Towards a Higher-Order Synchronous Data-Flow Language

Jean-Louis Colaço
Esterel Technologies
Toulouse, France
Jean-Louis.Colaco@esterel-technologies.com

Alain Girault
INRIA Rhône Alpes
Montbonnot, France
Alain.Girault@inrialpes.fr

Grégoire Hamon *
Chalmers University
Goeteborg, Sweden
hamon@cs.chalmers.se

Marc Pouzet †
Université Pierre et Marie Curie
Paris, France
Marc.Pouzet@lip6.fr

ABSTRACT

The paper introduces a higher-order synchronous data-flow language in which communication channels may themselves transport programs. This provides a mean to dynamically reconfigure data-flow processes. The language comes as a natural and strict extension of both LUSTRE and LUCID SYNCHRONE. This extension is conservative, in the sense that a first-order restriction of the language can receive the same semantics.

We illustrate the expressivity of the language with some examples, before giving the formal semantics of the underlying calculus. The language is equipped with a polymorphic type system allowing types to be automatically inferred and a clock calculus rejecting programs for which synchronous execution cannot be statically guaranteed. To our knowledge, this is the first higher-order synchronous data-flow language where stream functions are first class citizens.

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; F.3.2 [**Semantics of programming languages**]: Operational semantics.

General Terms

Languages, theory.

Keywords

Synchronous data-flow programming language, stream functions, Kahn processes, functional programming, dynamic reconfiguration, type systems.

*This work was partially financed by the Swedish Foundation for Strategic Research.

†This work was partially financed by the French ACI Sécurité Informatique Alidecs.

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.
EMSOFT'04, September 27–29, 2004, Pisa, Italy.
Copyright 2004 ACM 1-58113-860-1/04/0009 ...$5.00.

1. INTRODUCTION

Synchronous data-flow programming languages, such as LUSTRE [17] or SIGNAL [16], are domain specific languages dedicated to the implementation of critical real-time embedded software. They are based on the *synchronous* model of concurrency and time, providing high-level constructs while insuring strong guarantees such as the execution in both bounded time and memory. They are associated to efficient compilation methods into sequential code, verification tools, and automatic distribution mechanisms allowing programs to be executed in a distributed asynchronous context [2]. Introduced in the eighties, these languages are used now in various industrial domains such as nuclear power plants, avionics, and public transportation.

More recently, LUCID SYNCHRONE [25, 11][1] was introduced to demonstrate that LUSTRE could be extended with powerful features such as automatic type and clock inference, as well as some form of higher-order, while retaining the fundamental properties of LUSTRE. The language has been in use for several years now and serves as a laboratory for prototyping language extensions and program analysis [14, 13]. It is used by the SCADE team at ESTEREL TECHNOLOGIES for experimenting extensions of SCADE and the design of a new compiler [12].

Classical synchronous data-flow languages such as LUSTRE and SIGNAL are *first-order* languages in essence: a synchronous stream function corresponds to a finite state machine and there is a clear separation between a value — something which can be transported on a channel — and a program. *Higher order* comes naturally in this setting as a way to parameterize a function by another function, as is done in LUCID SYNCHRONE. Yet, LUCID SYNCHRONE provides only a *restricted* form of higher-order since the code taken as a parameter or returned as a result is determined statically and cannot evolve dynamically. Moreover, such *static* higher order can be eliminated by the compiler by means of inlining, producing a first-order program[2]. Thus, whereas static higher-order increases modularity, the program is entirely determined statically, and values which may be transmitted on channels are still non-functional values.

[1] `http://www-spi.lip6.fr/lucid-synchrone`

[2] Nonetheless, this is certainly not the right way to do. Separate compilation is lost and code size may grow dramatically.

Nonetheless, even in the area of real-time embedded software, there is a need to dynamically reconfigure a system, that is, to change the code to be executed during its execution or to dynamically load some code through a channel. Examples can be found in robot missions, networks switches, software-defined radio, or in contexts where quality of services must be adapted during the execution. In a dynamic reconfiguration, the new program to be run is not known statically when compiling the original program (though some constraints should be imposed on it).

Dynamic reconfiguration can be related to general higher order (i.e., a program becomes a first-class citizen). Whereas Dynamic reconfiguration has been extensively studied in the context of asynchronous process calculi [22, 26], existing synchronous data-flow languages do not provide a mean for describing or analyzing such dynamic features. This is the purpose of the present paper.

We propose a *conservative* higher-order extension of an existing synchronous data-flow language such as Lustre or Lucid Synchrone. By conservative, we mean that its first-order restriction corresponds to Lustre and shares its Kahn [19] semantics. Such an extension is obtained by defining a synchronous operational semantics as well as type and clock conditions rejecting programs for which synchronous execution cannot be statically guaranteed.

First, Section 2 gives an overview of the proposed language through examples. Then, the underlying calculus and its behavioral semantics are formally defined in Section 3. Section 4 defines static conditions rejecting programs which cannot be executed synchronously. They are given by means of type conditions and clock conditions. Section 5 compares the proposed calculus to existing ones, and finally, Section 6 gives future research directions and concludes.

2. MOTIVATIONS AND OVERVIEW

Our motivations are twofold. First, we advocate that higher-order provides more modular ways to specify programs mixing data-flow and control. Such programs are classically found in automatic control software, as is the case of the Airbus flight control software. Second, we advocate that higher-order is an essential feature in the domain of embedded systems like software-defined radio. We illustrate with several examples the features of the language and its expressive power. In these examples, we use clocks, a standard practice in data-flow programming languages. Indeed, such languages use clocks as powerful control structures to manipulate data, and clocks are a form of temporal types. Basically, it suffices to know that the clock of a stream defines the sequence of logical instants where it bears a value.

2.1 The need for better modularity

In synchronous data-flow languages, functions are either *sequential* or *combinatorial*: in the first case their results at a given instant depends on the history of their inputs, while in the second case they do not[3]. For example, the function computing the average sequence of two input sequences is a combinatorial function, written as:

```
let average (x, y) = (x + y) / 2
```

[3]The term *sequential* comes from the electronic circuit community. It designates circuits with latches, as opposed to *combinatorial* circuits.

Its type signature is:

```
node average : int * int -> int
```

which states that `average` is a combinatorial function from a pair of integer streams to an integer stream.

A typical example of a sequential function is the rectangle integration function. It is written as follows[4]:

```
let node rectangle (init, dt, dx) = x where
  rec x = init -> pre x +. dx *. dt
```

This time, the output `x` is defined by a recursive equation. `+.` and `*.` stand for the floating point addition and multiplication. The initialization operator `->` and the delay `pre` are the ones of Lustre and Lucid Synchrone.

