides.pdf`).

concurrent writes. Every state of an automaton is defined as a collection of equations as in the following code [2]:

```
let node updown(y) returns (o) where
  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

`o` is a shared variable and `last o` defines the "last" value of `o`. If `y` is the constant stream `1`, this program computes the repeating sequence `(0.1.2.3.4.3.2.1.0.-1.-2.-3.-4.-3.-2.-1)`. This separation of modes in different states and the restriction that at most one state is active during a reaction ensures the safe access to shared variables, simplifies the semantics and code generation.

Nonetheless, Mode-automata are still limited in the sense that it is not possible to define a system with modes independently from their use. From a software engineering point-of-view, an architecture team should meets and defines the functional requirements of the two main modes — say `up` and `down` — independently on their use. It also defines the interface between the modes, that is to say *the shared state variables the modes have to compute and exchange* together with their name, types, ranges, required precisions and timing characteristics and any other convenient high level shared features. This is not possible in a purely data-flow context without adding extra wiring to every mode (here, `last o` as an extra input and `o` as an extra output) together with equations of the form `o = up(last o, y)` and `o = down(last o, y)` in the final code. In this sense, the joint use of Simulink/Stateflow, while it may be unsafe, is more modular.

Separating the definition of modes from the automaton itself also allows for sharing a behavior between different states of the automaton: two different states may activate the very same mode because they are never activated at the same time. While this is transparent with mode-automata when modes are combinatorial, modes sharing variables should be put outside of the automaton and some update code would be duplicated. With the proposition given in the paper, such sharing is made easier and fits well with Mode-automata.

In this paper, we introduce a mechanism which allows to specify a system with modes in a modular way while insuring the absence of data-races. The designer specifies the shared variables used for communication and the various modes, all of them being gathered into an *object*. Together with the object, the designer defines its *scheduling policy* which specifies the directions for use of the various modes in order to produce a valid synchronous reaction of the system. Whereas Lustre essentially provides one scheduling policy through a unique *step* function, this new programming construct gives more freedom, allowing to describe each mode independently from the other. Let us continue our introductory example.

```
let twomodes x0 =
  object
    last o = x0

    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
```

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

`twomodes` is a function returning an object made of two modes. Together with the modes, it specify a *scheduling policy* stating that the two modes are exclusive, i.e., they should never be executed during the same reaction and this is the meaning of (`up # down`). Now, the object can be instantiated, e.g.:

```
let node main(y) returns (o) where
  new m = twomodes(0) in
  automaton
  | Up -> do o = m.up(y) until o = 4 then Down done
  | Down -> do o = m.down(y) until o = -4 then Up done
  end
```

val main : unit => int

The two pieces of code can be analyzed and compiled separately. Defining only exclusive modes would be overly restrictive as soon as modes share several state variables as it is shown in the following example :

```
let point x0 y0 =
  object
    last x = x0
    last y = y0

    when translate (nx, ny) returns (nb_translations)
     where
     do  nb_translations = 1 -> pre nb_translations + 1
     and x = last x +. nx
     and y = last y +. ny
     done

    when rotate (xc, yc, tetha) returns (nb_rotations)
     where var d in
     do  nb_rotations = 1 -> pre nb_rotations + 1
     and d =  sqrt ((last x -. xc) ** 2.0
            +. (last y -. yc) ** 2.0)
     and x = last x +. d *. tetha
     and y = last y +. d *. tetha
     done

    when cartesian () returns (x, y)

    when polar () returns (dd, tetha) where
     do  dd = x +. y
     and tetha = if x > 0.0 then (y /. x)
                 else if x = 0.0 then  2.0
         else (-. y /. x)
    done

    with (translate # rotate)
        < ((cartesian # {}) || (polar # {}))
  end
```

```
val point : float -> float ->
  < polar: unit => float * float;
    cartesian: unit => float * float;
    rotate: float * float * float => int;
    translate: float * float => int
    with (translate # rotate) < ((cartesian # {}
                                  || (polar # {})) >
```

Its scheduling policy states that any valid synchronous reaction should first call either `translate` or `rotate` then, it may call `cartesian`, `polar` or both. Notice that it would be also possible to give the policy `cartesian || (polar # {})` meaning that, `polar` could not be called without a call to `cartesian`. Scheduling policies

[2] All examples presented in this paper have been tested with a prototype compiler. Types computed by the compiler are printed like *this*. The compiler is available on demand.

are expressed in the following language:

$$P \quad ::= \quad P \,||\, P \mid P \# P \mid P < P \mid m \mid \epsilon \quad ^{3}$$

$P_1 \; || \; P_2$ stands for the parallel composition of two scheduling policies (or *shuffle* in term of scheduling); $P_1 \# P_2$ states we either have schedules from P_1 or schedules from P_2; $P_1 < P_2$ states that schedules from P_1 precede schedules from P_2; finally, m states that the method m should be called whereas ϵ stands for the empty schedule or the schedule with no calls.

In this paper we show that checking the correct use of objects amounts at checking policy inclusion between declared policies and computed ones. Moreover, when considering higher-order cases, the compilation infers scheduling constraints on unknown objects. To this effect we use a type system based on *row variables* [15] extended with scheduling policies.

