Synchronous Functional Programming: The Lucid Synchrone Experiment *

Paul Caspi †
VERIMAG
Grenoble

Grégoire Hamon
The MathWorks
Boston

Marc Pouzet ‡
LRI, Univ. Paris-Sud 11
Orsay

1 Introduction

Lucid Synchrone is a programming language dedicated to the design of reactive systems. It is based on the synchronous model of Lustre [25] which it extends with features usually found in functional languages such as higher-order or constructed data-types. The language is equipped with several static analysis, all expressed as special type-systems and used to ensure the absence of certain run-time errors on the final application. It provides, in particular, automatic type and clock inference and statically detects initialization issues or dead-locks. Finally, the language offers both data-flow and automata-based programming inside a unique framework.

This chapter is a tutorial introduction to the language. Its aim is to show that techniques can be applied, at the language level, to ease the specification, implementation, and verification of reactive systems.

1.1 Programming Reactive Systems

We are interested in programming languages dedicated to the design of reactive systems [31]. Such systems interact continuously with their external environment and have to meet strong temporal constraints: emergency braking systems, plane autopilots, electronic stopwatches, etc. In the mid-70's, with the shift from mechanical and electronic systems to logical systems, the question of the computer-based implementation of reactive systems arose. At first, these implementations used available general-purpose programming languages, such as assembly, C, or Ada. Low-level languages were prominent as they give strong and precise control over the resources. Reactive systems often run with very limited memory and their reaction time needs to be statically known.

General-purpose, low-level languages quickly showed their limits. The low-level description they offer is very far from the specification of the systems, making implementation error prone and certification activities very difficult. A large number of embedded applications are targeting critical functions – fly-by-wire systems, control systems in nuclear power plants, etc. As such, they have to follow constraints imposed by certification authorities that evaluate the quality of a design before production use, for example the standards DO-178B in avionics and IEC-61508 in transport.

Better adapted languages were needed. The first idea was to turn to concurrent programming model, with the goal of getting closer to the specification. Indeed, reactive systems are inherently concurrent: the system and its environment are evolving in parallel and concurrency is the natural way to compose systems from more elementary ones. Unfortunately, traditional models of

*Chapter published in *Real-Time Systems: Description and Verification Techniques: Theory and Tools, vol. 1*, Editor: Nicolas Navet and Stephan Mertz. ISTE 2008 (Hermes publisher)

†Laboratoire VERIMAG, Centre Equation, 2, avenue de Vignate, 38610 GIERES, France. Email: Paul.Caspi@imag.fr

‡LRI, Université Paris-Sud 11, 91405 Orsay, France. Email: Marc.Pouzet@lri.fr

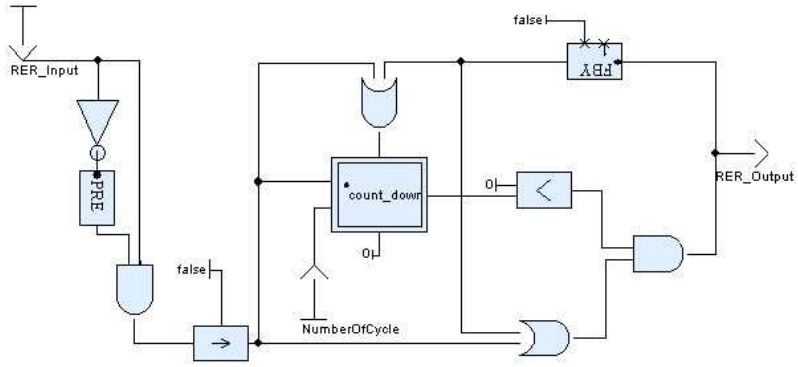


Figure 1: A SCADE design

concurrency come with numerous problems of system analysis, nondeterminism, or compilation. These models have therefore been seldom used in domains where security is a concern. Moreover, even if the goal is to get closer to a specification, the obtained models are far from formalisms traditionally used by engineers (such as continuous control and finite state transition systems).

