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

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. Conversely, the control activation of each law shall be described by a hierarchical automata a la StateCharts. 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. 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 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution.
Copyright © 2009 ACM 978-1-60558-356-3/09/06... $5.00

The ford gear-shift study (amp.ece.cmu.edu/ece-seminar/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:

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

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:

```ocaml
let twomodes x0 =
  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
    end
  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:
\[
P ::= P | P | P \# P | P < P | m | \epsilon
\]
\(P_1 \# P_2\) 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 \(\epsilon\) stands for the empty
schedule or the schedule with no calls.

In this paper we show that checking the correct use of ob-
jects amounts at checking policy inclusion between declared pol-
cies 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 vari-

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 } ->
'c -> '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 \(\ldots\) states that \(h\) is a function returning an object which
contains, 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 syn-
chronous 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 \((\text{var} x \in b)\) and instantiated objects
\((\text{new} x = e \in b)\) or a collection of equations \((D)\). A collection
can be an equation \((p = e)\), a parallel \((D_1 \& D_2)\) or sequential
composition \((D_1 \otimes D_2)\). Finally, it can be based on the disjunction
of two equations \((\text{if} e \text{ then } D_1 \text{ else } D_2)\) according to a boolean
condition \((e)\). An expression \((e)\) may be a constant \((v)\), a variable
\((x)\), a read access to a shared variable \((\text{last} x)\), a pair \((e_1, e_2)\),
a function application \((e_1 \ e_2)\), a function \((\text{lambda} (x.e))\), a method call
\((\text{o}(m(e)))\) of an object \(o\) with name \(m\) and, finally, the definition
of a system with modes \((\text{object fields} \ objs \ \text{modes with} \ P \ \text{end})\).
We call it an un-instantiated object (or simply object). An object has
instance variables \((\text{fields})\), instantiated objects \((\text{objs})\), a sequence of
modes \((\text{modes})\) and a scheduling policy \((P)\) specifying the way
modes can be called.

\[
\begin{align*}
d & ::= \text{let} x = e; d; d \\
p & ::= p, p | x \\
b & ::= \text{var} x \in b | \text{new} x = e \in b | D \\
e & ::= v | x | \text{last} x | \{e, e\} | \text{let} x = e | \text{lambda} (x.e) | \text{obj}(m(e)) \\
D & ::= p = e | D \& D | D \otimes D' | \text{if} e \text{ then } D \text{ else } D' \\
\text{fields} & ::= \text{last} x = e; \ldots; \text{last} x = e
\end{align*}
\]

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, \text{pre} and initialization operator \(\rightarrow\) of Lustre.

\textbf{Lustre nodes} A Lustre node is a statefull 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 follow-
ing definition of an object:

let node\_\_f =
\begin{align*}
\text{object} & \quad \text{when} \ \text{step}(p) \ \text{returns} \ (q) \ b \\
& \quad \text{with} \ \text{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 \textbf{Lustre delay \text{pre}} The initialized delay \text{pre}(y) is such that:

<table>
<thead>
<tr>
<th>(y)</th>
<th>(y_0)</th>
<th>(y_1)</th>
<th>(y_2)</th>
<th>(y_3)</th>
<th>(y_4)</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{pre}(y)</td>
<td>(x)</td>
<td>(y_0)</td>
