Synchronous Objects with Scheduling Policies

Introducing safe shared memory in Lustre

Paul Caspi
VERIMAG
Grenoble
Paul.Caspi@imag.fr

Jean-Louis Colaco
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

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. 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).
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

let 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 names, types, ranges, required preci-
sions and timing characteristics and any other convenient high level
shared features. This is not possible in a purely data-flow context
as modes share several state variables as it is shown in the following
example:

let twomodes x0 =
  object
    last x = x0
    last y = y0
    when translate (nx, ny) returns (nb_translations)
      do nb_translations = 1 -> pre nb_translations + 1
      and x = last x +. nx
      and y = last y +. ny
    done
  end

let main: unit => int

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

2All 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.

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

let 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)
      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)
    where
dd = x +. y
    and tetha = if x > 0.0 then (y /. x)
    then, it may call
the policy cartesian taking (x, y)
  else if x = 0.0 then 2.0
  else (-. y /. x)
  done
  with (translate # rotate)
    < ((cartesian # {}) || (polar # {}))
end

let point : float => float =>
  < polar: unit => 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
are expressed in the following language:

\[
P ::= P || P | P # P | P < P | P |
\]

\(P_1 \parallel P_2\) 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 \(\bot\) 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.

We define a core data-flow language extended with the ability to define methods but whose policy should contains the inferred policy.

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.

\[\begin{align*}
d &::= \text{let } x = e \mid d \mid d \\
p &::= p \mid p \\
b &::= \text{var } x \in b \mid \text{new } x = e \in b \mid D \\
e &::= e \mid \text{var } x \mid \text{last } x \mid (e) \mid e e \mid \text{new } x \mid e . o . m(e) \\
D &::= p \mid e \mid D \mid D' \mid \text{in } D' \\
f &::= \text{let } g h x y \text{ returns } (v) \text{ where new } o = h(x) \text{ in automaton } v \text{ with } y \\
\text{val } g : (a -> < up: b -> \text{int}; \text{down: } b' = \text{int} ... \text{ with } (\text{# up } \text{& down } >)) -> \\
aafter b -> \text{int} \text{ end}
\end{align*}\]

The function \(g\) can be applied to any function \(h\) returning an object with at least two methods \(\text{up}\) and \(\text{down}\) with policy \(\text{up} \# \text{down}\). The notation \(\text{...}\) states that \(h\) is a function returning an object which contains, at least, methods \(\text{up}\) and \(\text{down}\) 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 (var x in b) and instantiated objects (new x = e in b) or a collection of equations (D). A collection can be an equation (p = e), a parallel (D1, D2) or sequential composition (D1; D2). Finally, it can be based on the conjunction of two equations (if e then D1 else D2) according to a boolean condition (e).

An expression (e) may be a constant (v), a variable (x), a read access to a shared variable (last x), a pair (e1, e2), a function application (e1 e2), a function (λx.e), a method call (o.m(e)) of an object o with name m and, finally, the definition of a system with modes (object fields objs modes with P 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.

\[\begin{align*}
\text{d} &::= \text{let } x = e \mid d \mid d \\
\text{p} &::= p \mid p \\
\text{b} &::= \text{var } x \in b \mid \text{new } x = e \in b \mid D \\
\text{e} &::= e \mid \text{var } x \mid \text{last } x \mid (e) \mid e e \mid \text{new } x \mid e . o . m(e) \\
\text{D} &::= p \mid e \mid D \mid D' \mid \text{in } D' \\
\text{if} &::= \text{if } e \text{ then } D \text{ else } D' \\
\text{fields} &::= \text{last } x = e ... \text{last } x = e
\end{align*}\]

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

\[\begin{align*}
\text{let } \text{node}_f = \\
&\quad \text{object} \\
&\quad \text{when } \text{step}(p) \text{ returns } (q) \text{ b} \\
&\quad \text{with step #} \{} \\
\end{align*}\]

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

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

\[\begin{align*}
\text{pre}_x(y) &::= y \mid y_0 \mid y_1 \mid y_2 \mid y_3 \mid y_4 \mid ... \\
\end{align*}\]