1.1.1 The Synchronous Languages

Critical industries (e.g., avionics, energy), driven by high security needs, were the firsts to actively research new development techniques. An answer was proposed at the beginning of the 80's with the introduction of *synchronous languages* [3], the most famous of which are ESTEREL [5], LUSTRE [25] and SIGNAL [4]. These languages, based on a mathematical model of concurrency and time, are designed exclusively to describe reactive systems. The idea of dedicated programming languages with a limited expressive power, but perfectly adapted to their domain of application, is essential. The gain obtained lies in program analysis and the ability to offer compile-time guarantees on the runtime behavior of the program. For example, real-time execution and the absence of deadlocks are guaranteed by construction. Synchronous languages have been successfully applied in several industries and have helped in bringing formal methods to industrial developments.

Synchronous languages are based on the *synchronous hypothesis*. This hypothesis comes from the idea of separating the functional description of a system from the constraints of the architecture on which it will be executed. The functionality of the system can be described by making an hypothesis of instantaneous computations and communications, as long as it can be verified afterward that the hardware is fast enough for the constraints imposed by the environment. This is the classical *zero delay* model of electronics or control theory. It permits reasoning in logical time in the specification, without considering the real time taken by computations. Under the synchronous hypothesis, nondeterminism associated to concurrency disappears, as the machine is supposed to have infinite resources. It is possible to design high-level programming languages that feature parallel constructions, but can be compiled to efficient sequential code. The correspondence between logical time and physical time is verified by computing the worst case reaction time of the programs.

LUSTRE is based on the idea that the restriction of a functional language on streams, similar to LUCID [2] and the process network of Kahn [32], would be well adapted to the description of reactive systems. [1] LUSTRE is based on a data-flow programming model. A system is described as a set of equations over sequences. These sequences, or flows, represents the communications between the components of the system. This programming model is close to the specifications used by engineers of critical software thus reducing the distance between specification and implementation. In LUSTRE, the specification is the implementation. Moreover, the language is very well adapted to the application of efficient techniques for verification (see chapter 6 by Pascal Raymond), test [43],

[1] LUSTRE is the contraction of *Lucid, Synchrone et Temps Réel* — Lucid, Synchronous and Real-Time.

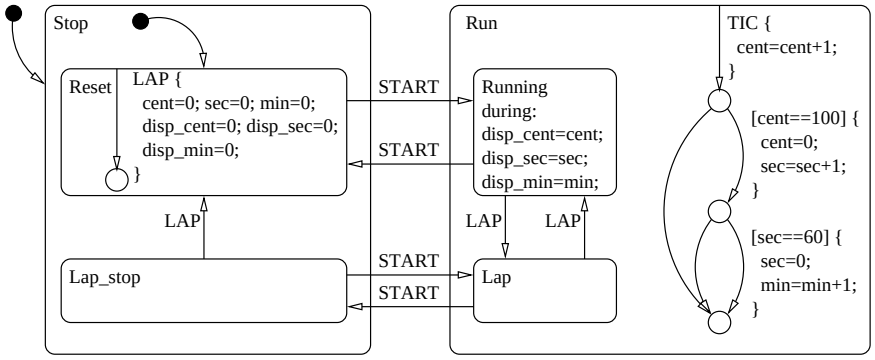


Figure 2: A Chronometer in STATEFLOW.

and compilation [26]. The adequation of the language with the domain and the efficiency of the technique that can be applied on it prompted the industrial success of LUSTRE and of its industrial version SCADE [44]. SCADE offers a graphical modeling of the LUSTRE equations, close to the diagrams used by control or circuits designers (figure 1). SCADE comes with formal validation tools and its code generator is certified (DO-178B, level A), reducing the need for testing. As of today, SCADE is used by numerous critical industries. It is for example a key element in the development of the new Airbus A380.

1.1.2 Model Based Design

Meanwhile, non-critical industries continued using general purpose languages, or low-level dedicated languages (norm IEC-1131). Security needs were much lower and did not justify the cost of changing development methods. It is pushed by the increasing size and complexity of systems, combined with a need for decreasing development cycles that they finally looked for other methods. It is the recent and massive adoption of *model based development environments*, the growing use of SIMULINK/STATEFLOW and the appearance of UML. Integrated environments offer a description of systems using high-level graphical formalisms close to automata or data-flow diagrams. They provide extensive sets of tools for simulation, test, or code-generation. Again, the distance between specification and implementation has been reduced, the system being automatically generated from the model.