<td>(y_1)</td>
<td>(y_2)</td>
<td>(y_3)</td>
<td>(y_4)</td>
</tr>
</tbody>
</table>

To replace this operator, we define the following function:

let \text{node\_\_x} =
\begin{align*}
\text{object} & \quad \text{last mem} = x \\
& \quad \text{when} \ \text{get} \ (x) \ \text{returns} \ (lm) \ \text{where} \ \text{do} \ \text{lm} = \text{last mem} \ \text{done} \\
& \quad \text{when} \ \text{set} \ (x) \ \text{returns} \ () \ \text{where} \ \text{do} \ \text{mem} = \text{v} \ \text{done} \\
\& \quad \text{with} \ \text{get} \ < \ \text{set} \ \{\} \\
\end{align*}

Val node\_\_p =
\begin{align*}
'a -> < \text{set}: 'a -> \text{unit}; \text{get}: \text{unit} -> 'a \\
\& \quad \text{with} \ \text{get} \ < \ \text{set} \ \} >
\end{align*}

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

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

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

\[
\begin{array}{cccccccc}
e_1 & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
e_2 & y_0 & y_1 & y_2 & y_3 & y_4 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccc}
e_1 \rightarrow e_2 & x_0 & y_1 & y_2 & y_3 & y_4 & \ldots \\
\end{array}
\]

We define the following function:

let \text{init} =
\begin{align*}
\text{object} & \quad \text{last init} = \ \text{true} \\
& \quad \text{when} \ \text{step}(x,y) \ \text{returns} \ (z) \ \text{where} \ \text{do} \ \text{z} = \ \text{if} \ \text{last init} \ \text{then} \ \text{else} \ \text{y} \\
& \quad \text{and} \ \text{init} = \ \text{false} \\
& \quad \text{done} \\
& \quad \text{with} \ \text{step} \ \{\} \\
\end{align*}
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 \( \epsilon \) 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 na"ive way of doing these computations is to represent \( P \) in a disjunctive normal form with (\#) as the disjunction operator:

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

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

\[
\begin{align*}
S < (S'_1 \# S'_2) &= (S < S'_1) \# (S < S'_2) & (1) \\
(S_1 \# S_2) < S' &= (S_1 < S') \# (S_2 < S') & (2) \\
(m < S) || (m' < S') &= \# \left( (S_1 < S') \# (m < S) \right) & (3) \\
\# S_i \mid (\# S'_j) &= \# (S_i \mid S'_j) & (4)
\end{align*}
\]

Policy language

\[
P ::= P \mid P \mid P \# P \mid P < P \mid m \mid \epsilon
\]

The disjunction operator \( P_1 \# P_2 \) states either \( P_1 \) or \( P_2 \) states. 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 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 \((||)\) and (\#) are. Moreover, (\#) and (\#) 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 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):

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

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

\[
\begin{align*}
(C || C')|_o &= C|_o || C'|_o & (C \# C')|_o &= C|_o \# C'|_o \\
(C < C')|_o &= C|_o < C'|_o & \epsilon|_o &= \epsilon \\
(\uparrow x)|_o &= \epsilon & (\downarrow x)|_o &= \epsilon \\
(\uparrow \text{last} x)|_o &= \epsilon & (\downarrow x)|_o &= \epsilon \quad o' \neq o \\
(o.m)|_o &= m
\end{align*}
\]

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

\[
\begin{align*}
(C || C')|_x &= C|_x || C'|_x & (C \# C')|_x &= C|_x \# C'|_x \\
(C < C')|_x &= C|_x < C'|_x & \epsilon|_x &= \epsilon \\
(\uparrow x)|_x &= \epsilon & (\downarrow x)|_x &= \epsilon \quad x' \neq x \\
(\uparrow \text{last} x)|_x &= \uparrow \text{last} x & (\downarrow x)|_x &= \epsilon \quad x' \neq x \\
(o.m)|_x &= \epsilon
\end{align*}
\]

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

\[
\begin{align*}
(C || C')|_o &= C|_o || C'|_o & (C \# C')|_o &= C|_o \# C'|_o \\
(C < C')|_o &= C|_o < C'|_o & \epsilon|_o &= \epsilon \\