To replace this operator, we define the following function:

\[\begin{align*}
\text{let } \text{node\_pre}_{x} = \\
&\quad \text{object} \\
&\quad \text{last } \text{mem} = x \\
&\quad \text{when } \text{get}() \text{ returns } (l m) \text{ where } \text{do } l m = \text{last mem done} \\
&\quad \text{when } \text{set}(v) \text{ returns } () \text{ where } \text{do } v = \text{v done} \\
&\quad \text{with (get < set) #} \{} \\
\text{end} \\
\text{val } \text{node\_pre}_x : \\
&\quad (a -> < set: 'a -> \text{unit}; \text{get: } \text{unit} -> 'a \\
&\quad \text{with (get < set) #} >)
\end{align*}\]

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

Note that this formulation coincides exactly with the way \(\text{pre}_x(e)\) 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 \(\text{get < set} \# \{}\).

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

\[\begin{align*}
e_1 &::= X_0 \mid X_1 \mid X_2 \mid X_3 \mid X_4 \mid ... \\
e_2 &::= Y_0 \mid Y_1 \mid Y_2 \mid Y_3 \mid Y_4 \mid ... \\
e_1 \rightarrow_{e_1} e_2 &::= X_0 \mid Y_1 \mid Y_2 \mid Y_3 \mid Y_4 \mid ...
\end{align*}\]

We define the following function:

\[\begin{align*}
\text{let } \text{init} = \\
&\quad \text{object} \\
&\quad \text{last init = true} \\
&\quad \text{when } \text{step}(x,y) \text{ returns } (z) \text{ where } \text{do } z = \text{if last init then } x \text{ else } y \\
&\quad \text{and init = false} \text{ done} \\
&\quad \text{with step #} \{} \\
\end{align*}\]
Any occurrence of \( e_1 \rightarrow e_2 \) must be replaced by \( o.\text{step}(e_1, e_2) \) provided \( \text{new } o = \text{init}() \) 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 \( S_i < S_j \), as the sequence operator which states that methods of \( S_i \) are called before the one of \( S_j \). The empty path is the neutral element of the sequence operator:

\[ S ::= S \triangleleft S \mid m \]

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 disjunction normal form with \((\#)\) as the disjunction operator:

\[ \text{Norm}(P) = \bigoplus_i S_i \text{ with } i \neq j \implies S_i \neq S_j \]

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

\[ S < (S' \# S'_2) = (S < S') \# (S < S'_2) \quad (1) \]
\[ (S_1 \# S_2) < S' = (S_1 < S') \# (S_2 < S') \quad (2) \]
\[ (m < S) \| (m' < S') = (m < (S \| (m' < S'))) \# (m' < (S' \| (m < S))) \quad (3) \]
\[ (\#S_i) \# (\#S'_j) = \#(S_i \| S'_j) \quad (4) \]

Policy language

\[ P ::= P \| P \# P \| P < P \| m \]

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 \( P_1 || P_2 \) stands for the parallel composition. In term of schedules, it is the shuffle operation, representing all the possible interleavings 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, \((\cdot)\) is obviously non-commutative while \((\|)\) and \((\#)\) are. Moreover, \((\|)\) and \((\cdot)\) are distributive over \((\#)\). All these properties make it easy to compute \( \text{Norm}(P) \).

\[ P_1 || (P_2 \# P_3) = (P_1 || P_2) \# (P_1 || P_3) \]

2.3 Soundness and correctness of scheduling policies

Consider an object \( o \) with memories \( x_i \), object instances \( a_i \), 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 \( C \) for constraint:

\[ C ::= C || C < C || C < C || x \mid \downarrow x \mid \downarrow \text{last } x \mid o.m \]

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, \( \downarrow x \) that the last value of \( x \) is read, \( \downarrow \text{last } x \) 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 \( o_i \) is the name of an object, \( C \mid o_i \) is the projection of \( C \) on \( o_i \) defined by:

\[ (C \mid C')_i = C \mid C' \quad (C \# C')_i = C \# C' \]
\[ (C < C')_i = C \triangleleft C' \quad (C > C')_i = C \triangleright C' \]
\[ (\downarrow x)_i = \downarrow x \quad (\downarrow \text{last } x)_i = \text{last } x \quad (o.m)_i = o.m \]