These tools have usually not been designed with formal development in mind. Their semantics is often partially and informally specified. This is a serious problem for verification or certification activities. It is also a growing problem for the development activity itself: the complexity of the systems increasing, the risk of misinterpreting the models is also increasing. An example of such an environment is SIMULINK/STATEFLOW [38] (figure 2) an industrial *de-facto* standard. SIMULINK/STATEFLOW does not have a formal semantics and it is possible to write programs that will fail at runtime. On the other hand, the environment offers a complete set of tools, in which both the model and its environment can be modeled, simulated and code can be automatically generated.

1.1.3 Converging Needs

Today, the needs of all industries dealing with reactive systems, critical or not, are converging. They are all facing problems due to the increasing size and complexity of the systems. Synchronous languages, created to answer the needs of specifics domains (for example critical software for LUSTRE) are not directly adapted to the description of complex systems mixing several domains. The needs for verification and system analysis, for a time specific to critical industries, are now everywhere. The size of the systems requires automation of verification or test generation activities,

themselves required by the global diffusion of systems and the costs induced by conception errors that go unnoticed. These converging needs require in particular:

- abstraction mechanism at the language level that allow to get even closer to the specification of the systems, better reusability of components and the creation of libraries;
- techniques of analysis that are automatic and modular, and techniques of code generation that can offer guarantees on the absence of runtime errors;
- a programming model in which sampled continuous models, such as LUSTRE models, and control-based description, such as automata, can be arbitrarily composed.

1.2 Lucid Synchrone

The LUCID SYNCHRONE programming language was created to study and experiment extensions of the synchronous language LUSTRE with higher-level mechanisms such that type and clock inference, higher-order or imperative control-structures. It started from the observation that there was a close relationship between three fields, synchronous data-flow programming, the semantics of data-flow networks [32] and compilation techniques for lazy functional languages [48]. By studying the links between those fields, we were interested in answering two kinds of questions: (1) theoretical questions, on the semantics, the static analysis, and the compilation of synchronous data-flow languages; (2) practical questions, concerning the expressiveness of LUSTRE and the possibilities of extending it to answer needs of SCADE users.

This paper presents the actual development of the language and illustrate its main features through a collection of examples. This presentation is voluntarily made informal and technical details about the semantics of the language can be found in related references. Our purpose is to convince the reader of the various extensions that can be added to existing synchronous languages while keeping their strong properties for programming real-time systems. An historical perspective on the development of the language and how some of its features have been integrated into the industrial tool SCADE shall be discussed in section 3.

The core LUCID SYNCHRONE language is based on a synchronous data-flow model in the spirit of LUSTRE and embedded into a functional notation *à la ML*. The language provides the main operations of LUSTRE allowing to write, up to syntactic details, any valid LUSTRE program. It also provides other operators such as the `merge` that is is essential to combine components evolving on different rates. These features are illustrated in sections 2.1 to 2.4. The language incorporates several features usually found in functional languages such as higher-order or data-types and pattern-maching. Higher-order, which means that a function can be parameterised by an other function, allows to define generic combinators that can be used as libraries. Combined with the compiler's support for separate compilation, this is a key to increase modularity. Higher-order is illustrated in section 2.5. To answer the growing need for abstraction, the language supports constructed data-types. Values of that type naturally introduce control behavior and we associate them to a pattern-matching operation thus generalizing the so-called *activation condition* construction (or *enable-subsystems*) in graphical tools such as SCADE or SIMULINK. Data-types and pattern-matching are presented in section 2.6. Concurrent processes need to communicate and, in a dataflow language, this can only be done through inputs and outputs (so-called *in-ports* and *out-ports* in graphical languages), which can be cumbersome. LUCID SYNCHRONE introduces the notion of a *shared memory* to ease the communication between several modes of executions. This mechanism is safe in the sense that the compiler statically checks the non-interference between the uses of the shared memories, rejecting, in particular programs with concurrent writes. The language also supports *signals* as found in ESTEREL and which simplify the writing of control-dominated systems. Shared memory and signals are presented in section 2.7 and 2.8. Maybe the most unique feature of LUCID SYNCHRONE is its direct support of hierarchical state machines. This opens up new ways of programming, as it allows the programmer to combine in any way he wants dataflow equations and finite state-machine. State machines are illustrated in sections 2.9

and 2.11. Finally, being a functional language, it was interesting to investigate more general features such as recursion. This part is discussed in section 2.12.