```
let node g h x y returns (v) where
  new o = h(x) 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 : ('a -> < up: 'b => int; down: 'b => int ...
                 with {} # up # down >) ->
         'a -> 'b => int
```

The function g can be applied to any function h returning an object with at least two methods up and down with policy up#down. The notation ... states that h is a function returning an object which containts, at least, methods up and down with the given policy. As a consequence, g can be applied to a function providing more methods but whose policy should contains the inferred policy.

The paper is organized as follows. Section 2 presents the core language and the algebra of policies. Section 3 presents the synchronous semantics and the type system of the language kernel. In Section 4 we discuss implementation issues, possible extensions and related works. We conclude in Section 5.

2. A Calculus of Synchronous Objects

We define a core data-flow language extended with the ability to define objects. A program defines a collection (d) of top-level values in sequential order. (p) range for patterns. The block (b) may define local variables ($\texttt{var}\, x \texttt{ in } b$) and instantiated objects ($\texttt{new}\, x = e \texttt{ in } b$) or a collection of equations (D). A collection can be an equation ($p = e$), a parallel ($D_1 \texttt{ and } D_2$) or sequential composition ($D_1 \texttt{ in } D_2$). Finally, it can be based on the disjunction of two equations ($\texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2$) according to a boolean condition (e). An expression (e) may be a constant (v), a variable (x), a read access to a shared variable ($\texttt{last}\, x$), a pair (e_1, e_2), a function application ($e_1 \; e_2$), a function ($\lambda x.e$), a method call ($o.m(e)$) of an object o with name m and, finally, the definition of a system with modes ($\texttt{object}\, \textit{fields objs modes}\ \texttt{with}\ P\, \texttt{end}$). We call it an un-instantiated object (or simply object). An object has instance variables (*fields*), instantiated objects (*objs*), a sequence of modes (*modes*) and a scheduling policy (P) specifying the way modes can be called.

$$\begin{array}{lcl}
d & ::= & \texttt{let}\, x = e \mid d; d \\
p & ::= & p, p \mid x \\
b & ::= & \texttt{var}\, x\ \texttt{in}\ b \mid \texttt{new}\, x = e\ \texttt{in}\ b \mid D \\
e & ::= & v \mid x \mid \texttt{last}\, x \mid (e, e) \mid e\ e \mid \lambda x.e \mid o.m(e) \\
 & & \mid \texttt{object}\, \textit{fields objs modes}\ \texttt{with}\ P\, \texttt{end} \\
D & ::= & p = e \mid D\ \texttt{and}\ D' \mid D\ \texttt{in}\ D' \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ D\ \texttt{else}\ D' \\
\textit{fields} & ::= & \texttt{last}\, x = e; ...; \texttt{last}\, x = e \\
\textit{objs} & ::= & \texttt{new}\, o = e; ...; \texttt{new}\, o = e \\
\textit{modes} & ::= & \texttt{when}\, m(p)\, \texttt{returns}\, (p)\ b; \\
 & & ...; \texttt{when}\, m(p)\, \texttt{returns}\, (p)\ b \\
v & ::= & C \mid i
\end{array}$$

[3] ϵ printed {} by the compiler.

We only provide a basic control structure. Other ones, such as automata, raise no particular difficulty from the typing point-of-view and are discarded in the language kernel.

2.1 Derived Operators

This core language is expressive enough to reprogram the classical node construction, pre and initialization operator -> of Lustre.

Lustre nodes A Lustre node is a *statefull* stream function f with argument p, result q and body b. A statefull function may depend on the history of its inputs. In concrete syntax, we have written $\texttt{node f}(p)\, \texttt{returns}\, (q)\ b$. Such a node is a shortcut for the following definition of an object:

```
let node_f =
  object
    when step(p) returns (q) b
  with step # {}
  end
```

Each node application $\texttt{f}(e)$ is replaced by a call to the method step, that is, $\texttt{f.step}(e)$ provided node_f has been instantiated by writing new f = node_f in the context of the method call.

The Lustre delay pre The initialized delay $\texttt{pre}_x(y)$ is such that:

y	y_0	y_1	y_2	y_3	y_4	$\ldots$
$\texttt{pre}_x(y)$	x	y_0	y_1	y_2	y_3	$\ldots$

To replace this operator, we define the following function:

```
let node_pre x =
  object
   last mem = x
   when get () returns (lm) where do lm = last mem done
   when set (v) returns () where
    do mem = v done
   with (get < set) # {}
  end
val node_pre :
  'a -> < set: 'a => unit; get: unit => 'a
         with (get < set) #  >
```

For every occurrence of $\texttt{pre}_{e_1}(e_2)$, an object "pre" is created by defining $\texttt{new}\, o = \texttt{node\_pre}(e_1)$ in the current block. Its value is read by writing $o.get()$ and a call to $o.set(e_2)$ is done afterward.

Note that this formulation coincides exactly with the way $\texttt{pre}_{e_1}(e_2)$ is compiled as a Moore machine in Lustre: it first emits its internal value then stores its argument and this is ensured by the scheduling policy (get < set) # {}.

Lustre initialization The arrow operator (e_1 -> e_2) emits the first time the first value of e_1 then it emits the current value of e_2 for the remaining instants:

e_1	x_0	x_1	x_2	x_3	x_4	$\ldots$
e_2	y_0	y_1	y_2	y_3	y_4	$\ldots$
e_1->e_2	x_0	y_1	y_2	y_3	y_4	$\ldots$

We define the following function