Here, the `rectangle` function takes three streams, `init`, `dt`, and `dx`, and returns a stream `x` so that $x_n = init_1 + \Sigma_{i=2}^{n}\, dx_i * dt_i$. Its type and clock signatures, as synthesized by the type system and the clock calculus, are respectively:

```
node rectangle :  float * float * float => float
node rectangle :: 'a * 'a * 'a -> 'a
```

The rectangle function is a *sequential* function and this is why its definition is annotated with the additional keyword `node`. In the type signature, we use `->` for combinatorial functions and `=>` for sequential functions. The clock signature specifies that the three inputs of `rectangle` must have the same clock, symbolically noted `'a`, which is also the clock of the result. Now, a function can be instantiated:

```
let node double_rect (x0, dx0, dt, d2x) = x where
  rec dx = rectangle (x0, dt, d2x)
  and x = rectangle (dx0, dt, dx)

node double_rect :
            float * float * float * float => float
node double_rect :: 'a * 'a * 'a * 'a -> 'a
```

This double integration is "wired" with the rectangle integrating function and must be rewritten if a better integrating function is preferred. Higher-order can be used here to obtain a more generic code:

```
let node double (node integr) (x0,dx0,dt,d2x) = x
  where rec dx = integr (x0, dt, d2x)
        and x = integr (dx0, dt, dx)

node double : ('a * 'b * 'c => 'c)
                -> ('a * 'a * 'b * 'c) => 'c
node double :: ('a * 'b * 'c -> 'c)
                -> ('a * 'a * 'b * 'c) -> 'c
```

The node `double` is parameterized by a node `integr` (this is why this parameter is preceeded by the keyword `node`) and a tuple `(x0,dx0,dt,d2x)`. Note that the type and clock calculus generate signatures as general as possible. One inferred constraint in the type signature comes from the fact that `dx` is used as the third parameter of `integr`, hence the type signature `'a * 'b * 'c => 'c` for `integr`. Similarly, the type signature of the tuple `(x0,dx0,dt,d2x)` is inferred to be `('a * 'a * 'b * 'c)`. Now, the `double` node can be instantiated as follows:

[4]Except the extra keyword `node` for qualifying sequential functions, programs are written in Lucid Synchrone (version 2.0) syntax. See [25] for a tutorial introduction of the language.

```
let double_rect = double rectangle
```

When the slope of the function to be integrated is too steep, it is preferable to use a trapeze integration:

```
let node trapeze (init, dt, dx) = x where
  rec x = init -> pre x + ((dx + pre dx) / 2) * dt

let double_trapeze = double trapeze
```

While modularity has increased through the use of functions as parameters, we can observe that this higher-order feature is *static*: the first argument of the function `double` is a constant node defined at top-level and can not evolve *dynamically*. What about a more general case where the integrating function in argument is not known at compile time and may evolve during the execution? In other words, what about having streams of (stream) functions? In order to use another integration method, we want this process to be reconfigured dynamically according to some criteria, materialized by the `res` argument:

```
let node server (node intgr, res) (x0, dt, dx) = o
  where
     rec o = (intgr every res) (po, dt, dx)
     and po = x0 -> pre o

node server :
   ('a * 'b * 'c => 'a) * bool
                        -> 'a * 'b * 'c => 'a
node server :: 'a * 'a -> 'a * 'a * 'a -> 'a
```

Think of the `server` function as a process which expects two inputs and returns one output. Its first input is a stream of sequential functions and `(res, dt, dx)` is a structured input of three streams. `(po, dt, dx)` is the current input given to the integrating function. The intuitive behavior of the `server` function is the following: every time the boolean `res` is true, a new reconfiguration occurs by instantiating the current value of `intgr`. Thus, if f is a stream of stream functions and r is a boolean stream, the construction f `every` r stands for a stream function which is reseted every time r is true. The following table is an example of a run of the `server` function:

`res`	f	f	f	t	f	f	f	t	f	f
`intgr`	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}
`dt`	1	1	1	1	1	1	1	1	1	1
`dx`	dx_1	dx_2	dx_3	dx_4	dx_5	dx_6	dx_7	dx_8	dx_9	dx_{10}
(`intgr every res`) (`po`, `dt`, `dx`)	u_1	u_2	u_3	v_1	v_2	v_3	v_4	w_1	w_2	w_3
i_1(`po`, `dt`, `dx`)	u_1	u_2	u_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
i_4(`po`, `dt`, `dx`)	$\perp$	$\perp$	$\perp$	v_1	v_2	v_3	v_4	$\perp$	$\perp$	$\perp$
i_8(`po`, `dt`, `dx`)	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	w_1	w_2	w_3

`(intgr every res) (po, dt, dx)` first computes the result of i_1`(po, dt, dx)`, until the reset condition `res` is true, which occurs at the fourth instant. When `res` is true, a reconfiguration occurs and the system computes i_4`(po, dt, dx)`, and so on.

Looking at the clock of the `server` function, we can observe that the clock of `intgr` is `'a`. This means that `intgr` is a stream which is present when `'a` is true.

Now, we have to feed the function `main` with some input. We define the generic two-state automaton `switch` and a `feed` function.

```
let node switch (o1, o2, c1, c2) = o where
  rec (o, s1, s2) =
              if c1 & true -> pre s1 then
                (o2, false, true)
              else if false -> pre s2 then
                 if not c2 then
                   (o2, false, true)
                 else
                   (o1, true, false)
              else
                (o1, true, false)

node switch :  'a * 'a * bool * bool => 'a
node switch :: 'a * 'a * 'a * 'a -> 'a
```

The `switch` function expects four streams and returns a stream. Now, we have to transform the functions `rectangle` and `trapeze` into constant streams of functions. This is achieved by the `let/static` construction:

```
let static rectangle = rectangle
let static trapeze = trapeze

node rectangle : float * float * float => float
node rectangle :: 'a

node trapeze : float * float * float => float
node trapeze :: 'a
```

The construction `let static` x `=` e defines the stream x which is equal to the value of e. Such a definition is accepted by the compiler if the value of e does not evolve during the execution. In the above example, `rectangle` is a constant functional value and this is why it can be transformed into an infinite constant stream (we also say *lifted* as a stream).

The `feed` function selects the `trapeze` integrator as soon a the slope is too steep and switches to the `rectangle` integrator otherwise. To avoid oscillation, we introduce an hysteresis cycle[5].

```
let node feed (dx, thres1, thres2) = (intgr, new)
 where
  rec up = abs(dx) > thres2
  and down = abs (dx) < thres1
  and intgr = switch (rectangle, trapeze, up, down)
  and new = up or down

node feed :
  float * float * float
        => (float * float * float => float) * bool
node feed :: 'a * 'a * 'a -> 'a * 'a
```

Clearly, higher-order provides better modularity here. In particular, the `switch` function can be used equally well with scalars as with functions!

2.2 The need for dynamicity

Software-defined radio has recently emerged as an important research area for mobile telephone operators. Quoting Mitola, "a software radio is a radio whose channel modulation waveforms are defined in software" [23, 4][6]. The

[5]Using two thresholds to create a hysteresis cycle is standard practice in automatic control.