(\uparrow x)|_o &= \epsilon & (\downarrow x)|_o &= \epsilon \\
(\uparrow \text{last} x)|_o &= \uparrow \text{last} x & (\downarrow x)|_o &= \epsilon \quad o' \neq o \\
(o.m)|_o &= \epsilon
\end{align*}
\]

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 \( m_1 \ldots m_n \), we compute the constraint \( \text{C}^{m_n} \) for each method definition of \( c \) by following syntactically the code of each method. The construction of \( \text{C}^{m_n} \) is built during typing. The resulting constraint for the object is obtained by substituting \( m_i \) by \( \text{C}^{m_i} \) in \( P \), that is:

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

**Soundness** \( C \) is sound, written \( \text{Sound}(C) \), if for every shared memory \( x, C|_o \) does not include \( (\uparrow x < \downarrow x) \) nor \( (\downarrow x < \uparrow \text{last} x) \). This also prevents \( (\uparrow x || \downarrow x) \) and \( (\downarrow x || \uparrow \text{last} x) \) from \( o \). So the soundness ensures that there is no race condition among the writes and reads. Moreover, the constraint \( \downarrow x < \uparrow \text{last} x \) 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 \( o_i, C|_{o_i} \) 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:

```plaintext
let simple_colored_point x0 y0 c =
    object
        new p = point x0 y0
        last color = c
```

when translate (nx,ny) returns (nx, 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 color() returns (color)
with {{lastcolor # ()} < (color # ())}
|| (translate < (polar # ()))
end

val simple_colored_point : float -> float -> 'a ->
{color: 'a => unit; translate < cartesian < polar
when polar() returns (p.polar())
when translate (nx,ny) returns (n, x, y) where
|| (translate < (polar # ()))
end

We compute the effect of each method:
\[
C^\text{translate} = p.\text{translate} < p.\text{cartesian}
\]
\[
C^\text{color} = |\text{color}|
\]
\[
C^\text{lastcolor} = |\text{last color}|
\]
\[
C^\text{polar} = |\text{polar}|
\]

We compute C by applying the substitutions to P and get:
\[
C = (|\text{last color} # ()| < (|\text{color} # ()|)
|| (p.\text{translate} < p.\text{cartesian}) < (|\text{polar} # ()|))
\]

and the projections:
\[
C|_{\text{color}} = (|\text{last color} # ()| < (|\text{color} # ()|)
\]
\[
C|_{\text{polar}} = \text{translate} < \text{cartesian} < (|\text{polar} # ()|)
\]

C is sound and correct, since the memory color is correctly used
as is the object p. Indeed \(C|_y \subseteq P_x: \text{translate} < \text{cartesian} < \text{polar}\) are legal uses of the point
object p which has the policy
\[
(\text{translate} \neq \text{rotate}) < ((\text{cartesian} # ()) || (\text{polar} # ()))
\]

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 \(\# \epsilon\). The constraint for the wrong use is:
\[
C^\text{step} = o.\text{up} || (\epsilon \neq o.\text{down})
\]
Thus:
\[
C^\text{step}|_o = \text{up} || (\epsilon \neq \text{down})
\]

which is not included in the policy associated to o (\(\text{up} \neq \text{down})

. The compiler rejects this code with the following error message:

Type error: the object o is used with policy
done // (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
\[
C^\text{up}|_o = |\text{last o} < \text{o}|
C^\text{down}|_o = |\text{last o} < \text{o}|
\]

which gives
\[
C|_o = |\text{last o} < \text{o} < \text{last o} < \text{o}|
\]

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

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(a) is the set of called
methods of the object a. w is well formed meaning that names of
methods are pairwise distinct. w1, w2 is the concatenation of the
two provided names do not interect. An entry m(v)=v’ states that
the method m returns v’ on input v.