```
let init =
  object
   last init = true
   when step(x,y) returns (z) where
    do z = if last init then x else y
    and init = false
    done
   with step # {}
  end
```

*val init : < step: 'a * 'a => 'a with step # {} >*

Any occurrence of e_1 `->` e_2 must be replaced by `o.step(`e_1, e_2`)` provided `new o = init()` is defined in the context where the application occurs.

2.2 Scheduling Policies

We first define formally what is a schedule. A schedule S is a path of methods m with $S_1 < S_2$ as the sequence operator which states that methods of S_1 are called before the one of S_2. The empty path ϵ is the neutral element of the sequence operator.

$$S ::= S < S \mid m \mid \epsilon$$

A scheduling policy P associated to an object is its set of accepted schedules. It can be seen as the directions for use of the object. Since P is a set of schedules, usual operations like equality, inclusion, intersection or difference can be performed. A naive way of doing these computations is to represent P in a disjunctive normal form with $(\#)$ as the disjunction operator:

$$Norm(P) = \underset{i}{\#} S_i \quad \text{with } i \neq j \implies S_i \neq S_j$$

This representation is very expensive, so that we extend the sequence operator and introduce the shuffle operator $(||)$:

$$S < (S'_1 \# S'_2) = (S < S'_1) \# (S < S'_2) \tag{1}$$
$$(S_1 \# S_2) < S' = (S_1 < S') \# (S_2 < S') \tag{2}$$

$$(m < S) \mid\mid (m' < S') = \tag{3}$$
$$(m < (S \mid\mid (m' < S'))) \# (m' < (S' \mid\mid (m < S)))$$
$$(\underset{i}{\#} S_i) \mid\mid (\underset{j}{\#} S'_j) = \underset{i,j}{\#}(S_i \mid\mid S'_j) \tag{4}$$

Policy language

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

The disjunction operator $P_1 \# P_2$ states we either have schedules of P_1 or schedules of P_2, it is the union set operator. The sequence operator $P_1 < P_2$ is the set of all schedules composed of a schedule from P_1 followed by a schedule of P_2. The parallel operator $P_1 \mid\mid P_2$ stands for the parallel composition. In term of schedules, it is the *shuffle* operation, representing all the possible inter-leavings schedules from P_1 with schedules from P_2.

All the properties showed when extending operator of S equations (1,2,3,4) are properties of P. To sum up, the three operators are associative, $(<)$ is obviously non-commutative while $(||)$ and $(\#)$ are. Moreover, $(||)$ and $(<)$ are distributive over $(\#)$. All these properties make it easy to compute $Norm(P)$.

$$P_1 \mid\mid (P_2 \# P_3) = (P_1 \mid\mid P_2) \# (P_1 \mid\mid P_3)$$

2.3 Soundness and correction of scheduling policies

Consider an object c with memories x_i, object instances o_i, methods m_i and a scheduling policy P. During the typing process, two properties related to the scheduling policies have to be checked. The soundness and the correctness of P. The soundness will ensure that P prevents race conditions on shared memories x_i, while the correctness will ensure that the objects o_i are used according to their policies P_i. To this end, we introduce a small effect language C (C for constraint):

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

In the same way as P defined a set of schedule of method calls, C defines a set of schedule of effects. $\uparrow x$ states that variable x is read, $\uparrow \texttt{last}\, x$ that the last value of x is read, $\downarrow x$ that x is written and $o.m$ is the effect of calling the method m of the object o.

Projection and complement If C is a constraint and o is the name of an object, $C|_o$ is the projection of C on o defined by :

$$(C \mid\mid C')|_o = C|_o \mid\mid C'|_o \qquad (C \# C')|_o = C|_o \# C'|_o$$
$$(C < C')|_o = C|_o < C'|_o \qquad \epsilon|_o = \epsilon$$
$$(\downarrow x)|_o = \epsilon \qquad (\uparrow x)|_o = \epsilon$$
$$(\uparrow \texttt{last}\, x)|_o = \epsilon \qquad (o'.m)|_o = \epsilon \text{ with } o' \neq o$$
$$(o.m)|_o = m$$

The projection C_x of C on a shared variable x is defined similarily:

$$(C \mid\mid C')|_x = C|_x \mid\mid C'|_x \qquad (C \# C')|_x = C|_x \# C'|_x$$
$$(C < C')|_x = C|_x < C'|_x \qquad \epsilon|_x = \epsilon$$
$$(\downarrow x)|_x = \downarrow x \qquad (\downarrow x')|_x = \epsilon \text{ with } x' \neq x$$
$$(\uparrow x)|_x = \uparrow x \qquad (\uparrow x')|_x = \epsilon \text{ with } x' \neq x$$
$$(\uparrow \texttt{last}\, x)|_x = \uparrow \texttt{last}\, x \qquad (\uparrow \texttt{last}\, x')|_x = \epsilon \text{ with } x' \neq x$$
$$(o.m)|_x = \epsilon$$

We also define the complement C_o of C as the resulting constraint where every access to o has been eliminated.

$$(C \mid\mid C')_o = C_o \mid\mid C'_o \qquad (C \# C')_o = C_o \# C'_o$$
$$(C < C')_o = C_o < C'_o \qquad \epsilon_o = \epsilon$$
$$(\downarrow x)_o = \downarrow x \qquad (\uparrow x)_o = \uparrow x$$
$$(\uparrow \texttt{last}\, x)_o = \uparrow \texttt{last}\, x \qquad (o'.m)_o = o'.m \text{ with } o' \neq o$$
$$(o.m)_o = \epsilon$$

This is easily computed in one shot with $C|_o$ while C_o is equivalent to C for all future projections, but smaller and uncluttered from method calls of o.

Given an object definition with policy P and methods $m_1 ... m_n$, we compute the constraint C^{m_i} for each method definition of c by following syntactically the code of each method. The construction of C^{m_i} is built during typing. The resulting constraint for the object is obtained by substituting m_i by C^{m_i} in P, that is:

$$C = P[C^{m_1}/m_1; \ldots ; C^{m_n}/m_n]$$

Soundness C is sound, written $Sound(C)$, if for every shared memory x, $C|_x$ does not include $(\downarrow x < \downarrow x)$ nor $(\downarrow x < \uparrow x)$ nor $(\downarrow x < \uparrow \texttt{last}\, x)$. This also prevents $(\downarrow x \mid\mid \downarrow x)$ and $(\downarrow x \mid\mid \uparrow x)$. So the soundness ensures that there is no race condition among the writes and reads. Moreover, the constraint $\downarrow x < \uparrow \texttt{last}\, x$ ensures that the two values `last` x and x can be represented with only one store.

The soundness of a policy is a property attached to the definition of an object itself, e.g., two methods m_1 and m_2 can be put in parallel in a policy whenever they do not have concurrent writes.

Correctness C is correct, written $Correct(C)$, if for every object o_i, $C|_{o_i}$ is included in P_i.

Correctness is a property of the calling context of an object. A object is correctly used in it is used with a policy which is included in the set of schedules declared with the object.

2.4 Examples

A correct example:

```
let simple_colored_point x0 y0 c =
  object
    new p = point x0 y0
    last color = c
    when translate (nx,ny) returns (n, x, y) where
      do n = p.translate(nx,ny)
      in (x,y) = p.cartesian()
      done

    when color(c) returns () where
      do color = c done

    when lastcolor() returns (last color)

    when polar() returns (p.polar())

    with ((lastcolor # {}) < (color # {}))
         || (translate < (polar # {}))
  end
```

```
val simple_colored_point : float -> float -> 'a ->
  < polar: unit => float * float;
    lastcolor: unit => 'a;
    color: 'a => unit;
    translate: float * float => float * float
    with (lastcolor # {}) < (color # {})
         || (translate < (polar # {})) >
```

We compute the effect of each method:

$$\begin{array}{rcl} C^{translate} & = & p.translate < p.cartesian \\ C^{color} & = & \downarrow color \\ C^{lastcolor} & = & \uparrow \texttt{last}\, color \\ C^{polar} & = & p.polar \end{array}$$

We compute C by applying the substitutions to P and get:

$$\begin{array}{rl} C = & (\uparrow \texttt{last}\, color \# \epsilon) < (\downarrow color \# \epsilon) \\ & ||(p.translate < p.cartesian) < (p.polar \# \epsilon) \end{array}$$

and the projections :

$$\begin{array}{rcl} C|_{color} & = & (\uparrow \texttt{last}\, color \# \epsilon) < (\downarrow color \# \epsilon) \\ C|_{p} & = & translate < cartesian < (polar \# \epsilon) \end{array}$$

C is sound and correct, since the memory `color` is correctly used as is the object `p`. Indeed $C|_p \subset P_p : translate < cartesian$ and $translate < cartesian < polar$ are legal uses of the `point` object `p` which has the policy

$$(translate \# rotate) < ((cartesian \# \epsilon) \;||\; (polar \# \epsilon))$$

A wrong example:

```
let node wrong_use(b) returns (x,y) where
  new o = twomodes(0) in
  do x = o.up(1)
  and if b then do y = 0 done else do y = o.down(1) done
  done
```

This node is encoded with an object with only one method `step` with policy $step \# \epsilon$. The constraint for the body of `wrong_use` is:

$$C^{step} = o.up \;||\; (\epsilon \# o.down)$$

Thus:

$$C^{step}|_o = up \;||\; (\epsilon \# down)$$

which is not included in the policy associated to `o` ($up \# down$). The compiler rejects this code with the following error message:

```
Type error: the object o is used with policy
up || (down # {})
but is expected to have policy
up # down.
```

In the same way, the policy associated to the following object is unsound:

```
let w_up_down =
  object
    last o = 0
    when up(x) returns (o) where
      do o = last o + x done
    when down(x) returns (o) where
      do o = last o - x done
    with up < down
  end
Type error: the declared policy is incompatible with
the definition of modes. The shared variable o
may be written before its last value is read.
```

Indeed it computes

$$C^{up}|_o = \uparrow \texttt{last}\, o < \downarrow o \qquad C^{down}|_o = \uparrow \texttt{last}\, o < \downarrow o$$

which gives

$$C|_o = \uparrow \texttt{last}\, o < \downarrow o < \uparrow \texttt{last}\, o < \downarrow o$$

having among its errors the infringement $\downarrow o \;< \uparrow \texttt{last}\, o$.

3. Synchronous Semantics

The synchronous reaction semantics follows the existing formulation for the logical semantics of Esterel [4]. The run of a program is a sequence of reactions to external inputs. For that purpose, we define R as a reaction environment which associates values to variables (either regular or state variables). v is the set of values produced by a reaction. It may be either an immediate value (e.g., an integer value), a boolean (tt and ff), a pair (v, v), a combinatorial function or an object. The environment O records all the method calls done during the reaction, such that $O(o)$ is the set of called methods of the object o. w is well formed meaning that names of methods are pairwise distinct. w_1, w_2 is the concatenation of the two provided names do not intersect. An entry $m(v){=}v'$ states that the method m returns v' on input v. :

$$\begin{array}{rcl} R & ::= & \emptyset \mid R + [v/x] \mid R + [v/\texttt{last}\, x] \\ O & ::= & \emptyset \mid [w_1/o_1, ..., w_m/o_m] \\ v & ::= & i \mid tt \mid ff \mid (v, v) \mid \lambda x.e \\ & & \mid \texttt{object}\, fields\ objs\ modes\ \texttt{with}\ P\ \texttt{end} \\ w & ::= & [m_1(v_1){=}v'_1, ..., m_n(v_n){=}v'_n] \end{array}$$

If R is a reaction environment, we write $R(x)$ the value associated to x ($R + [v/y](x) = v$ if $x = y$ and $R(x)$ otherwise). We lift it to patterns and write $R(p)$. $R(e)$ returns a new expression where has been applied. $R + [(v_1, v_2)/(p_1, p_2)]$ is a shortcut for $R{+}[v_1/p_1]{+}[v_2/p_2]$. If R_1 and R_2 are two reaction environments, R_1, R_2 is the union of the two provided their domain do not intersect. If O_1 and O_2 are two environments, we define $O_1 \;||\; O_2$:

$$O_1 \;||\; O_2 = O_1, O_2 \qquad \text{if } Dom(O_1) \cap Dom(O_2) = \emptyset$$

$$O_1 + [w_1/o] \;||\; O_2 + [w_2/o] = (O_1 \;||\; O_2) + [(w_1, w_2)/o]$$

We define the next value of a state variable x:

$$\begin{array}{rl} Next_R(v)(x) = & R(x) \text{ if } x \in Dom(R) \\ = & v \text{ otherwise} \end{array}$$

The semantics is defined by two predicates:

$$R, O \vdash e_1 \xrightarrow{v} e_2 \qquad R, O \vdash d_1 \xrightarrow{R'} d_2$$

The first states that under the environment R and O, e_1 emits the value v and rewrites to e_2. The second one states that under R and O, the declaration d_1 produces the reaction environment R' and rewrites to d_2. These predicates are defined in figure 1.

(app) is the evaluation rule for the application of a stateless function (e.g., external operation like an integer addition). $(e_1\ e_2)$ produces a value v' and rewrites to $(e'_1\ e'_2)$ when (e_1) produces a function $\lambda x.e$, (e_2) produces a value v and (e) produces v' under the environment $R' + [v/x]$.

$$\text{(app)}\ \frac{R, O_1 \vdash e_1 \xrightarrow{\lambda x.e} e_1' \quad R, O_2 \vdash e_2 \xrightarrow{v} e_2' \quad R' + [v/x], \emptyset \vdash e \xrightarrow{v'} e}{R, O_1 \mid\mid O_2 \vdash e_1\ e_2 \xrightarrow{v'} e_1'\ e_2'} \qquad \text{(pair)}\ \frac{R, O_1 \vdash e_1 \xrightarrow{v_1} e_1' \quad R, O_2 \vdash e_2 \xrightarrow{v_2} e_2'}{R, O_1 \mid\mid O_2 \vdash (e_1, e_2) \xrightarrow{(v_1, v_2)} (e_1', e_2')}$$

$$\text{(if-t)}\ \frac{R, O \vdash e \xrightarrow{tt} e' \quad R, O_1 \vdash d_1 \xrightarrow{R'} d_1'}{R, O \mid\mid O_1 \vdash \texttt{if}\ e\ \texttt{then}\ d_1\ \texttt{else}\ d_2 \xrightarrow{R'} \texttt{if}\ e'\ \texttt{then}\ d_1'\ \texttt{else}\ d_2} \qquad \text{(if-f)}\ \frac{R, O \vdash e \xrightarrow{ff} e' \quad R, O_2 \vdash d_2 \xrightarrow{R'} d_2'}{R, O \mid\mid O_2 \vdash \texttt{if}\ e\ \texttt{then}\ d_1\ \texttt{else}\ d_2 \xrightarrow{R'} \texttt{if}\ e'\ \texttt{then}\ d_1\ \texttt{else}\ d_2'}$$

$$\text{(and)}\ \frac{R, R_2,\ O_1 \vdash D_1 \xrightarrow{R_1} D_1' \quad R, R_1,\ O_2 \vdash D_2 \xrightarrow{R_2} D_2'}{R, O_1 \mid\mid O_2 \vdash D_1\ \texttt{and}\ D_2 \xrightarrow{R_1, R_2} D_1'\ \texttt{and}\ D_2'} \qquad \text{(in)}\ \frac{R, O_1 \vdash D_1 \xrightarrow{R_1} D_1' \quad R, R_1,\ O_2 \vdash D_2 \xrightarrow{R_2} D_2'}{R, O_1 \mid\mid O_2 \vdash D_1\ \texttt{in}\ D_2 \xrightarrow{R_1, R_2} D_1'\ \texttt{in}\ D_2'} \qquad \text{(eq)}\ \frac{R + [v/p], O \vdash e \xrightarrow{v} e'}{R, O \vdash p = e \xrightarrow{[v/p]} p = e'}$$

$$\text{(im)}\ R, \emptyset \vdash i \xrightarrow{i} i \qquad \text{(read)}\ R + [v/x], \emptyset \vdash x \xrightarrow{v} x \qquad \text{(m-call)}\ \frac{R, O \vdash e \xrightarrow{v} e'}{R, O \mid\mid [m(v){=}v'/o] \vdash o.m(e) \xrightarrow{v'} o.m(e')}$$

$$\text{(fun)}\ R, \emptyset \vdash \lambda x.e \xrightarrow{R(\lambda x.e)} \lambda x.e \qquad \text{(last)}\ R + [v/\texttt{last}\ x], \emptyset \vdash \texttt{last}\ x \xrightarrow{v} \texttt{last}\ x$$

$$\text{(modes)}\ R, \emptyset \vdash \texttt{object}\ \mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \texttt{with}\ P\ \texttt{end} \xrightarrow{R(\texttt{object}\ \mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \texttt{with}\ P\ \texttt{end})} \texttt{object}\ \mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \texttt{with}\ P\ \texttt{end}$$

$$\text{(var)}\ \frac{R, O \vdash d \xrightarrow{R' + [v/x]} d'}{R, O \vdash \texttt{var}\ x\ \texttt{in}\ d \xrightarrow{R'} \texttt{var}\ x\ \texttt{in}\ d'} \qquad \text{(new)}\ \frac{R, \emptyset \vdash e \xrightarrow{v} e' \quad R, O_1 \vdash v \overset{w}{\rightsquigarrow} v' \quad R, O_2 + [w/o] \vdash d \xrightarrow{R'} d'}{R, O_1 \mid\mid O_2 \vdash \texttt{new}\ o = e\ \texttt{in}\ d \xrightarrow{R'} \texttt{new}\ o = v'\ \texttt{in}\ d'}$$

Figure 1. The Synchronous Semantics

(pair) pair's members are reduced in parallel.

(if-t) and (if-f) are for conditionals. The body (d_1) is evaluated if the condition evaluates to true (noted tt). Otherwise, (d_2) is evaluated.