Besides these language extensions, the language features several static analysis in order to contribute to make programming both easier and safer. In Lucid Synchrone, types are automatically inferred and do not have to be given explicitly given by the programmer. Automatic type inference and parametric polymorphism have been originally introduced in ML languages and have proved their interest for general purpose programming [41]. Nonetheless, they were not provided, as such, in existing synchronous tools. Type polymorphism allows to define generic components and increase code reuse while type inference frees the programmer from declaring the type of variables. The language is also founded on the notion of clocks as a way to specify the various paces in a synchronous system. In Lucid Synchrone, clocks are types and the so-called *clock calculus* of synchronous languages which aims at checking the coherency between those paces is defined as a type inference problem. Finally, the compiler provides two other static analysis also defined by special type systems. The initialization analysis checks that the behavior of the system do not depend on the initial values of delays while the causality analysis detects deadlocks.

2 Lucid Synchrone

We cannot present the language without a word about its two parents Objective Caml [35] on one side and Lustre on the other. The functional programmer may consider Lucid Synchrone as a subset of OCaml managing synchronous streams. The synchronous programmer will instead consider it as an extension of Lustre with functional features and control structures. The language has largely evolved since its first implementation in 1995. We present here the current version (V3) [2].

2.1 An ML Dataflow Language

2.1.1 Infinite Streams as Basic Objects

Lucid Synchrone is a *dataflow* language. It manages infinite sequences of streams as primitive values. A variable x stands for the infinite sequence of values $x_0, x_1, x_2, ...$ that x get during the execution. In the same way, `1` denotes the infinite sequence $1, 1, 1, ...$ The type `int` denotes the type of integer sequences. Scalar functions (e.g., `+`, `*`) apply point-wisely to streams:

`c`	t	f	...
`x`	x_0	x_1	...
`y`	y_0	y_1	...
`x+y`	$x_0 + y_0$	$x_1 + y_1$	...
`if c then x else y`	x_0	y_1	...
`if c then (x,y) else (0,0)`	(x_0, y_0)	$(0, 0)$	...

`x` and `y` being two integer streams, `x+y` produces a stream obtained by adding point-wisely the current values of `x` and `y`. Synchrony find a natural characterization here: at the instant i, all the streams take their i-th value.

Tuples of streams can be build and are considered as streams, allowing to have a homogeneous view of a system: every value is a time evolving value.

2.1.2 Temporal Operations: Delay and Initialization

The initialized delay `fby` (or *followed by*) comes from Lucid [2], the ancestor of dataflow languages. This operation takes a first argument that gives the initial value for the result and a second argument that is the delayed stream. In the following example, `x fby y` returns the stream `y` delayed and initialized with the first value of `x`.

[2]The manual and distributions are available at `www.lri.fr/~pouzet/lucid-synchrone`.

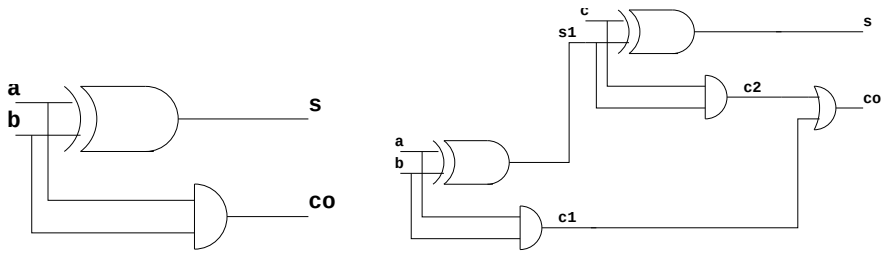


Figure 3: Hierarchical definition of a 1-bit adder

It is often useful to separate the delay from the initialization. This is obtained by using the uninitialized delay `pre` (for *previous*) and the initialization operator `->`. `pre x` delays its argument `x` and has an unspecified value *nil* at the first instant. `x -> y` returns the first value of `x` at its first instant then the current value of `y`. The expression `x -> pre y` is equivalent to `x fby y`.

x	x_0	x_1	x_2	...
y	y_0	y_1	y_2	...
x fby y	x_0	y_0	y_1	...
pre x	*nil*	x_0	x_1	...
x -> y	x_0	y_1	y_2	...