[6]`http://ourworld.compuserve.com/homepages/jmitola`

basic functionalities that both the emitter (e.g., the base station, the wireless network hub...) and the receiver (e.g., the cell phone terminal, the PDA...) must run are digital to analog conversion, analog to digital conversion, modulation/demodulation, radio frequency conversion, and so on. Software radio means that these functionalities are implemented as *software* modules run on general purpose hardware. There are two reasons for this:

- The increase in complexity of functions and components, coupled with the rapid changes in standards, requires the modification of the functionalities of existing base stations several times during their life. Thanks to software radio, these changes will be conducted by reprogramming some modules rather than by replacing some hardware or maintaining several infrastructures in parallel.
- The mobile terminal has to adapt seamlessly to its environment, for instance when moving from a UMTS zone to a WIFI zone. In order to achieve this, it must update its signal processing functions on the fly. There are even cases where the mobile terminal does not possess a copy of some of the modules that are needed to replace existing ones. Thanks to software radio, modules will be dynamically downloaded from the local base station to the mobile terminal.

A concrete example where dynamicity is required is the secure downloading procedure run by a terminal handset. It involves five cryptographic components [20]: three primitives (a hash function, a digital signature, and a symmetric key ciphering) and two keys (one secret and one public). The problem is that, at some point during the life of the terminal, a security flaw might be discovered, incriminating one or more of these components, therefore requiring their dynamic update. But to achieve this, the secure download procedure must be run, which requires the usage of a secure version of all five components. This is why each terminal possesses, for each component, *several* versions of it with different performance levels (teh most performant one being used). When one component currently in use is detected to be flawed, it is first dynamically replaced by another safer but slower version of itself, then this new version is used to download the patched version of the flawed component.

As of today, the existing solutions for programming reconfigurable systems are mainly based on middleware approaches (e.g., the Fractal component model [8], the RMA approach [24], the ERLANG language [1] used for programming networks switches) or general asynchronous process calculi [22, 26] and languages [15]. The drawback of middleware approaches is the absence of high-level constructs and clear semantics, in particular for time, communication and parallel composition. Asynchronous process calculi and associated languages are dedicated to global computing in an open asynchronous environment. They provide powerful features such as dynamic creation and reconfiguration as well as mobility features. Nonetheless, asynchrony comes at the price of non-determinism and there is no simple semantics of time, making their use in the context of safety critical embedded systems inappropriate. Moreover, identifying sub-parts of the calculus for which synchronous execution and efficient compilation can be guaranteed statically remains open.

Thus, starting from the observation that the synchronous model has proved to be well adapted for programming digital components, we propose here an extension of this model that uses higher-order features to address the above-mentioned needs. This is a first step toward reconfigurable systems, and we only address language aspects in the present paper.

As a final remark, it is important to notice that building our proposal on the synchronous model does not mean that the global system is expected to run in a synchronous manner; however, synchrony ensures that an asynchronous implementation of the system will not need synchronisation features at run-time involving possibly unbounded buffering [9]. Yet, the asynchronous execution of the proposed calculus and the generalization of distribution techniques such as [10] remains to be done.

2.3 Stream functions vs streams of functions

Lifting functional values to build *stream* values raises several problems. We investigate them informally in this section, before tackling them formally in the semantics (Section 3).

2.3.1 Instantiating a non constant stream of functions

The first problem is that one should be very careful when instantiating a non constant stream of sequential functions. Consider for instance `f` and `g`, which have the same type and clock signatures:

```
let node f x = y where rec y = 0 -> pre y + x

node f : int => int
node f : 'a -> 'a

let node g x = y where
  rec y = 0 -> pre (1 -> pre y + x) + pre y

node g : int => int
node g : 'a -> 'a
```

Note that `f` is a sequential function with only one memory, while `g` has two. Below is an example of a run of the `f` and `g` functions, fed with a simple constant stream of integers `z`:

z	1	1	1	1	1	1	1	1	...
f(z)	0	1	2	3	4	5	6	7	...
g(z)	0	1	2	4	7	12	20	33	...

The stream function `f` is instantiated by simply writing the application `f(z)`. Consider now the functions `f` and `g` lifted as the constant streams of functions `lf` and `lg`, and the non constant stream of functions `h`:

```
let static lf = f
let static lg = g
let h = lf -> lg
```

At the first instant `h` bears the value `f`, while at the subsequent instants it bears the value `g`. What happens if we instantiate `g` with the stream `z`? Computing `(g 1) (g 1) (g 1) (g 1)...` and transmitting from one instant to the next the values stored in the two memories is probably what the user expects since it coincides with `g(z)`. In other words, the *constant* stream of functions `g` instantiated with its argument `z` coincides with the result stream of the stream

function `g`, fed with the successive values of the stream of integers `z`.

But what happens now if we instantiate `h` with the same stream `z`? Computing `(f 1) (g 1) (g 1) (g 1)...` and trying to transmit the values stored in the memories does not work since `f` and `g` do not have the same number of memories! In contrast, building at each instant a fresh instantiation of the `h` function works but is costly and is probably not what the user expects. This is why we have introduced the construction `every` in Section 2.1: to allow the user to instantiate a stream of functions with some arguments and to specify exactly at what instants the instantiation of the code must be done. Thus, the regular application `h(z)` where `h` is a stream will be rejected by the compiler whereas the following program will be accepted:

```
let node main z c = y where
  rec h = lf -> lg
  and y = (h every c)(z)
```

That way, the stream of functions `lf -> lg` is instantiated (with a fresh memory) at each instant where `c` bears the value `true`.

2.3.2 Free variables

The second problem is raised by free variables. Consider for instance:

```
...
let node f x = x -> y
```

Here, `y` is a free variable of the stream function `f`. This function `f` can be instantiated with any stream of integers by writing simply `f(z)` if `z` is an integer stream. Now, if we consider `f` as a stream of functions, then it is legitimate to write `pre f`. But what does it mean? Actually, the value of `f` at a given instant can be memorized an arbitrary number of times (e.g., `pre(pre(pre f))`). And each time, the *context* of `f`, i.e., the values of its free variables, must be stored for further instantiation. Whereas emitting a function with evolving free variables is useful, its semantics is still unclear and we propose to reject those programs. Thus, we shall consider that only "constant" functions, that is, functions which do not depend on evolving free variables, can be transformed into streams. This is achieved with the `let/static` construction, which lifts a constant value into an infinite constant stream. Here, the following definition will be rejected, because `f` depends on some stream `y` which may evolve during the execution:

```
let static lf = f
```

3. SYNCHRONOUS SEMANTICS

In this section, we define a synchronous data-flow calculus into which the source programs given in Section 2 can be easily translated.

An expression e may be an immediate value imported from the host language (i), a variable (x), the application of an initialized delay (e `fby` e)[7], a combination (`merge` $e\ e\ e$), a sampling operator (e `when` e), a reconfiguration (e `every` e), and a pair (e, e) with its access functions `fst` and `snd`, a combinatorial function ($\lambda x.e$ `where` D), or finally a node ($\Lambda x.e$ `where` D).