The projection \( C \mid o_i \) of \( C \) on a shared variable \( x \) is defined similarly:

\[ (C < C')_i = C \mid x \quad (C > C')_i = C \triangleright x \]
\[ (\downarrow x)_i = \downarrow x \quad (\downarrow \text{last } x)_i = \text{last } x \quad (o.m)_i = o.m \]

We also define the complement \( C \mid o_i \) of \( C \) as the resulting constraint where every access to \( o \) has been eliminated.

\[ (C \mid C')_i = C \mid C' \quad (C \# C')_i = C \# C' \]
\[ (C < C')_i = C \mid C' \quad (C > C')_i = C \triangleright C' \]
\[ (\downarrow x)_i = \downarrow x \quad (\downarrow \text{last } x)_i = \text{last } x \quad (o.m)_i = o.m \]

This is easily computed in one shot with \( C \triangleright o_i \), while \( C \triangleright o_i \) 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 \( m_1 \ldots m_n \), we compute the constraint \( C \mid m \) for each method definition of \( C \) by following syntactically the code of each method. The construction of \( C \mid m \) is built during typing. The resulting constraint for the object is obtained by substituting \( m_i \) by \( C \mid m \). In that case:

\[ C = C \mid m_1 / m_2 / \ldots / C \mid m_n \]

Soundness. \( C \) is sound, written \( \text{Sound}(C) \), if for every shared memory \( x \), \( C \mid x \) does not include \((\downarrow x \mid \downarrow x)\) nor \( (\downarrow x \mid \uparrow x)\) nor \( (\downarrow x \mid \downarrow \text{last } x)\). This also prevents \((\downarrow x \mid \downarrow x)\) and \((\downarrow x \mid \uparrow x)\).

So the soundness ensures that there is no race condition among the reads and writes. Moreover, the constraint \( \downarrow x \mid \downarrow \text{last } x \) ensures that the two values \( \text{last } x \) 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 \( \text{Correct}(C) \), if for every object \( o_i \), \( C \mid o_i \) is included in \( P_i \).

Correctness is a property of the calling context of an object. An 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:

\begin{verbatim}
let simple_colored_point x0 y0 c = object
  new p = point x0 y0
  last color = c
end
\end{verbatim}
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 < cartesian < polar)
  || (translate < (polar # {}))
end

val simple_colored_point : float -> float -> 'a ->
< polar: unit => float * float;
lastcolor: unit => 'a;
rotate: (lastcolor # {}) < (polar # {}))
end

We compute the effect of each method:

C_translate = p.translate < p.cartesian
C_color = | color
C_lastcolor = | last color
C_polar = | p.polar

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

C = (|| last color # ) < (| color # ) || (p.translate < p.cartesian) < (| p.polar # )
and the projections:

C_color = (|| last color # ) < (| color # )
C_polar = translate < cartesian < (| polar # )

C is sound and correct, since the memory color is correctly used
as is the object p. Indeed C_polar C P: translate < cartesian
and translate < cartesian < polar are legal uses of the point
object p which has the policy

(translate # rotate) < ((cartesian # ) || (polar # ))

A wrong example

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

This node is encoded with an object with only one method step #1. The constraint for the body of wrong_use is:

C_step = o.up || ( # o.down)

Thus:

C_step | o = up || ( # 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 u_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

\[ C_{up, o}^\text{up} = \top \text{last } o < | o \quad C_{down, o}^\text{down} = \top \text{last } o < | o \]
which gives

\[ C_{o} = \top \text{last } o < | o < \top \text{last } o < | o \]

having among its errors the infringement \( | o < \top \text{last } o \).