\[
\begin{align*}
R & ::= \emptyset | R + [v/x] | R + [v/\text{last x}] \\
O & ::= \emptyset | [w_1/o_1,\ldots,w_m/o_m] \\
v & ::= \text{if } t \text{ tt } | \text{ff } (v,v) | \lambda x.e \\
\text{object fields } o & = o_1,\ldots,o_n \\
w & ::= [m_1(v_1)=v_1',\ldots,m_n(v_n)=v_n']
\end{align*}
\]

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 R1 and R2 are two reaction environments,
R1 R2 is the union of the two provided domain do not intersect. If O1 and O2 are two environments, we define O1 || O2:
\[
O_1 || O_2 = O_1, O_2, \text{ if Dom(O_1) \cap Dom(O_2) = \emptyset}
O_1 + [w_1/o] || O_2 + [w_2/o] = (O_1 || O_2) + [(w_1,w_2)/o]
\]

We define the next value of a state variable x:
\[
\text{Next}_R(v)(x) = R(x) \text{ if } x \in \text{Dom}(R)
= v \text{ otherwise}
\]

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 under the environment R and O, \(e_1\) emits the
value v and rewrites to e2. The second one states that under R and
O, the declaration d1 produces the reaction environment R’ and
rewrites to d2. 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). (e1 e2) produces a value v’ and rewrites to (e’1 e’2) when (e1) produces
a function Ax.e, (e2) produces a value v and (e) produces v’ under
the environment R’ + [v/x].
The definition of a method defines the value of an object. For an object definition, their definition is given in figure 2.

\[(\text{pair})\]  
\[R, O \vdash e \equiv e' \quad R, O \vdash d_1 \rightarrow d'_1 \]  
\[R, O \parallel O_1 \vdash e_1 \equiv e'_1 \quad R, O_2 \vdash e_2 \equiv e'_2 \]

\[(\text{if-t})\]  
\[R, O \parallel O_1 \vdash e \rightarrow e' \quad R, O_1 \parallel d_1 \rightarrow d'_1 \]  
\[R, O \parallel O_2 \parallel e' \quad R, O_2 \parallel d_2 \rightarrow d'_2 \]

\[(\text{if-f})\]  
\[R, O \parallel O_1 \vdash e \rightarrow e' \quad R, O_1 \parallel d_1 \rightarrow d'_1 \]  
\[R, O \parallel O_2 \parallel e' \quad R, O_2 \parallel d_2 \rightarrow d'_2 \]

\[(\text{var})\]  
\[R, O \vdash d \rightarrow [v/x] d' \]  
\[R, O \parallel O_1 \vdash \text{var} x \in d \rightarrow \text{var} x \in d' \]

\[(\text{new})\]  
\[R, O \parallel O_1 \vdash \text{new} o = e \in d \rightarrow \text{new} o = v' \in d' \]

\[(\text{modes})\]  
\[R, O \parallel O_1 \vdash \text{fields} \rightarrow \text{fields'} \]  
\[R, O \parallel O_1 \vdash \text{objs} \rightarrow \text{objs'} \]  
\[R, O \parallel O_1 \vdash \text{modes} \rightarrow \text{modes'} \]

Their definition is given in figure 2.

\[(\text{modes-react})\] makes fields, local object declarations and modes react in parallel. Fields are evaluated in the closure environment \(R_c\), 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.

\[(\text{last-def})\] The next value for \(x\) is \(v' \mid x\) has been computed during the reaction or it keeps its previous value \(v\).

\[(\text{new-def})\] is similar to the rule \((\text{new})\).

\[(\text{when-def})\] 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 \((\sigma)\) 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 enumerable 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 \(P\).

\[\sigma ::= \forall \alpha_1, ..., \alpha_n. \forall \rho_1, ..., \rho_m.t \]  
\[t ::= t \rightarrow t \mid t \times t \mid \alpha \mid c(t, ..., t) \mid \{r\} \]  
\[r ::= m : t \mid r \mid w \mid r' \mid \rho \]

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

\[H ::= \emptyset \mid H + \text{self} : t \mid H + \text{last} x : t \mid H + \text{new} o : t \mid H + x : \sigma \]

We make several name spaces in the environment. The type entry \(\text{self} : t\) defines the current object to have the type \(t\); \(\text{last} x : t\)
We define $H, C$ as the typing rule for an object expression. We deal with parallel ($\&$) and sequential composition ($\&$). The definitions of these predicates are given in figure 3.