Definitions (D) contain equations ($x = e$), parallel equations (D `and` D), sequential equations (D `in` D), the result of an application ($x = e(e)$), clock definitions (`clock`$(x) = e$), definitions of static values (`static`$(x) = e$), reseted definitions (`reset` $x = e(e)$ `every` x) and sequences of equations (`do` D `until` e `then` D).

$$
\begin{array}{rl}
e & ::= i \mid x \mid e\ \texttt{fby}\ e \mid \texttt{merge}\ e\ e\ e \mid e\ \texttt{when}\ e \\
 & \quad \mid (e, e) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid e\ \texttt{every}\ e \\
 & \quad \mid \lambda x.d \mid \Lambda x.d \\
d & ::= e\ \texttt{where}\ D \\
D & ::= x = e \mid \texttt{clock}(x) = e \mid \texttt{static}(x) = e \\
 & \quad \mid x = e(e) \mid D\ \texttt{in}\ D \mid D\ \texttt{and}\ D \\
 & \quad \mid \texttt{reset}\ x = e(e)\ \texttt{every}\ x \\
 & \quad \mid \texttt{do}\ D\ \texttt{until}\ e\ \texttt{then}\ D \\
i & ::= \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \ldots
\end{array}
$$

Our calculus has a *3-address code* flavor since every result of a computation (i.e., an application) must be named, following the presentation of the semantics of SIGNAL [3]. We adopt a slightly different syntax for our calculus to ease the description of the semantical rules.

Thus, the definition of a node `let node` $f\ x = e$ `where` D as given in Section 2 is translated into the definition $f = \Lambda x.e$ `where` D. The function definition `let` $f\ x = e$ `where` D is translated into the definition $f = \lambda x.e$ `where` D. A complex equation of the form $f = e_1(e_2)$ is translated into the sequential equations $x_1 = e_1$ `in` $x_2 = e_2$ `in` $f = x_1(x_2)$ where x_1 and x_2 are fresh variables.

In this calculus, the primitive `every` used in the introduction is a shortcut for:

$$f\ \texttt{every}\ c = \Lambda x.y\ \texttt{where reset}\ y = f(x)\ \texttt{every}\ c$$

The semantics is given by reaction rules. We define the set of instantaneous values (v), reaction environments (R), and four predicates defining reactions.

Values:	$v ::= p \mid (v, v) \mid \bot$
	$c ::= i \mid \lambda x.e$ `where` $D \mid \Lambda x.e$ `where` D
	$abs ::= \bot \mid (abs, abs)$
	$p ::= c \mid (p, p)$
Environment:	$R ::= [v_1/x_1, ..., v_n/x_n]$
Reaction:	$R \vdash e_1 \xrightarrow{v} e_2$
	$R \vdash D \xrightarrow{R'} D'$ with $R' \subseteq R$
Sequence:	$S ::= \epsilon \mid R.S$
Execution:	$S \vdash D : S'$

Instantaneous values (v) are made of absent values ($\bot$), immediate values (i), combinatorial or sequential functions, and pairs of values. We distinguish present values (p) from absent ones (abs). A composed value is present if all its components are present. A composed value is absent if all its components are absent.

$Dom(R)$ denotes the domain of R. $Var(R)$ is the set of free variables appearing in R. R_1, R_2 denotes the concatena-

[7] `fby` is the initialized delay such that e_1 `->` e_2 can be translated into `if true fby false then` e_1 `else` e_2. An uninitialized delay `pre` (e) can be translated into an initialized delay provided that expressions are correctly initialized [12].

tion of R_1 and R_2, provided there is no name conflict, that is, $Dom(R_2) \cap Dom(R_1) = \emptyset$. In other words, a variable defined in R_2 must not be already defined in R_1. Finally, rules are considered modulo renaming (α-conversion).

The predicate $R \vdash e_1 \xrightarrow{v} e_2$ means that the expression e_1 under the environment R emits the value v and rewrites into the expression e_2. The predicate $R \vdash D \xrightarrow{R'} D'$ means that the declaration D defines the instantaneous environment R' and rewrites into the declaration D'. The complete execution of a program, under a sequence of input environments $S = R_1.R_2...$ produces a sequence of environments $S' = R'_1.R'_2...$ so that:

$$\frac{R_{in}, R \vdash D \xrightarrow{R_{out}} D' \quad R_{out} \subseteq R \quad S \vdash D' : S'}{R_{in}.S \vdash D : R_{out}.S'}$$

During a synchronous reaction, computations may observe both input and local or output signals emitted during the reaction. This is why the reaction is computed in an extended environment R_{in}, R containing input, local, and output signals. Moreover, this extended environment R is temporary and can be removed at the end of the reaction.

These two predicates are formally defined in Figures 1, 2, and 3. Figure 1 gives the semantics to the set of synchronous primitives following standard formulation [18, 11]. The initialized delay e_1 `fby` e_2 returns the first value of e_1, then it emits the previous value of e_2. Thus, it is equivalent to e_1 `-> pre` (e_2). The sampling operator `when` is the one of LUSTRE. `merge` is borrowed from LUCID SYNCHRONE. The sampling operator `when` emits a value if its two inputs are present and the second one is `true`; the `merge` operator combines two complementary streams with opposite clocks.

Let us comment on rules in Figure 2. Constant values (immediate of functional ones) can be emitted or not (rules (Constant-*abs*) and (Constant)). A signal x emits its current value (rule (TAUT)). Rules (DEF) and (AND) are for variable definitions and parallel definitions respectively. Some definitions may introduce clock names (rule (Clock)) or static values (rule (Static)). A clock name is a particular boolean variable which can then be used to sample a stream (this construction has been first introduced in [13]). The `static` keyword states that the value is static and will not evolve during the execution of the program. The rules (app) and (App) express how functions are applied. A function is instantiated once by replacing itself by its body and arguments. Finally, the rules for building and un-building pairs are straightforward ((PAIR), (fst), and (snd)).

Finally, we add rules for reconfiguration operators (Figure 3). While the inputs of a `reset` operator are absent, a reset operator computes nothing (rule (Reset-*abs*)). When the condition is present, the body is instantiated and `reset` rewrites into a sequential operator `do/until` (rule (Reset)). The body D is computed while the condition e is false. When the condition becomes `true`, the continuation D_2 is activated (rules (Do-*abs*), (Do-t) and (Do-f)).