3. Synchronous Semantics

The synchronous reaction semantics follows the existing formulation for the logical semantics of Estend [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, 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_i, w_2 are 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:

\[ R ::= \emptyset | R + [v, x] | R + [v/\text{last } x] \]
\[ O ::= \emptyset | [w_1, O_1, ..., w_m, O_m] \]
\[ v ::= i | tt | ff | (v, v) | \lambda x.e \]
\[ w ::= [m_1(v_1) = v_1, ..., m_n(v_n) = v_n] \]

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 \cap 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 \cup O_2:

\[ O_1 \cup O_2 = O_1, O_2 \quad \text{if } Dom(O_1) \cap Dom(O_2) = \emptyset \]
\[ O_1 + [w_1/O_1] \cup O_2 + [w_2/O_2] = (O_1 \cup O_2) + ([w_1, w_2]/O) \]

We define the next value of a state variable x:

\[ \text{Next}_R(v)(x) = R(x) \text{ if } x \in Dom(R) \]
\[ = v \text{ otherwise} \]

The semantics is defined by two predicates:

\[ R, O \vdash e_1 \Rightarrow e_2 \quad R, O \vdash d_1 \Rightarrow d_2 \]

The first states that when the environment R and O, e_1 emits the value v and rewrites to e_2. The second one states that when 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 \( \lambda x.e_2 \), (e_1 e_2) produces a value v and (e) produces v' under the environment R' + [v, x].
pair's members are reduced in parallel.

The definition of a method makes fields, local object declarations and modes define the access to a state variable (modes-react) and (modes).

R, O

following predicates:

O

Their definition is given in figure 2.

reacts by producing corresponding to those used in

R, O, v

provided \( v \) is evaluated.

R, O || O

\( e \) produces \( v \) with \( |v/\rho| = m(e) \).

(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).

We make several name spaces in the environment. The type entry \( c \) is itself computed by the reaction of modes. The reaction of \( \alpha \) 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 \( (\alpha) \) is a type quantified over type variables \( (\alpha_1, ..., \alpha_n) \) and row variables \( (\rho_1, ..., \rho_m) \). A type can be a function type \( (t_1 \to t_2) \), 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 denuerable 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 \( P \).

\[
\begin{align*}
\sigma &::= \forall \alpha_1, ..., \alpha_n \forall \rho_1, ..., \rho_m. t \\
\tau &::= t \to t | t \times t | c(t_1,...,t_n) | \{r\} \\
r &::= \emptyset | m : t | r \text{ with } P | \rho
\end{align*}
\]

A typing environment \( H \) is defined in the following way:

\[
\begin{align*}
H &::= \emptyset | H + \text{self : } t \\
H + \text{last } x &:: t \\
H + \text{new } o &:: t \\
H + x &:: \sigma
\end{align*}
\]

We make several name spaces in the environment. The type entry \( \text{self : } t \) defines the current object to have the type \( t \).
We define $H, C$ in our kernel language, functions cannot access shared state. The resulting constraint is the parallel composition.

For an equation $t$ and (new-in) deal with method call and object instantiations, they are used to type state variable definitions. A variable stored with type scheme $\{a_1, ..., a_n\}$ of $\sigma$ whereas $FR(\sigma)$ stands for the free row type variables $\{\rho_1, ..., \rho_m\}$ of $\sigma$. These functions are lifted to typed-environments.

Instantiation: $t\vdash r. e \vdash r. p \leq \forall e. p$. Generalization: $\text{gen}_H(t) = \forall \alpha_1, ..., \alpha_n. \forall \rho_1, ..., \rho_m. t$.

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 type environment $H$ and scheduling constraints $C$.

The definitions of these predicates are given in figure 3.

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

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

(imm) The fourth rule is for immediate constants.

(fun) In our kernel language, functions cannot access shared state variables or 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 if then $d_1$ else $d_2$ is typed classically. From the scheduling constraint point-of-view, the constraints of the two branches are exclusive ($C_1 \neq C_2$) but $C$ must be scheduled before one of the two.

(and) and (in) deal with parallel ($D_1$ and $D_2$) and sequential composition ($D_1$ 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-m) For an equation $x = e$ where $x$ is a state variable and $C$ is the scheduling constraint for $e$, we generate the constraint $C' \leq \{x\}$ 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 $t$.

(m-call) and (new-in) deal with method call and object instantiations. 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 $P = C_{\|}$, 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 $\{r \ with \ P\}$. $C_{\|}$ is the constraint of $o$ obtained by erasing accesses to object $o$.

$\text{new-in}$ 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 (last $x = e$), local object instantiations (new $o = e$) and the definition of methods (when $m(p)$ returns $q$) $b$.