Since the `pre` operator may introduce undefined values (represented by *nil*), it is important to check that the program behavior does not depend on these values. This is done statically by the *initialization analysis*.

2.2 Stream Functions

The language makes a distinction between two kinds of functions: combinatorial and sequential. A function is combinatorial if its output at the current instant only depends on the current value of its input. This is a stateless function. A sufficient condition for an expression to be combinatorial is that it does not contain any delay, initialization operator or automaton. This sufficient condition is easily checked during typing.

A one-bit adder is a typical combinatorial function. It takes three boolean inputs `a`, `b` and a carry `c` and returns a result `s` and a new carry `co`.

```
let xor (a, b) = (a & not(b)) or (not(a) & b)

let full_add (a, b, c) = (s, co) where
       s = xor (xor (a, b), c)
   and co = (a & b) or (b & c) or (a & c)
```

When this program is given to the compiler, we get:

```
val xor :  bool * bool -> bool
val xor :: 'a * 'a -> 'a
val full_add :  bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
```

For every declaration, the compiler infers types (`:`) and clocks (`::`). The type signature `bool * bool -> bool` states that `xor` is a combinatorial function that returns a boolean stream when receiving a pair of boolean streams. The clock signature `'a * 'a -> 'a` states that `xor` is a length preserving function: it returns a value every time it receives an input. Clocks will be explained later.

A more efficient adder can be defined as the parallel composition of two half-adders as illustrated in figure 3.

```
let half_add (a,b) = (s, co)
  where s = xor (a, b) and co = a & b

let full_add(a,b,c) = (s, co) where
  rec (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c2 or c1
```

Sequential (or state-full) functions, are functions whose output at time n may depend on their inputs' history. Sequential functions are introduced with the keyword `node` and they receive a different type signature. A front edge detector `edge` is written:

```
let node edge c = false -> c & not (pre c)

val edge :  bool => bool
val edge :: 'a -> 'a
```

A possible execution is given in the following diagram.

c	f	f	t	t	f	t	...
false	f	f	f	f	f	f	...
c & not (pre c)	nil	f	t	f	f	t	...
edge c	f	f	t	f	f	t	...

The type signature `bool => bool` states that `edge` is a function from boolean streams to boolean streams and that its result depends on the history of its input. The class of a function is verified during typing. For example, forgetting the keyword `node` leads to a type error.

```
let edge c = false -> c & not (pre c)

>let edge c = false -> c & not (pre c)
>             ^^^^^^^^^^^^^^^^^^^^^^^^
This expression should be combinatorial.
```

In a dataflow language, stream definitions can be mutually recursive and given in any order. This is the natural way to describe a system as the parallel composition of subsystems. For example, the function that decrements a counter according to an boolean stream can be written:

```
let node count d t = ok where
  rec ok = (cpt = 0)
  and cpt = 0 -> (if t then pre cpt + 1 else pre cpt) mod d

val count : int -> int => bool
val count :: 'a -> 'a -> 'a
```

`count d t` returns the boolean value true when `d` occurrences of `t` have been received.

d	3	2	2	2	2	4	...
t	t	t	t	t	t	t	...
cpt	3	2	1	2	1	4	...
ok	t	f	f	t	f	t	...

The language is functional, it is possible to write partial applications by fixing one parameter of a function.

```
let count10 t = count 10 t

val count10 :  int => int
val count10 :: 'a -> 'a
```

2.3 Multi-sampled Systems

In all our previous examples, all the parallel processes share the same rate. At any instant n, all the streams take their n-th value. These systems are said to be single-clocked, a synchronous circuit is a typical example of a single-clocked system.