$$\text{(Fby-}abs\text{)}\frac{R \vdash e_1 \xrightarrow{abs} e'_1 \quad R \vdash e_2 \xrightarrow{abs} e'_2}{R \vdash e_1 \,\texttt{fby}\, e_2 \xrightarrow{abs} e'_1 \,\texttt{fby}\, e'_2}$$

$$\text{(Fby)}\frac{R \vdash e_1 \xrightarrow{p_1} e'_1 \quad R \vdash e_2 \xrightarrow{p_2} e'_2}{R \vdash e_1 \,\texttt{fby}\, e_2 \xrightarrow{p_1} p_2 \,\texttt{fby}\, e'_2}$$

$$\text{(When-}abs\text{)}\frac{R \vdash e_1 \xrightarrow{abs} e'_1 \quad R \vdash e_2 \xrightarrow{abs} e'_2}{R \vdash e_1 \;\texttt{when}\; e_2 \xrightarrow{abs} e'_1 \;\texttt{when}\; e'_2}$$

$$\text{(When-t)}\frac{R \vdash e_1 \xrightarrow{p} e'_1 \quad R \vdash e_2 \xrightarrow{\texttt{true}} e'_2}{R \vdash e_1 \;\texttt{when}\; e_2 \xrightarrow{p} e'_1 \;\texttt{when}\; e'_2}$$

$$\text{(When-f)}\frac{R \vdash e_1 \xrightarrow{p} e'_1 \quad R \vdash e_2 \xrightarrow{\texttt{false}} e'_2}{R \vdash e_1 \;\texttt{when}\; e_2 \xrightarrow{abs} e'_1 \;\texttt{when}\; e'_2}$$

$$\text{(Merge-}abs\text{)}\frac{R \vdash e_1 \xrightarrow{abs} e'_1 \quad R \vdash e_2 \xrightarrow{abs} e'_2 \quad R \vdash e_3 \xrightarrow{abs} e'_3}{R \vdash \texttt{merge}\; e_1\; e_2\; e_3 \xrightarrow{abs} \texttt{merge}\; e'_1\; e'_2\; e'_3}$$

$$\text{(Merge-t)}\frac{R \vdash e_1 \xrightarrow{\texttt{true}} e'_1 \quad R \vdash e_2 \xrightarrow{p} e'_2 \quad R \vdash e_3 \xrightarrow{abs} e'_3}{R \vdash \texttt{merge}\; e_1\; e_2\; e_3 \xrightarrow{p} \texttt{merge}\; e'_1\; e'_2\; e'_3}$$

$$\text{(Merge-f)}\frac{R \vdash e_1 \xrightarrow{\texttt{false}} e'_1 \quad R \vdash e_2 \xrightarrow{abs} e'_2 \quad R \vdash e_3 \xrightarrow{p} e'_3}{R \vdash \texttt{merge}\; e_1\; e_2\; e_3 \xrightarrow{p} \texttt{merge}\; e'_1\; e'_2\; e'_3}$$

Figure 1: Synchronous primitives

4. TYPES AND CLOCKS

4.1 Typing

Our calculus is statically typed and we give it an ML-like type system [21]. We distinguish type schemes (σ) which can be quantified from regular types (t). A regular type is made of basic atomic types (B), combinatorial function types ($t \xrightarrow{0} t$) and sequential function types ($t \xrightarrow{1} t$), product types ($t \times t$) and type variables (α). Combinatorial function types were printed $t_1 \rightarrow t_2$, while sequential functions (i.e., node) types were printed $t_1 \Rightarrow t_2$ in Section 2.

$$\begin{array}{rcl} \sigma & ::= & \forall\alpha.\sigma \mid t \\ t & ::= & B \mid t \xrightarrow{k} t \mid t \times t \mid \alpha \\ B & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \\ k & ::= & 0 \mid 1 \\ H & ::= & [x_1 : \sigma_1, ..., x_n : \sigma_n] \end{array}$$

4.1.1 Initial conditions, instantiation and generalization

Types can be instantiated or generalized over free type variables. A type scheme $\forall\alpha_1, ..., \alpha_n.t$ can be instantiated into a type $t[t_1/\alpha_1, ..., t_n/\alpha_n]$ by substituting types t_i to type variables α_i. A type t can be generalized into a type scheme $\forall\alpha_1, ..., \alpha_n.t$ if the type variables α_i do not appear free in the typing environment H. $FV(t)$ stands for the set of type variables (α) from t, $FV(\sigma)$ stands for the set of free type variables from σ and $FV(H)$ stands for the set of free type variables appearing in H.

$$t[t_1/\alpha_1, ..., t_n/\alpha_n] \leq \forall\alpha_1, ..., \alpha_n.t$$

$$\begin{array}{ll} gen_H(t) & = \forall\alpha_1, ..., \alpha_n.t \text{ where} \\ & \quad \{\alpha_1, ..., \alpha_n\} = FV(t) - FV(H) \\ gen_H(H_0)(x) & = gen_H(H_0(x)) \end{array}$$

$$\text{(Constant-}abs\text{)}\ R \vdash c \xrightarrow{abs} c \qquad \text{(Constant)}\ R \vdash c \xrightarrow{c} c \qquad \text{(TAUT)}\ R[v/x] \vdash x \xrightarrow{v} x$$

$$\text{(DEF)}\frac{R \vdash e \xrightarrow{v} e'}{R \vdash x = e \xrightarrow{[v/x]} x = e'} \qquad \text{(AND)}\frac{R \vdash D_1 \xrightarrow{R_1} D_1' \quad R \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash D_1 \texttt{ and } D_2 \xrightarrow{R_1,R_2} D_1' \texttt{ and } D_2'} \qquad \text{(SEQ)}\frac{R \vdash D_1 \xrightarrow{R_1} D_1' \quad R \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash D_1 \texttt{ in } D_2 \xrightarrow{R_1,R_2} D_1' \texttt{ in } D_2'}$$

$$\text{(Clock)}\frac{R \vdash e \xrightarrow{v} e'}{R \vdash \texttt{clock}(x) = e \xrightarrow{[v/x]} \texttt{clock}(x) = e'} \qquad \text{(Static)}\frac{R \vdash e \xrightarrow{v} e}{R \vdash \texttt{static}(x) = e \xrightarrow{[v/x]} \texttt{static}(x) = e}$$

$$\text{(app)}\frac{R \vdash e_1 \xrightarrow{\lambda y.e \texttt{ where } D} e_1' \quad R \vdash x = e \texttt{ and } y = e_2 \texttt{ and } D \xrightarrow{R'} D'}{R \vdash x = e_1(e_2) \xrightarrow{R'} D'}$$

$$\text{(App)}\frac{R \vdash e_1 \xrightarrow{\Lambda y.e \texttt{ where } D} e_1' \quad R \vdash x = e \texttt{ and } y = e_2 \texttt{ and } D \xrightarrow{R'} D'}{R \vdash x = e_1(e_2) \xrightarrow{R'} D'}$$

$$\text{(PAIR)}\frac{R \vdash e_1 \xrightarrow{v_1} e_1' \quad R \vdash e_2 \xrightarrow{v_2} e_2'}{R \vdash (e_1, e_2) \xrightarrow{(v_1,v_2)} (e_1', e_2')} \qquad \text{(fst)}\frac{R \vdash e \xrightarrow{(v_1,v_2)} e'}{R \vdash \texttt{fst } e \xrightarrow{v_1} \texttt{fst } e'} \qquad \text{(snd)}\frac{R \vdash e \xrightarrow{(v_1,v_2)} e'}{R \vdash \texttt{snd } e \xrightarrow{v_2} \texttt{snd } e'}$$

Figure 2: Behavioral synchronous semantics

Expressions and declarations are typed in an initial environment H_0.