(and), (in) and (eq) are respectively for equations (D_1 `and` D_2), (D_1 `in` D_2) and ($p = e$). Two equations (D_1 `and` D_2) react in parallel by producing an environment R_1, R_2. In doing that D_1 sees what D_2 is producing and conversely. On the contrary, (D_1 `in` D_2) is evaluated in sequence during the reaction. An equation ($p = e$) reacts by producing an environment $[v/p]$ provided e produces v and $R(p) = v$.

(im) and (read) define respectively the reaction rules for immediate values (i) and variables (x).

(last) defines the access to a state variable `last` x.

(m-call) defines a call to the method m. This reaction is possible when $m(v){=}v'$ is defined in the environment of method calls.

(modes) defines the value of an object.

(new) For an object definition, e is evaluated into v. Then, the result v reacts producing a collection of method calls w corresponding to those used in d.

We define the predicate $R, O \vdash v \overset{w}{\rightsquigarrow} v'$ stating that under R and O, v reacts by producing w and rewrites to v'. It is based on the following predicates:

$$R, O \vdash \mathit{fields} \xrightarrow{R'} \mathit{fields}'$$
$$R, O \vdash \mathit{objs} \xrightarrow{O'} \mathit{objs}'$$
$$R, O \vdash \mathit{modes} \xrightarrow{w} \mathit{modes}'$$

Their definition is given in figure 2.

(modes-react) makes fields, local object declarations and modes react in parallel. Fields are evaluated in the closure environment R_o extended with R' which updates some of the state variables of the object. Local objects are evaluated into O' provided O' is itself computed by the reaction of modes. The reaction of modes may call itself methods of local objects.

(last-def) The next value for x is v' if x has been computed during the reaction or it keeps its previous value v.

(new-def) is similar to the rule **(new)**.

(when-def) The definition of a method m changes only if there is a call to m, that is, $m(v){=}v'$ is used. The reaction of b may define new values for state variables (R').

3.1 The Type System

The type language is defined below. We borrow notations from the type system by Rémy & Vouillon for ML [15]. Nonetheless, to simplify the presentation, we make the use of row variables explicit.

A type-scheme (σ) is a type t quantified over type variables ($\alpha_1, ..., \alpha_n$) and row variables ($\rho_1, ..., \rho_m$). A type can be a function type ($t_1 \rightarrow t_2$), a product type ($t_1 \times t_2$), a constructed type ($c(t_1, ..., t_n)$ where c is a name of the type and range over a denumerable set or the type of an object ($\{r\}$). r stands for a (possibly empty) sequence of method names with their types together with a policy P.

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

A typing environment H is defined in the following way:

$$\begin{array}{lcl} H & ::= & \emptyset \mid H + \texttt{self} : t \\ & & \mid H + \texttt{last}\ x : t \\ & & \mid H + \texttt{new}\ o : t \\ & & \mid H + x : \sigma \end{array}$$

We make several name spaces in the environment. The type entry `self` $: t$ defines the current object to have the type t; `last` $x : t$

$$\text{(modes-react)}\ \frac{R_o + R', \emptyset \vdash \mathit{fields} \xrightarrow{R_l} \mathit{fields}' \quad R_o + R_l, O \vdash \mathit{objs} \xrightarrow{O'} \mathit{objs}' \quad R_o + R' + R_l, O' \vdash \mathit{modes} \xrightarrow{w} \mathit{modes}'}{R, O \vdash \texttt{object}\,\mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \texttt{with}\ P\,\texttt{end} \overset{w}{\rightsquigarrow} \texttt{object}\,\mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \texttt{with}\ P\,\texttt{end}}$$

$$\text{(last-def)}\ \frac{R, \emptyset \vdash e \xrightarrow{v} e' \quad \mathit{Next}_R(v)(x) = v'}{R, \emptyset \vdash \texttt{last}\,x = e \xrightarrow{[v/\texttt{last}\,x]} \texttt{last}\,x = v'} \qquad \text{(when-def}_a\text{)}\ R, \emptyset \vdash \texttt{when}\,m(p)\,\texttt{returns}\,(q)\ b \xrightarrow{\emptyset} \texttt{when}\,m(p)\,\texttt{returns}\,(q)\ b$$

$$\text{(new-def)}\ \frac{R, \emptyset \vdash e \xrightarrow{v} e' \qquad R, O \vdash v \overset{w}{\rightsquigarrow} v'}{R, O \vdash \texttt{new}\,o = e \xrightarrow{[w/o]} \texttt{new}\,o = v'} \qquad \text{(when-def}_b\text{)}\ \frac{R + R' + [v/p], O \vdash b \xrightarrow{R'} b' \quad v' = R'(q)}{R + R', O \vdash \texttt{when}\,m(p)\,\texttt{returns}\,(q)\ b \xrightarrow{m(v)=v'} \texttt{when}\,m(p)\,\texttt{returns}\,(q)\ b'}$$

Figure 2. Synchronous Reaction Rules for Objects

defines the shared memory $\texttt{last}\,x$ with type t; $\texttt{new}\,o : t$ defines the object o with its type t; $x : \sigma$ defines a name x to have the type scheme σ. $H_1 + H_2$ is the union of the two typing environment, provided their domains do not intersect.

DEFINITION 1 (Type Instantiation and Generalization). *We define the instantiation relation $t \leq \sigma$ between a type an a type scheme. $t \leq \sigma$ holds when there is a substitution of the quantified variables of σ which returns t. The generalization of a type t in an environment H quantifies all type and row variables which do not appear free in H. $FV(\sigma)$ stands for the free type variables $\{\alpha_1, ..., \alpha_n\}$ of σ whereas $FR(\sigma)$ stands for the free row type variables $\{\rho_1, ..., \rho_m\}$ of σ. These functions are lifted to type-environments.*

Instantiation:	$t[\vec{t'}/\vec{\alpha}][\vec{r}/\vec{\rho}] \leq \forall \vec{\alpha}.\vec{\rho}.t$
Generalization:	$gen_H(t) = \forall \alpha_1, ..., \alpha_n.\forall \rho_1, ..., \rho_m.t$
	where $\{\alpha_1, ..., \alpha_n\} = FV(t) - FV(H)$
	and $\{\rho_1, ..., \rho_m\} = FR(t) - FR(H)$

Typing is defined by the following type judgments:

- $H, C \vdash e : t$ states that the expression e has type t under the type environment H and scheduling constraints C.
- $H, C \vdash b : H'$ states that the block b produces the type environment H' under the type environment H and scheduling constraints C.
- $H, C \vdash D : H'$ states that the collection of equations D produces the type environment H' under the environment H and constraints C.

The definitions of these predicates are given in figure 3.

(taut) A variable stored with type scheme σ in the environment can be used with an instantiated type t. No scheduling constraint is necessary so we write ϵ in the constraint.