We now consider systems that evolve at different rates. We use a clock mechanism that was first introduced in LUSTRE and SIGNAL. Every stream s is paired with a boolean sequence c, or *clock*, that defines the instants when the value of s is present (precisely c is true if and only if s is present). Some operations allow to produce a stream with a slower (sampling) or faster (oversampling) rate. To be valid, programs have to verify a set of clock constraints, this is done through the *clock calculus*. In LUCID SYNCHRONE, clocks are represented by types and the clock calculus is expressed as a type-inference system.

2.3.1 The Sampling Operator `when`

The operator `when` is a sampler allowing to communicate values from fast processes to slower ones by extracting sub-streams according to a boolean condition.

`c`	f	t	f	t	f	t	...
`x`	x_0	x_1	x_2	x_3	x_4	x_5	...
`x when c`		x_1		x_3		x_5	...
`x whenot c`	x_0		x_2		x_4		...

The stream `x when c` is slower that the stream `x`. Supposing that `x` and `c` are produced at a clock ck, we say that `x when c` is produced at the clock ck `on` c.

The function `sum` that computes the sum of its input (i.e., $s_n = \Sigma_{i=0}^{n} x_i$) can be used at a slower rate by sampling its input:

```
let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum :  int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0
```

Whereas the type signature abstracts the value of a stream, the clock signature abstract its temporal behavior: it characterizes the instants where the value of a stream is available. `sampled_sum` has a functional clock signature that states that for any clock `'a` and boolean stream `_c0`, if the first argument `x` has clock `'a`, the second argument is equal to `_c0` and has clock `'a` then the result will receive the clock `'a on _c0` (variables like `c` are renamed to avoid name conflicts). An expression with clock `'a on _c0` is present when both its clock `'a` is true and the boolean stream `_c0` is present and true. Thus, an expression with clock `'a on _c0` has a slower rate than an expression with clock `'a`. Coming back to our example, the output of `sum (x when c)` is present only when `c` is true.

This sampled sum can be instantiated with a particular clock. For example:

```
let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten

val ten :  bool
val ten :: 'a
val sum_ten :  int => int
val sum_ten :: 'a -> 'a on ten
```

The keyword `clock` introduces a clock name (here `ten`) from a boolean stream. This clock name can in turn be used to sample a stream.

Clocks express control properties in dataflow systems. Filtering the inputs of a node according to a boolean condition, for example, means that the node will be executed only when the condition is true.

c	f	t	f	t	f	f	t	...
1	1	1	1	1	1	1	1	...
sum 1	1	2	3	4	5	6	7	...
(sum 1) when c		2		4			6	...
1 when c		1		1			1	...
sum (1 when c)		1		2			3	...

Thus, sampling the input of a sequential function f is not equivalent, in general, to sampling its output, that is, $(f(x \texttt{ when } c) \neq (fx) \texttt{ when } c)$.

Clocks can be nested arbitrarily. For example, a watch can be written:

```
let clock sixty = count 60 true
let node hour_minute_second second =
  let minute = second when sixty in
  let hour = minute when sixty in
  hour,minute,second

val hour_minute_second ::
     'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

A stream with clock `'a on sixty on sixty` is present only one instant over 3600 which is the expected behavior.

2.3.2 The Combination Operator `merge`

`merge` slow processes to communicate with faster ones by combining two streams according to a condition. Two arguments of a `merge` must have complementary clock.

c	f	t	f	f	f	t	...
x		x_0				x_1	...
y	y_0		y_1	y_2	y_3		...
merge c x y	y_0	x_0	y_1	y_2	y_3	x_1	...

Using the `merge` operator, we can define a holder (the operator `current` of Lustre) which hold the value of a stream between to successive sampling. Here, `ydef` is a default value used when no value have yet been received:

```
let node hold ydef c x = y
    where rec y = merge c x ((ydef fby y) whenot c)

val hold :  'a -> bool -> 'a => 'a
val hold :: 'a -> (_c0:'a) -> 'a on _c0 -> 'a
```

2.3.3 Oversampling

Using these two operators, we can define oversampling functions, that is, functions whose internal rate is faster than the rate of their inputs. In this sense, the language is strictly more expressive than Lustre and can be compared to Signal.

Oversampling appears naturally when considering a long duration task made during several time step (for example when its computation time is two important or because the architecture does not have enough resources to make the computation in one step). Consider the computation of the sequence $y_n = (x_n)^5$. This sequence can be computed by writing:

```
let power x = x * x * x * x * x
val power :: 'a -> 'a
```

The output is computed at the same rate as the input (as stated by the signature `'a -> 'a`). Four multiplications are necessary at every cycle. Suppose that only one multiplication is feasible at every instant. We can replace this instantaneous computation by an iteration through time by slowing the clock of `x` by a factor of four.

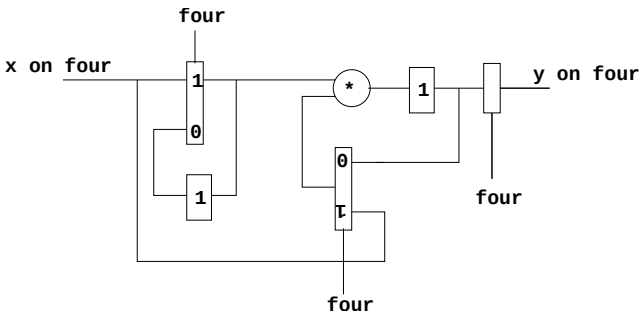


Figure 4: A example of oversampling

```
let clock four = count 4 true

let node iterate cond x = y where
  rec i = merge cond x ((1 fby i) whenot cond)
  and o = 1 fby (i * merge cond x (o whenot cond))
  and y = o when cond

let node spower x = iterate four x

val iterate :  bool -> int => int
val iterate :: (_c0:'a) -> 'a on _c0 -> 'a on _c0

val spower :  int => int
val spower :: 'a on four -> 'a on four
```

The corresponding dataflow network is given in figure 4.

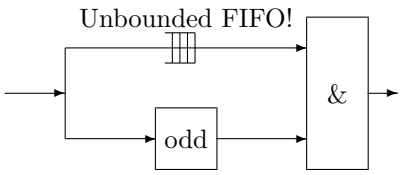
four	t	f	f	f	t	f	f	f	t
x	x_0				x_1				x_2
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2
o	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5
spower x	1				x_0^5				x_1^5
power x	x_0^5				x_1^5				x_2^5

Since the function `power` has the polymorphic clock signature `'a -> 'a`, the sequence `power x` has the same clock as `x`. Thus, `spower x` produces the same sequence as `1 fby (power x)` but with a slower rate.

An important consequence of the use of clocks and automatic inference is the possibility to replace every use of `(1 fby power x)` by `spower x` without modification of the rest of the program. The global system is automatically slowed-down to adapt to the clock constraints imposed by the function `spower`. This property contributes to the modular design and code reuse. Nonetheless, because clocks in Lucid Synchrone can be built from any boolean expression, the compiler is unable to infer quantitative information (which would allow to prove the equivalence between `(1 fby power x)` and `spower x`). `four` is considered by the compiler as a symbolic name and its periodic aspect is hidden. Recent works have considered an extension of synchronous language with periodic clocks [10, 14].

2.3.4 Clock Constraints and Synchrony

When stream functions are composed together, they must verify clock constraints. For example, the arguments of a `merge` must have complementary clocks and the arguments of an operator applied point-wisely must have the same clock. Consider for example the program:



`x`	x_0	x_1	x_2	x_3	x_4	x_5
`half`	t	f	t	f	t	f
`x when half`	x_0		x_2		x_4	
`x & (odd x)`	$x_0 \,\&\, x_0$		$x_1 \,\&\, x_2$		$x_2 \,\&\, x_4$	

Figure 5: A non synchronous program