$$\begin{aligned} H_0 = [&. \texttt{fby} . : \forall\alpha.\alpha \xrightarrow{0} \alpha \xrightarrow{1} \alpha,\\ &. \texttt{ when } . : \forall\alpha.\alpha \xrightarrow{0} \texttt{bool} \xrightarrow{1} \alpha,\\ &\texttt{merge } . \ . \ . : \forall\alpha.\texttt{bool} \xrightarrow{0} \alpha \xrightarrow{0} \alpha \xrightarrow{1} \alpha,\\ &\texttt{fst } . : \forall\alpha_1,\alpha_2.\alpha_1 \times \alpha_2 \xrightarrow{0} \alpha_1,\\ &\texttt{snd } . : \forall\alpha_1,\alpha_2.\alpha_1 \times \alpha_2 \xrightarrow{0} \alpha_2] \end{aligned}$$

Typing is obtained by asserting judgments of the form $H \overset{k}{\vdash} e : t$ and $H \overset{k}{\vdash} D : H_0$. The first one states that "the expression e has type t in environment H". The second one states that "the definition D is well typed in H and produces the typing environment H_0". $k = 0$ means that the expression is combinatorial (no internal state is modified during the computation) whereas $k = 1$ stands for a statefull expression. These two predicates are defined in Figure 4.

If a variable has a type scheme, then it can be instantiated (rule (Inst)). A local definition $x = e$ defines a local type environment (rule (Def)). An application is typed by first typing the function and then its argument (rule (app)). In doing so, we must check that the function is combinatorial if the current expression is expected to be combinatorial. This is ensured by the inequality $k_1 \leq k$.

Apart from the separation between combinatorial and sequential functions and up to syntactic details, this type system is a classical ML type system. It provides type polymorphism and type inference. In Lustre or Signal, programs are also statically typed but types are monomorphic and they are verified instead of being synthesized. Type polymorphism was also provided in Lucid Synchrone [25] but its type system has never been published so far. Moreover, the type system of Lucid Synchrone and its implementation were far more complex than what is presented here. In particular, it incorporated equational rules to deal with pairs of streams and stream of pairs, and streams of functions were forbidden. Finally, it appears that the separation between *combinatorial* and *sequential* functions introduced in the present calculus by giving them different types eases the communication between the host language and the dataflow part[8]. Thus, the present type system is both simpler and more expressive.

4.2 Clock calculus

The purpose of the type calculus is to check the *data* consistency of the program. A regular type gives some information on *what* is transported on a signal. The purpose of the clock calculus is to check the *time* consistency of the program. Thus, a clock can also be considered as a type that gives some information on *when* a value is transported on a signal.

In this paper, we adopt the *clock as type* approach which has been introduced already [11]. In this section, we define the clock type language and the clock calculus and base it on the system presented in [13].

The goal of the clock calculus is to produce judgments of the form $H \vdash e : cl$ for expressions and $H \vdash D : H_0$ for definitions. $H \vdash e : cl$ means that "the expression e has clock cl in the environment H". $H \vdash D : H_0$ means that "the definition D defines the local environment H_0 under the environment H".

$$\begin{aligned} \rho &::= \forall\alpha_1,...,\alpha_n.\forall X_1,...,X_m.cl \mid cl\\ cl &::= cl \to cl \mid cl \times cl \mid (c : s) \mid s\\ s &::= \texttt{base} \mid \alpha \mid s \texttt{ on } c\\ c &::= X \mid n\\ H &::= [x_1 : \rho_1, ..., x_n : \rho_n] \end{aligned}$$

We distinguish clock schemes (ρ) that can be quantified, from regular clocks (cl). A regular clock is made of function

[8]Such a separation already exists in industrial tools such as Scade.

$$\text{(Reset-}abs\text{)}\frac{R \vdash e_1 \xrightarrow{abs} e_1' \quad R \vdash e_2 \xrightarrow{abs} e_2' \quad R \vdash e_3 \xrightarrow{abs} e_3'}{R \vdash \texttt{reset}\ x = e_1(e_2)\ \texttt{every}\ e_3 \xrightarrow{[abs/x]} \texttt{reset}\ x = e_1'(e_2')\ \texttt{every}\ e_3'}$$

$$\text{(Reset)}\frac{R \vdash e_3 \xrightarrow{i} e_3' \quad R \vdash x = e_1(e_2) \xrightarrow{R'} D'}{R \vdash \texttt{reset}\ x = e_1(e_2)\ \texttt{every}\ e_3 \xrightarrow{R'} \texttt{do}\ D'\ \texttt{until}\ e_3'\ \texttt{then}\ \texttt{reset}\ x = e_1(e_2)\ \texttt{every}\ e_3'}$$

$$\text{(Do-}abs\text{)}\frac{R \vdash e \xrightarrow{abs} e' \quad R \vdash D_1 \xrightarrow{R_1} D_1 \quad R \vdash D_2 \xrightarrow{R_2} D_2}{R \vdash \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2 \xrightarrow{R_1} \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2}$$

$$\text{(Do-f)}\frac{R \vdash e \xrightarrow{\texttt{false}} e' \quad R \vdash D_1 \xrightarrow{R_1} D_1'}{R \vdash \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2 \xrightarrow{R_1} \texttt{do}\ D_1'\ \texttt{until}\ e'\ \texttt{then}\ D_2}$$

$$\text{(Do-t)}\frac{R \vdash e \xrightarrow{\texttt{true}} e' \quad R \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2 \xrightarrow{R_2} D_2'}$$

Figure 3: Behavioral synchronous semantics for imperative constructs

$$\text{(Inst)}\frac{t \leq H(x)}{H \overset{k}{\vdash} x : t} \qquad \text{(Clock)}\frac{H \overset{k}{\vdash} e : \texttt{bool}}{H \overset{k}{\vdash} \texttt{clock}(x) = e : [\texttt{bool}/x]}$$

$$\text{(Def)}\frac{H \overset{k}{\vdash} e : t}{H \overset{k}{\vdash} x = e : [t/x]} \qquad \text{(Static)}\frac{H \overset{0}{\vdash} e : t}{H \overset{k}{\vdash} \texttt{static}(x) = e : [t/x]}$$

$$\text{(app)}\frac{H \overset{k}{\vdash} e_1 : t_1 \xrightarrow{k_1} t_2 \quad H \overset{k}{\vdash} e_2 : t_1 \quad k_1 \leq k}{H \overset{k}{\vdash} x = e_1(e_2) : [t_2/x]}$$

$$\text{(fun)}\frac{H, x : t_1 \overset{0}{\vdash} d : t_2}{H \overset{k}{\vdash} \lambda x.d : t_1 \xrightarrow{0} t_2} \qquad \text{(Fun)}\frac{H, x : t_1 \overset{1}{\vdash} d : t_2}{H \overset{k}{\vdash} \Lambda x.d : t_1 \xrightarrow{1} t_2}$$

$$\text{(And)}\frac{H \overset{k}{\vdash} D_1 : H_1 \quad H \overset{k}{\vdash} D_2 : H_2}{H \overset{k}{\vdash} D_1\ \texttt{and}\ D_2 : H_1, H_2}$$