### Figure 2. Synchronous Reaction Rules for Objects
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: \( \{ m : t_1, \ldots, m_n : t_n \} \) with \( \{ p : t, r \} \) or \( \{ m : t_1, \ldots, m_n : t_n \} \) with \( \{ p : t, r \} \). 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

\[
\text{new o1 = f (x) in} \\
\text{new o2 = f (x*1) in} \\
\text{if (x = 0) then do r = o1.m(x) + o2.m(x) done} \\
\text{else do r = o1.n(x) + o1.m(x) done} \\
\text{val g :} \\
\text{(int -> m: int) -> n: int -> m: int ...} \\
\text{with {} # m # (m || n) > -> int -> int ...}
\]

Suppose that \( H = [ f : \alpha_1, x : \alpha_2 ] \) where \( \alpha_1, \alpha_2 \) are fresh type variables. After having typed the declarations of \( \alpha_1 \) and \( \alpha_2 \), we have \( H' = H + [ \alpha_1 : (p), \alpha_2 : (p)] \). Then, the typing of the first branches states that:

\[
H', (\alpha_1, m | \alpha_2, m) \vdash \alpha_1, m(x) + \alpha_2, m(x) : \text{int}
\]

and

\[
H', (\alpha_1, n | \alpha_2, m) \vdash \alpha_1, n(x) + \alpha_2, m(x) : \text{int}
\]

with \( \alpha_1 = \text{int} \rightarrow \{ p \} \). We obtain the final constraint:

\[
C = (\alpha_1, m || \alpha_2, m) \neq (\alpha_1, n || \alpha_1, m)
\]

Projecting it on \( \alpha_1 \), we get:

\[
C|_{\alpha_1} = (\ || m) \neq (\ || ) = m \\
\]

Thus,

\[
\rho = m : \text{int} \rightarrow \text{int}, n : \text{int} \rightarrow \text{int} \text{ with m } \neq , \rho'
\]

Then, we compute the resulting constraint \( C' \) by eliminating \( \alpha_2 \):

\[
C' = \alpha_1, m \neq (\alpha_1, n || \alpha_1, m)
\]

\[
C|_{\alpha_1} = m \neq (n || m)
\]

Thus, \( \rho' = n || m, \rho' \). Finally, we get the type:

\[
g : \forall p, (\text{int} \rightarrow \{ m : \text{int} \rightarrow \text{int}, n : \text{int} \rightarrow \text{int} \text{ with } \neq m \neq (n || m), \rho \} \rightarrow \text{int} \rightarrow \text{int}
\]

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 \( t_1 \rightarrow t_2 \). The language also provides richer control-structures (e.g., hierarchical automata as proposed in [7]), Lustre.
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:

$$\text{int} \rightarrow \exists \text{o} : \text{int} \times (\text{int} \rightarrow \text{int}) \times (\text{int} \rightarrow \text{int})$$

stating that it returns a pair of functions with respectively effects $(\lnot \text{last} < \lnot \text{o})$ and $(\lnot \text{last} < \lnot \text{o})$ where $\text{o}$ is quantified existentially. 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., $\text{up}$ and $\text{down}$). A natural extension is to consider a class mechanism and inheritance as a way to build systems incrementally. The team $\text{up}$ and $\text{down}$ define their own class:

```ocaml
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 $\text{up}$ is $C^{\up} = \lnot \text{last} x < \lnot x$). Then, the integration of the two to form the class updown becomes:

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

One should write instead:

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

operators ($\text{pre}$ and $\rightarrow$) 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 $C_1 \land \ldots \land C_n$.

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 ones eliminate signals. The next one completes partial definitions. Completion consists in transforming an equation:

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

into:

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

The next one eliminates Lustre primitives ($\text{pre}$ and $\rightarrow$) 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 simpler 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 instanciated objects are not first-class values as this is typically the case in object-oriented languages. For example, the following program is statically rejected:

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

One should write instead:
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 explicit by the inheritance mechanism and to write:

```csharp
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 account 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 useful 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 \( P_1 \rightarrow P_2 \) 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 one 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 inputs/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 dataflow languages such as Lustre and his functional variant Lucid Synchronize. The main novelty is to provide a new programming construct 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 notion to express scheduling policies. A scheduling policy is described by a simple regular expression giving the possible combination of modes 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