Figure 2. Synchronous Reaction Rules for Objects

defines the shared memory $\text{last } x$ with type $t$; $\text{new } o : t$ defines the object $o$ with its type $t$; $x : \sigma$ defines a name $x$ to have the type scheme $\sigma$. $H_1 + H_2$ 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 $t$ and a type scheme $\sigma$. $t \leq \sigma$ holds when there is a substitution of the quantified variables of $\sigma$ 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 $H$. $\mathit{FV}(\sigma)$ stands for the free type variables $\{\alpha_1, \ldots, \alpha_n\}$ of $\sigma$ whereas $\mathit{FR}(\sigma)$ stands for the free row type variables $\{\rho_1, \ldots, \rho_m\}$ of $\sigma$. These functions are lifted to type-environments.

Instantiation: $t[t'/\alpha][\rho]/[\rho'] \leq \forall \alpha, \rho.t$

Generalization: $\mathit{gen}_H(t) = \forall \alpha_1, \ldots, \alpha_n \forall \rho_1, \ldots, \rho_m.t$

where $\{\alpha_1, \ldots, \alpha_n\} = \mathit{FV}(t) - \mathit{FV}(H)$

and $\{\rho_1, \ldots, \rho_m\} = \mathit{FR}(t) - \mathit{FR}(H)$

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 $\sigma$ in the environment can be used with an instantiated type $t$. No scheduling constraint is necessary so we write $\varepsilon$ in the constraint.

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

(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 $\varepsilon$.

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

(if) The control structure (if $e$ then $d_1$ else $d_2$) is typed classically. From the scheduling 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 ($D_1 \& D_2$). 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 $C < x$ 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 (var $x$ 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 $P = C|e$, 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: $\{r \text{ with } P\}; C_o$ is the constraint $C$ obtained by erasing accesses to object $o$.

$P$ 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 ($\text{last } x = e$), local object instantiations ($\text{new } o = e$) and the definition of methods (when $m(p)$ 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:

\[
\begin{align*}
  t & ::= \{r\} \\
  r & ::= \emptyset \mid m : t, r \mid r \text{ with } P \mid \rho
\end{align*}
\]

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

\[
\begin{align*}
  (r \text{ with } P_1) \text{ with } P_2 & = (r \text{ with } P_2) \text{ with } P_1 \\
  (r \text{ with } P_1) \text{ with } P_2 & = r \text{ with } P_1 \# P_2 \\
  m_1 : t_1, m_2 : t_2, r & = m_2 : t_2, m_1 : t_1, r
\end{align*}
\]
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: \( \{ m : t_1 \to t_2, \ldots, m : t_n \} \) or \( \{ m_1 : t_1, \ldots, m_n : t_n \} \). The unification algorithm of Rémy & Vuillon is modified accordingly.