$$\text{(Seq)}\frac{H \overset{k}{\vdash} D_1 : H_1 \quad H, gen_H(H_1) \overset{k}{\vdash} D_2 : H_2}{H \overset{k}{\vdash} D_1\ \texttt{in}\ D_2 : H_2}$$

$$\text{(Reset)}\frac{H \overset{1}{\vdash} e_1 : t_2 \xrightarrow{1} t_1 \quad H \overset{1}{\vdash} e_2 : t_2 \quad H \overset{1}{\vdash} e_3 : \texttt{bool}}{H \overset{1}{\vdash} \texttt{reset}\ x = e_1(e_2)\ \texttt{every}\ e_3 : [t_1/x]}$$

$$\text{(Do)}\frac{H \overset{1}{\vdash} D_1 : H_1 \quad H \overset{1}{\vdash} e : \texttt{bool} \quad H \overset{1}{\vdash} D_2 : H_2 \quad H_2 \subseteq H_1}{H \overset{1}{\vdash} \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2 : H_2}$$

$$\text{(Where)}\frac{H, H_0 \overset{k}{\vdash} D : H_0 \quad H, H_0 \overset{k}{\vdash} e : t}{H \overset{k}{\vdash} e\ \texttt{where}\ D : t}$$

$$\text{(Pair)}\frac{H \overset{k}{\vdash} e_1 : t_1 \quad H \overset{k}{\vdash} e_2 : t_2}{H \overset{k}{\vdash} (e_1, e_2) : t_1 \times t_2}$$

Figure 4: Typing rules

clocks ($cl \to cl$), product clocks ($cl \times cl$), a dependence ($c : s$), and stream clocks (s). A stream clock may be the base clock (`base`), a clock variable (α), a sampled clock (s `on` c) on a condition c. Here, c can be either a name (n) or a variable (X).

$FV(cl)$ defines the set of free variables (α) of cl. $Dom(H)$ is the domain of H. $FN(cl)$ stands for the set of free name variables (X). $N(cl)$ is the set of names (n) of cl. Their definitions are straightforward and not given here.

Expressions and declarations are clocked in an initial environment H_0 giving clock types to synchronous primitives. The rules for these primitives are the one given in [13] and we do not go into much details. `fby` expects its two arguments to be on the same clock. An expression e_1 `when` e_2 is well clocked if e_1 and e_2 have the same clock α. In that case, the clock of the result is a sub-clock of α that we write α `on` X where X stands for the boolean value of e_2. The clock type ($X : \alpha$) given to e_2 says that e_2 has clock α and has some value X. `merge` expects a boolean stream and two complementary streams (that is, with opposite clocks).

$$\begin{aligned} H_0 = [&.\ \texttt{fby}\ . : \forall\alpha.\alpha \to \alpha \to \alpha,\\ &.\ \texttt{when}\ . : \forall\alpha.\forall X.\alpha \to (X : \alpha) \to \alpha\ \texttt{on}\ X,\\ &\texttt{merge}\ .\ .\ . : \forall\alpha.\forall X.\\ &\qquad (X : \alpha) \to \alpha\ \texttt{on}\ X \to \alpha\ \texttt{on not}\ X \to \alpha,\\ &\texttt{fst}\ . : \forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \to \alpha_1,\\ &\texttt{snd}\ . : \forall\alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \to \alpha_2] \end{aligned}$$

Clocks can be instantiated or generalized in the following way:

$$cl[\vec{s}/\vec{\alpha}][\vec{c}/\vec{X}] \leq \forall\vec{\alpha}.\forall\vec{X}.cl$$

$$\begin{aligned} gen_H(cl) &= \forall\alpha_1, ..., \alpha_n.\forall X_1, ..., X_m.cl \text{ where}\\ &\quad \{\alpha_1, ..., \alpha_n\} = FV(cl) - FV(H) \text{ and}\\ &\quad \{X_1, ..., X_m\} = FN(cl) - FN(H)\\ gen_H(H_0)(x) &= gen_H(H_0(x)) \end{aligned}$$

When clocking programs, we have to consider some equivalence rules between clocks. For example, the initialized delay `fby` receives a clock type $\forall\alpha.\alpha \to \alpha \to \alpha$, meaning

that it takes two streams of values and emits a stream of values with the same clock. But, what about the clock of an expression $(1,2)$ `fby` $(3,4)$? First, the clock of the pair of streams $(1,2)$ is $\alpha_1 \times \alpha_2$. As is, the clock type variable α cannot be instantiated by a pair clock (remind that a clock type variable is given to signals and can only be instantiated by stream clock types (s)). We say that $(1,2)$ is a *pair of streams* and it can be considered as a *stream of pairs* as soon as the clocks of the two streams can be *synchronized*, that is, they can be made equal. Thus, we consider that a structured value can receive a stream clock s when all its components can be synchronized on that clock. We shall write $cl \rhd_H s$ when a clock type cl can be synchronized to the stream clock s (called its *apparent clock*). The predicate is defined below:

$$\text{(Eq)}\ s \rhd_H s$$

$$\text{(Product)}\frac{cl_1 \rhd_H s \quad cl_2 \rhd_H s}{cl_1 \times cl_2 \rhd_H s}$$

$$\text{(Fun)}\frac{cl_1 \rhd_H s \quad cl_2 \rhd_H s \quad FV(s) \cap FV(H) = \emptyset}{cl_1 \to cl_2 \rhd_H s}$$

A pair clock $cl_1 \times cl_2$ can be synchronised with s as soon as both cl_1 and cl_2 can be synchronised with s. A function clock type $cl_1 \to cl_2$ can be considered as a stream of functions with apparent clock s as soon as cl_1 and cl_2 can be synchronised with clock s, and s can be fully generalized. This means that we have written a constant stream function. This is stated by rule (Fun).

The two predicates $H \vdash e : cl$ and $H \vdash D : H_0$ for clocking expressions and definitions are defined in Figure 5.

The system states that an immediate value i may receive any stream clock s (rule (Im)). The clock of a variable can be instantiated (rule (Inst)). In our calculus, clocks are introduced with a special keyword `clock`, following the approach first introduced in [13]. The `clock` construction states that x is a boolean stream with clock s (rule (Clock)). Moreover, this boolean stream can in turn be used to down-sample an other stream, and we give a unique symbolic value n so that only streams down-sampled with the same source name n can be considered to be synchronous. An expression e is a static expression if its clock can be synchronised with some clock s which can itself be fully generalised (rule (Static)). The clock rules for definitions, applications and functions are straitforward. The `reset` constructions expects all its argument to be synchronised on some clock s.