(taut-c) and (taut-l) A read to a state variable $\texttt{last}\,x$ of type t is of type t and we add $\uparrow x$ in the scheduling constraints. Similarly, reading $\texttt{last}\,x$ adds $\uparrow \texttt{last}\,x$.

(im) The fourth rule is for immediate constants.

(fun) In our kernel language, functions cannot access shared state variables nor objects defined above. We impose that the body e has the empty constraint e.

(app) and (pair) The resulting constraint is the parallel composition of the two.

(if) The control structure ($\texttt{if}\ \ e\ \ \texttt{then}\ \ d_1\ \ \texttt{else}\ \ d_2$) is typed classically. From the scheduling constraint point-of-view, the constraints of the two branches are exclusive ($C_1 \# C_2$) but C must be scheduled before one of the two.

(and) and (in) deal with parallel ($D_1\ \ \texttt{and}\ \ D_2$) and sequential composition ($D_1\ \ \texttt{in}\ \ D_2$). In the first case, the resulting constraint put the two constraints in parallel whereas they are put in sequence in the remaining one.

(eq-l) For an equation $x = e$ where x is a state variable and C is the scheduling constraint for e, we generate the constraint $C < \downarrow x$ stating that C appears before the write to x.

(eq) states that an equation $p = e$ defines a typing environment $[p : t]$ if p and e are of type t.

(var) types a local declaration ($\texttt{var}\,x\ \ \texttt{in}\ \ d$). This is obtained by typing d, supposing that x is of type t.

(m-call) and (new-in) deal with method call and object instantiation. They are thus the most important rules of the system. When calling a method m from object o, we check that o is defined and has a method with a valid type. Moreover, we generate a scheduling constraint stating that $o.m$ has been called after C.

(new-in) first checks that e is an object. Then, it types the body d. For that, it generates a constraint C. Let $P = C|_o$ the policy obtained by projecting C on name o. The object o is correctly used if P belongs to the policy of o, that is, o has a type of the form: $\{r\ \texttt{with}\ P\}$. C_o is the constraint C obtained by erasing accesses to object o.

o may have a greater policy than P (for the inclusion order) but it cannot be less.

(modes) is the typing rule for an object expression.

(last), (new) and (when) are used to type state variable definitions ($\texttt{last}\,x = e$), local object instantiations ($\texttt{new}\,o = e$) and the definition of methods ($\texttt{when}\,m(p)\ \texttt{returns}\ (q)\ b$).

3.2 Unification

The type system rely on the unification algorithm for row types developed by Rémy & Vouillon [15]. Our contribution is to show how scheduling policies enter in this setting. Internally, the types for objects are represented as:

$$\begin{array}{lcl} t & ::= & \{r\} \\ r & ::= & \emptyset \mid m : t, r \mid r\ \texttt{with}\ P \mid \rho \end{array}$$

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

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

$$\text{(taut)}\ \frac{t \le \sigma}{H + x : \sigma, \epsilon \vdash x : t} \quad \text{(taut-c)}\ H + \mathtt{last}\,x : t, \uparrow x \vdash x : t \quad \text{(taut-l)}\ H + \mathtt{last}\,x : t, \uparrow \mathtt{last}\,x \vdash \mathtt{last}\,x : t \quad \text{(im)}\ H, \epsilon \vdash i : \mathtt{int}$$

$$\text{(fun)}\ \frac{H + x : t_1, \epsilon \vdash e : t_2}{H, \epsilon \vdash \lambda x.e : t_1 \to t_2} \quad \text{(app)}\ \frac{H, C_2 \vdash e_2 : t_1 \to t_2 \quad H, C_1 \vdash e_1 : t_1}{H, C_2 \mid\mid C_1 \vdash e_2\ e_1 : t_2} \quad \text{(pair)}\ \frac{H, C_1 \vdash e_1 : t_1 \quad H, C_2 \vdash e_2 : t_2}{H, C_1 \mid\mid C_2 \vdash (e_1, e_2) : t_1 \times t_2}$$

$$\text{(if)}\ \frac{H, C \vdash e : \mathtt{bool} \quad H, C_1 \vdash d_1 : H_0 \quad H, C_2 \vdash d_2 : H_0}{H, C < (C_1 \,\#\, C_2) \vdash \mathtt{if}\ e\ \mathtt{then}\ d_1\ \mathtt{else}\ d_2 : H_0}$$

$$\text{(and)}\ \frac{H, C_1 \vdash D_1 : H_1 \quad H, C_2 \vdash D_2 : H_2}{H, C_1 \mid\mid C_2 \vdash D_1\ \mathtt{and}\ D_2 : H_1 + H_2} \quad \text{(in)}\ \frac{H, C_1 \vdash D_1 : H_1 \quad H, C_2 \vdash D_2 : H_2}{H, C_1 < C_2 \vdash D_1\ \mathtt{in}\ D_2 : H_1 + H_2}$$

$$\text{(eq-l)}\ \frac{H, C \vdash e : t}{H + \mathtt{last}\,x : t,\ C < \downarrow x \vdash x = e : [x : t]} \quad \text{(eq)}\ \frac{H, \epsilon \vdash p : t \quad H, C \vdash e : t}{H, C \vdash p = e : [p : t]} \quad \text{(var)}\ \frac{H + x : t, C \vdash d : H_0}{H, C \vdash \mathtt{var}\,x\ \mathtt{in}\ d : H_0}$$

$$\text{(m-call)}\ \frac{H + \mathtt{new}\,o : \{m : t_1 \to t_2, r\}, C \vdash e : t_1}{H + \mathtt{new}\,o : \{m : t_1 \to t_2, r\},\ C < o.m \vdash o.m(e) : t_2} \quad \text{(pair-pat)}\ \frac{H, \epsilon \vdash p_1 : t_1 \quad H, \epsilon \vdash p_2 : t_2}{H, \epsilon \vdash p_1, p_2 : t_1 \times t_2}$$

$$\text{(new-in)}\ \frac{H, \epsilon \vdash e : \{r\ \mathtt{with}\ C|_o\} \quad H + \mathtt{new}\,o : \{r\ \mathtt{with}\ C|_o\}, C \vdash d : H_0}{H, C_o \vdash \mathtt{new}\,o = e\ \mathtt{in}\ d : H_0}$$

$$\text{(modes)}\ \frac{H \vdash \mathit{fields} : H_0 \quad H, s(P) \vdash \mathit{objs} : H_1 \quad H + H_0 + H_1 + \mathtt{self} : \{r\ \mathtt{with}\ P\} \vdash \mathit{modes} : r, s \quad \mathit{Sound}(s(P))}{H, \epsilon \vdash \mathtt{object}\,\mathit{fields}\ \mathit{objs}\ \mathit{modes}\ \mathtt{with}\ P\ \mathtt{end} : \{r\ \mathtt{with}\ P\}}$$

$$\text{(last)}\ \frac{\forall i \in I.\ H, \epsilon \vdash e_i : t_i}{H \vdash (\mathtt{last}\,x_i = e_i)_{i \in I} : [\mathtt{last}\,x_i : t_i]_{i \in I}} \quad \text{(new)}\ \frac{\forall i \in J.\ H, \epsilon \vdash e_j : \{r_j\ \mathtt{with}\ C|_{o_j}\}}{H, C \vdash (\mathtt{new}\,o_j = e_j)_{j \in J} : [\mathtt{new}\,o_j : \{r_j\ \mathtt{with}\ C|_{o_j}\}]_{j \in J}}$$

$$\text{(when)}\ \frac{\forall k \in K.\ H_k \vdash p_k : t_k \quad H + H_k, C^k \vdash b_k : H'_k \quad H + H_k + H'_k \vdash q_k : t'_k}{H, C \vdash (\mathtt{when}\,m_k(p_k)\,\mathtt{returns}\,(q_k)\ b_k)_{k \in K} : [m_k : t_k \to t'_k]_{k \in K} + [m_k : C^k]_{k \in K}}$$

Figure 3. The Type System

with the well-formation properties that $m : t, r$ is well formed iff m does not appear in r. This invariant is maintained during typing. Using above properties, a row type can be normalized into: $\{m_1 : t_1, ..., m_n : t_n\ \mathtt{with}\ P; \rho\}$ or $\{m_1 : t_1, ..., m_n : t_n\ \mathtt{with}\ P\}$. The unification algorithm of Rémy & Vouillon is modified accordingly.

We illustrate its use on the following example:

```
let node g(f)(x) returns (r) where
  new o1 = f (x) in
  new o2 = f (x+1) 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
```

Suppose that $H = [f : \alpha_1, x : \alpha_2]$ where α_1, α_2 are fresh type variables. After having typed the declarations of o_1 and o_2, we have: $H' = H + [o_1 : \{\rho\}, o_2 : \{\rho\}]$. Then, the typing of the first branches states that:

$$H', (o_1.m \mid\mid o_2.m) \vdash o_1.m(x) + o_2.m(x) : \mathtt{int}$$
$$\text{and} \quad H', (o_1.n \mid\mid o_1.m) \vdash o_1.n(x) + o_1.m(x) : \mathtt{int}$$

with $\alpha_1 = \mathtt{int} \to \{\rho\}$. We obtain the final constraint:

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

Projecting it on o_2 , we get:

$$C|_{o_2} = (\epsilon \mid\mid m) \,\#\, (\epsilon \mid\mid \epsilon) = m \,\#\, \epsilon$$

Thus,

$$\rho = m : \mathtt{int} \to \mathtt{int},\ n : \mathtt{int} \to \mathtt{int}\ \mathtt{with}\ m \,\#\, \epsilon,\ \rho'$$

Then, we compute the resulting constraint C' by eliminating o_2:

$$C' = o_1.m \,\#\, (o_1.n \mid\mid o_1.m)$$
$$C'|_{o_1} = m \,\#\, (n \mid\mid m)$$

Thus, $\rho' = n \mid\mid m,\ \rho''$. Finally, we get the type:

$$g : \forall \rho.(\mathtt{int} \to \{m : \mathtt{int} \to \mathtt{int}, n : \mathit{int} \to \mathtt{int}\ \mathtt{with}\ \epsilon \,\#\, m \,\#\, (n \mid\mid m), \rho\} \to \mathtt{int} \to \mathtt{int})$$

4. Discussion

4.1 Implementation

Based on the presented material, a complete compiler has been implemented. Its organization is given in figure 4. The actual language provides more programming constructs than the kernel considered in Section 2. In particular, it is possible to mix functions and nodes directly. A node is a shortcut for an object with a single step function. Nodes receive a special type $t_1 \Rightarrow t_2$ whereas the type of functions is $t_1 \to t_2$. The language also provides richer control-structures (e.g., hierarchical automata as proposed in [7]), Lustre

Administrative code	Ocaml LOC
abstract syntax, printers, lexer, parser	2600
main driver (e.g., modules, symbol tables, loader)	450
Basis graph structures, cycle detection, free variables	100
Scoping renaming to have unique names	550
Typing unification and simplifications	600
policies (inclusion, equality, difference)	400
causality analysis	200
main typing engine	1200
auxiliary functions (e.g., error message, printers)	450
Source-to-source rewriting automata elimination	450
signal elimination	250
completion (adding explicit equations $x = \texttt{last}\, x$)	150
elimination of Lustre operators (e.g., `pre`)	300
elimination of nodes and translation into objects	450
Emission of sequential code scheduling	150
abstract syntax + printers	400
translation to Ocaml	250

Figure 4. Organization of the compiler

operators (`pre` and `->`)) and signals [6]. Automata constructs are treated like conditionals. At most one state is active at a time so the scheduling constraint of an automaton with n states is of the form $C_1 \# ... \# C_n$.

Once typing is done, the compilation is done through a sequence of source-to-source transformations, each of these eliminating one programming construct. The first step eliminates automata by translating them into more basic control-structures. The following one eliminates signals. The next one completes partial definitions. Completion consists in transforming an equation:

```
if c then do x = 1 done else do done
```

into:

```
if c then do x = 1 done else do x = last x done
```

The next one eliminates Lustre primitives (`pre` and `->`) following the encoding given in Section 2. The final step translates remaining constructs such as nodes into objects. In doing so, the result of each transformation is a valid program and could thus be given again to the type-checker. The last step of the compiler produces sequential code. We have only considered the generation of Ocaml code in the present implementation. This step is made simple since instanciated objects in the source language are translated into objects of the target language.

4.2 Shared Variables and Effects

The present extension allows to specify several behaviors sharing a common resource inside what we have called an "object". The resource is encapsulated and can only be accessed through methods in a quite standard fashion. The scheduling policy attached to the object specifies the possible use of it. This ability to separate the specification from the actual use increases the expressiveness of a language such as Lustre, avoiding extra wiring which should be needed otherwise.

One must notice that instantiated objects are not first-class values as this is typically the case in object-oriented languages. For example, the following program is statically rejected:

```
let node f(x) returns (y) where
  do y = x.m1(1) + x.m2(2) done
```

One should write instead:

```
let node f(x) returns (y) where
  new o = x in
  do y = o.m1(1) + o.m2(2) done

val f : < m1 : int -> int m2 : int -> int
          with m1 || m2 > -> int
```

An alternative to the use of objects to increase the modularity of a synchronous data-flow language would rely on the introduction of references with imperative updates. For example, the pseudo-code:

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

would define a function taking a reference to a shared variable $\texttt{last}\, x$ and making an imperative update on it. An object sharing a common state could be emulated by:

```
let twomodes x0 =
  last o = x in
  (fun (y) returns (o) do o = last o + y done),
  (fun (y) returns (o) do o = last o - y done)
```

The absence of concurrent write (and more generally the soundness property of section 2.3) relies on an effect-type system [16], e.g., taking C as the effect language. Following this, `twomodes` would receive a type signature of the form:

$$\texttt{int} \rightarrow \exists o : \texttt{int}.(\texttt{int} \xrightarrow{\uparrow\texttt{last}\, o < \downarrow o} \texttt{int}) \times (\texttt{int} \xrightarrow{\uparrow\texttt{last}\, o < \downarrow o} \texttt{int})$$

stating that it returns a pair of functions with respectively effects $(\uparrow\texttt{last}\, o < \downarrow o)$ and $(\uparrow\texttt{last}\, o < \downarrow o)$ where o is quantified existencially. Introducing a type system with effects to ensure the absence of data-races in a synchronous data-flow language would lead to a language more expressive that what is presented here. This solution has been experimented by Hamon in his PhD. thesis [9]. Nonetheless, the resulting type-system is more complex than the technique presented, in particular if higher-order is allowed and type signatures become difficult to read. It is also not conservative in the sense that most type signatures are modified even for functions which are side-effect free (e.g., the signature of a higher-order function).

4.3 Inheritance

The extension we have proposed is nonetheless limited since it does not completely allow several independent teams to develop and test modes (e.g., `up` and `down`). A natural extension is to consider a class mechanism and inheritance as a way to build systems incrementally. The team `up` and `down` define their own class:

```
class up =
  last x = 0
  when up(y) returns (x)
    do x = last x + y done
end

class down =
  last x = 0
  when down(y) returns (x)
    do x = last x - y done
end
```

When typing each mode, it is possible to keep the way `x` is accessed (e.g., the scheduling constraint for up is $C^{\text{up}} = \uparrow\texttt{last}\, x < \downarrow x$). Then, the integration of the two to form the class updown becomes:

```
class two =
  inherit up
  inherit down
end
```

The absence of concurrent writes can be checked simply. In this idea of inheritance, the important point is the mapping of the shared variable, if one inherit from two objects, the aim is to create new sharing of state variables. The fact that up and down share then the same state variable should certainly be explicited by the inheritance mechanism and to write:

```
class two =
  object
    inherit up, down with up.x=down.x
  end
```

The synchronization on x is now made explicit provided the interface of up and down make the name x public. Extending the present proposal to accont for inheritance is a matter of future work.

4.4 Policies and Initialization

A policy is a particular case of a contract [13]. Being unable to express dynamic properties, they can be ensured during typing. Between them and a general notion of policies, a usefull and yet simple extension towards dynamic policies is to add a special treatment for the first reaction [14]. A natural way to account it is to introduce a policy P_1 `->` P_2 to distinguish the initialization policy P_1 from the permanent one P_2.

4.5 Related Work

The interest of a modularity construct paired with an interface language to specify valid behaviors in the context of real-time systems has been considered largely. In [1], Alur & Henzinger introduce *Reactive Modules* as a way to specify temporal properties of concurrent systems (synchronous or asynchronous). With *Interface automata* [8] possible activations are expressed as automata. It is thus possible to describe rich dynamic properties over execution traces which are, in turn, verified by model-checking techniques. In comparison, scheduling policies are unable to express dynamic properties and describe only star free languages. Being less expressive, their static verification can be done by simple typing techniques, they are fully modular and apply to higher-order programs.

This work is also related to the *42* model of Maraninchi & Bouhadiba [11]. Their purpose is to develop a general model of component for embedded software, general enough to express various models of computations (e.g., synchronous, asynchronous, FIFO communication). A component is made of a graph of nodes and a controller which describes how blocks are activated. A controller can be arbitrarily complex and express more general scheduling policies on sub-components than what we do. In comparison, our proposal is dedicated to synchronous systems. A scheduling policy relies on the fact that each method is atomic and that a valid reaction is a finite composition of method calls.

The proposed notion of object generalizes the basic notion of a reaction in a language such as Lustre. Instead of having a unique *step* function for a node, we provide a collection of atomic functions and define a step as a composition of these atomic functions. There is, in particular, no need for a Kleene star operation.

The idea of policies was somehow hidden in the Signal language [3]. Indeed, a Signal process defines a relation between the presence/absence of its input/outputs. This relation is a property on clocks and expresses precedence constraints between signals, thus set of possible scheduling of the process. It would be possible to define a system with modes in Signal, for example, by putting the definitions of each mode in parallel and associating them different clocks. Nonetheless, checking the exclusion of modes and that process are correctly instantiated would rely on the boolean analysis done by the clock-calculus. In comparison, we provide a special programming construct to clearly identify modes. The absence of concurrent writes is checked by simpler means, it is defined as a typing problem and is compatible with higher-order.

5. Conclusion and Future Work

In this paper, we have introduced a calculus of synchronous reactive objects. The calculus builds on existing synchronous data-flow languages such as Lustre and his functional variant Lucid Synchrone. The main novelty is to provides a new programming constructs which allows to defines systems with modes. This construction is reminiscent to object-oriented features: a mode can be seen as a method and an object gathers a possible set of modes. In order to ensure the determinacy of programs (that is, the absence of data-races), we introduce a notation to express *scheduling policies*. A scheduling policy is described by a simple regular expression giving the possible combination of methods to build a valid synchronous reaction. We have shown that scheduling policies can be automatically verified through a variant of an existing type-systems with rows.

References

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[3] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] G. Berry. The constructive semantics of pure esterel. Draft book, 1999.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 2005. Special Issue on Embedded Software.

[6] J.L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[7] J.L. Colaço, B. Pagano, and M. 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.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference*, page 109120, New York, NY, USA, 2001. ACM Press.

[9] G. Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 14 novembre 2002.

[10] D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.

[11] F. Maraninchi and T. Bouhadiba. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Sixth ACM International Conference on Generative Programming and Component Engineering (GPCE'07)*, Salzburg, Austria, October 2007.

[12] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[13] B. Meyer. *Eiffel: An Introduction*. Interactive Software Eng, 1988.

[14] D. Pilaud. Personnal communication, March 2009.

[15] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[16] J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.