We illustrate its use on the following example:

\[
\begin{align*}
&\text{let node } g(f)(x) \text{ returns } (r) \text{ where} \\
&\text{new o1 = f (x) in} \\
&\text{new o2 = f (x+1) in} \\
&\text{if } (x = 0) \text{ then do } r = o1.m(x) + o2.m(x) \text{ done} \\
&\text{else do } r = o1.n(x) + o1.m(x) \text{ done} \\
&\text{val g : } \langle \text{int} \to < m : \text{int} \to \text{int}; n : \text{int} \to \text{int} \rangle \\
&\text{with } \{ \# m \# (m \parallel n) \} \\
&\text{Suppose that } H = [ f : \alpha_1, x : \alpha_2 ] \text{ where } \alpha_1, \alpha_2 \text{ are fresh type variables. After having typed the declarations of } \alpha_1 \text{ and } \alpha_2, \text{ we have: } H' = H + [ \{ \rho \}, \alpha_2 : \{ \rho \} ] \text {. Then, the typing of the first branches states that:} \\
&H',(\alpha_1,m \parallel o_2,m) \vdash o_1.m(x) + o_2.m(x) : \text{int} \\
&\text{and } H',(\alpha_1,n \parallel o_1,m) \vdash o_1.n(x) + o_1.m(x) : \text{int} \\
&\text{with } \alpha_1 = \text{int} \to \{ \rho \}. \text{ We obtain the final constraint:} \\
&C = (\alpha_1,m \parallel o_2,m) \not\# (\alpha_1,n \parallel o_1,m)
\end{align*}
\]

Projecting it on \( o_2 \), we get:

\[
C|_{o_2} = (\epsilon \parallel m) \not\# (\epsilon \parallel \epsilon) = m \not\# \epsilon
\]

Thus,

\[
\rho = m : \text{int} \to \text{int}; n : \text{int} \to \text{int} \text{ with } m \not\# \epsilon, \rho'
\]

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

\[
C' = o_1.m \not\# (o_1.n \parallel o_1.m) \\
C'|_{o_1} = m \not\# (n \parallel m)
\]

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

\[
g : \forall \rho.((\text{int} \to [ m : \text{int} \to \text{int}; n : \text{int} \to \text{int} \rangle\text{ with } \epsilon \not\# m \not\# (n \parallel m), \rho) \to \text{int} \to \text{int})
\]

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 \( t_1 \to t_2 \). The language also provides richer control-structures (e.g., hierarchical automata as proposed in [7]).
operators (prim 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
\[ C_1 \# \ldots \# C_n. \]

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

\[
\textbf{let node } f(x) \textbf{ returns } (y) \textbf{ where } \begin{aligned}
\text{new } o &= x \text{ in } \\
\text{do } y &= o.\text{m1}(1) + o.\text{m2}(2) \text{ done}
\end{aligned}
\]

\[
\textbf{val } f : \text{int} \rightarrow \text{int} \, m1 : \text{int} \rightarrow \text{int} \, m2 : \text{int} \rightarrow \text{int}
\]

\[
\text{with } m1 \parallel m2 \rightarrow \text{int}
\]

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:

\[
\begin{aligned}
\textbf{let up(last } x)(y) \textbf{ returns } () \\
\text{do } x &= \text{last } x + y \text{ done}
\end{aligned}
\]

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

\[
\begin{aligned}
\text{let twomodes } x0 &= \\
\text{last } o &= x \text{ in } \\
\text{(fun } y \text{ returns } o \text{ do } o &= \text{last } o + y \text{ done),} \\
\text{(fun } y \text{ returns } o \text{ do } o &= \text{last } o - y \text{ done)}
\end{aligned}
\]

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, \textbf{twomodes}
would receive a type signature of the form:

\[
\text{int} \rightarrow \exists o: \text{int}.(\text{int} \, \text{last } o < \downarrow o) \times (\text{int} \, \text{last } o < \downarrow o)
\]

stating that it returns a pair of functions with respectively effects
(\( \text{last } o < \downarrow o \)) and (\( \text{last } o < \downarrow o \)) where \( o \) is quantified ex-
istentially. 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 func-
tions 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
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 \textbf{up} and \textbf{down} define their own
class:

\[
\begin{aligned}
\text{class up =} \\
\text{last } x &= 0 \\
\text{when up(y) returns (x)} \\
\text{do } x &= \text{last } x + y \text{ done}
\end{aligned}
\]

\[
\begin{aligned}
\text{class down =} \\
\text{last } x &= 0 \\
\text{when down(y) returns (x)} \\
\text{do } x &= \text{last } x - y \text{ done}
\end{aligned}
\]

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

\[
\begin{aligned}
\text{class two =} \\
\text{inherit up} \\
\text{inherit down}
\end{aligned}
\]
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:

```java
class two {
    object
    inherit up, down with up.x=down.x
}
```

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 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 provide a new construction 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