5. RELATED WORK

Higher-order features and dynamic reconfiguration has been extensively studied in the context of asynchronous and non-deterministic process calculi [22, 26]. These calculi are dedicated to global computing in an open network and thus, provide very powerful features (e.g., dynamic creation, migration, scope extrusion phenomena). Nonetheless, this expressivity comes at a price. In particular, recognizing subsets which are deterministic and can be executed in both bounded time and memory is hard. We address the more restrictive domain of embedded systems for which we need to ensure safety properties (e.g., execution in bounded time and memory, deadlock freedom). This is why we build our proposal on the *synchronous* model, and study the minimal extension that allows us to increase modularity and to

$$\text{(Im)}\ H \vdash i : s \qquad \text{(Inst)}\frac{cl \le H(x)}{H \vdash x : cl}$$

$$\text{(Clock)}\frac{H \vdash e : s \quad n \notin N(H)}{H \vdash \texttt{clock}(x) = e : [(n:s)/x]}$$

$$\text{(Static)}\frac{H \vdash e : cl \quad cl \rhd_H s \quad FV(s) \cap FV(H) = \emptyset}{H \vdash \texttt{static}(x) = e : [s/x]}$$

$$\text{(Def)}\frac{H \vdash e : cl}{H \vdash x = e : [cl/x]}$$

$$\text{(app)}\frac{H \vdash e_1 : cl_1 \to cl_2 \quad H \vdash e_2 : cl_1}{H \vdash x = e_1(e_2) : [cl_2/x]}$$

$$\text{(fun)}\frac{H, x : cl_1 \vdash d : cl_2}{H \vdash \lambda x.d : cl_1 \to cl_2} \qquad \text{(Fun)}\frac{H, x : cl_1 \vdash d : cl_2}{H \vdash \Lambda x.d : cl_1 \to cl_2}$$

$$\text{(And)}\frac{H \vdash D_1 : H_1 \quad H \vdash D_2 : H_2}{H \vdash D_1 \ \texttt{and}\ D_2 : H_1, H_2}$$

$$\text{(Seq)}\frac{H \vdash D_1 : H_1 \quad H, gen_H(H_1) \vdash D_2 : H_2}{H \vdash D_1 \ \texttt{in}\ D_2 : H_2}$$

$$\text{(Reset)}\frac{H \vdash e_1 : s \quad H \vdash e_2 : s \quad H \vdash e_3 : s}{H \vdash \texttt{reset}\ x = e_1(e_2)\ \texttt{every}\ e_3 : [s/x]}$$

$$\text{(Do)}\frac{H \vdash D_1 : H_1 \quad H \vdash e : s \quad H \vdash D_2 : H_2 \quad H_2 \subseteq H_1}{H \vdash \texttt{do}\ D_1\ \texttt{until}\ e\ \texttt{then}\ D_2 : H_2}$$

$$\text{(Where)}\frac{H, H_0 \vdash D : H_0 \quad H, H_0 \vdash e : cl}{H \vdash e\ \texttt{where}\ D : cl}$$

$$\text{(Pair)}\frac{H \vdash e_1 : cl_1 \quad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2}$$

Figure 5: Clocking rules

describe systems that can emit or receive programs.

This work is also related to the *reactive* approach pioneered by Boussinot [6, 7]. This approach combines the principles of synchronous programming — the existence of a global time shared by all the processes put in parallel — with the ability to dynamically create processes.

Besides the π-calculus and its variants for mobility [22, 26], Boudol has recently proposed ULM, "Un Langage pour la Mobilité" [5]. ULM has a traditional call-by-value functional and imperative core, enhanced with some reactive constructs for emitting signals, suspending, and aborting, and constructs for manipulating threads and mobile agents. ULM was inspired by SL [6] for the reactive part, and by process calculi for the mobility part. The principle is that a thread is suspended as soon as it refers to an absent signal. When all the threads are suspended or terminated, agents willing to migrate do so, possibly activating suspended threads.

Compared to these works, our proposal is built on a dataflow model *à la* Lustre instead of an imperative one. Moreover, programs can be compiled statically whereas they are scheduled dynamically in the reactive approach. Finally, our proposal provides static conditions (e.g., type and clocks) which are absent in the reactive approach.

6. CONCLUSION AND FUTURE WORK

Our contribution is a synchronous data-flow language with *higher order* features. The expressive power gained thanks to higher-order allows programs with dynamic loading and dynamic reconfiguration capabilities. These dynamic capabilities are essential to address the key challenges of embedded software like software defined radio, the future architecture of mobile phones and base stations. In contrast with most existing work, either based on middleware/OS or asynchronous process calculi approaches, our contribution is based on a synchronous programming language. There are several advantages: first the synchrony yields a clean parallel composition operation; and second it is a conservative higher-order extension of LUSTRE, meaning that the first-order part share the same semantics as LUSTRE, and can thus be compiled into sequential imperative code.

Future work is plethoric: the build of a compiler based on the existing LUCID SYNCHRONE implementation is under way. Extending and adapting classical program analyzes, such as causality analysis, to the language has also to be considered. Finally, we shall investigate clock-driven distribution techniques for programs. This will allow us to obtain automatically, from a unique and centralized higher-order data-flow program, a distributed program exchanging dynamically code over communication channels, with guarantees on the reaction time, on the maximal size of the communication buffers, and on the size of the memory necessary to run each distributed fragment of the program.

7. REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

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

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

[4] S. Blust. SDR forum roles and global work focus on radio software download. *IEICE Trans. on Communications*, E85-B(12):2581–2587, December 2002.

[5] G. Boudol. ULM: A core programming model for global computing. In *European Symposium on Programming, ESOP'04*, volume 2986 of *LNCS*, Barcelona, Spain, April 2004. Springer-Verlag.

[6] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4):256–266, April 1996.

[7] F. Boussinot and J.-F. Susini. The Sugarcubes tool box: A reactive Java framework. *Software-Practice and Experience*, 28(14):1531–1550, December 1998.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *Int. Symposium on Component Based Software Engineering, CBSE'04*, volume 3054 of *LNCS*, Edinburgh, Scotland, May 2004.

[9] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[10] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3):416–427, May/June 1999.

[11] P. Caspi and M. Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN Int. Conference on Functional Programming*, Philadelphia, USA, May 1996.

[12] J.-L. Colaço and M. Pouzet. Type-based Initialization of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming, SLAP'02*, volume 65.5 of ENTCS, Elsevier, 2002.

[13] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *Third Int. Conference on Embedded Software, EMSOFT'03*, Philadelphia, USA, October 2003.

[14] P. Cuoq and M. Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming, ESOP'01*, Genova, Italy, April 2001.

[15] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23th ACM Symposium on Principles of Programming Languages, POPL'96*. ACM, 1996.

[16] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal: A data-flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[18] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third Int. Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, Passau, Germany, August 1991.

[19] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP World Congress*. North Holland, Amsterdam, 1974.

[20] L.B. Michael, M. Mihaljević, S. Haruyama, and R. Kohno. Security issues for software defined radio: Design of a secure download system. *IEICE Trans. on Communications*, E85-B(12):2588–2600, December 2002.

[21] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[22] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, May 1999.

[23] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[24] K. Moessner, S. Hope, P. Cook, W. Tuttlebee, and R. Tafazolli. The RMA – a framework for reconfiguration of SDR equipment. *IEICE Trans. on Communications*, E85-B(12):2573–2580, December 2002.

[25] M. Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual*. UPMC, LIP6, May 2001. `http://www-spi.lip6.fr/lucid-synchrone`.

[26] D. Sangiorgi and D. Walker. *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.