```
let clock half = count 2 true
let node odd x = x when half
let node wrong x = x & (odd x)
```

This program adds the stream `x` to the sub-stream of `x` obtained by filtering one input over two [3]. This function would compute the sequence $(x_n \& x_{2n})_{n \in \mathbb{N}}$. Graphically, the computation of `x & (odd x)` corresponds to the Kahn network [32] depicted in figure 5. In this network, the input `x` is duplicated, one going through the function `odd` whose role is to discard one input over two. The two stream are in turn given to an `&` gate. If no value is lost, this network cannot be executed without a buffering mechanism, as time goes on, the size of the buffer will grow and will finally overflow. This program can not be efficiently compiled and boundedness execution can not be guaranteed. This is why we reject it statically. The goal of the clock calculus is to reject these type of program that cannot be executed synchronously without buffering mechanism [9].

In Lucid Synchrone, clocks are types and the clock calculus is expressed as a type inference problem [12, 19]. When the previous program is given to the compiler, it displays:

```
> let node wrong x = x & (odd x)
>                        ^^^^^^^
This expression has clock 'b on half,
but is used with clock 'b.
```

Using the formal definition of the clock calculus, we can show the following correction theorem: every well clocked program can be executed synchronously [12, 13]. This result can be compared to the type preservation theorem in ML languages [41]. Clocks being types, they can be used to abstract the temporal behavior of a system and be used as programming interfaces. Moreover, they play a fundamental role during the compilation process to generate efficient code. Intuitively, clocks become control-structures in the sequential code and the compiler gathers as much as possible all computations under the same clock (as soon as data-dependences are preserved).

2.4 Static Values

Static values are constant values that are useful to define parameterized system, these parameters being fixed at the beginning of an execution.

```
let static m = 100.0
let static g = 9.81
let static mg = m *. g
```

[3] In control-theory, `x when half` is a half frequency sampler.

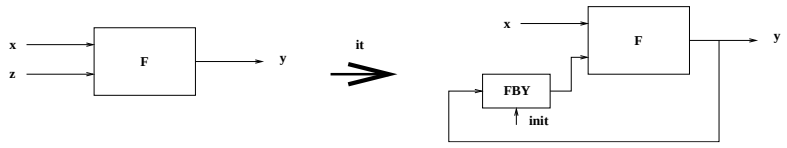


Figure 6: An Iterator

The compiler checks that a variable which is declared to be static is bound to an expression that produces a constant stream.

The language considered so far is not that different from LUSTRE. The main differences are the automatic type and clock inference and the operation `merge` which is an essential operation in order to program multi-sampled systems. We now introduce several extensions which are specific to the language.

2.5 Higher-order Features

Being a functional language, functions are first class objects which can be passed to functions or returned by functions. For example, the function `iter` is a serial iterator. Its graphical representation is given in figure 6.

```
let node iter init f x = y where
  rec y = f x (init fby y)

val iter : 'a -> ('b -> 'a -> 'a) => 'a
val iter :: 'a -> ('b -> 'a -> 'a) -> 'a
```

such that:

```
let node sum x = iter 0 (+) x
let node mult x = iter 1 ( * ) x
```

Higher-order is compatible with the use of clocks and we can thus write a more general version of the serial iterator defined in section 2.3.3.

```
let node iterate cond x init f = y where
  rec i = merge cond x ((init fby i) whenot cond)
  and o = init fby (f i (merge cond x (o whenot cond)))
  and y = o when cond

let node spower x = iterate four x 1 ( * )

val iterate :  bool -> 'a -> 'a -> ('a -> 'a) => int
val iterate ::
  (_c0:'a) -> 'a on _c0 -> 'a -> ('a -> 'a -> 'a ) -> 'a on _c0

val spower :  int => int
val spower :: 'a on four -> 'a on four
```

Higher-order is a natural way to define new primitives from basic ones. For example, the "activation condition" is a primitive operator in graphical tools like SCADE/LUSTRE or SIMULINK. It takes a condition `c`, a function `f`, a default value `default`, an input `input` and computes `f(input when c)`. It holds the last computed value when `c` is false.

```
let node condact c f default input = o where
  rec o = merge c (run f (input when c))
```

```
                ((default fby o) whenot c)

val condact :  bool -> ('a => 'b) -> 'b -> 'a -> 'b
val condact :: (_c0:'a) -> ('a on _c0 -> 'a on _c0) -> 'a -> 'a -> 'a
```

The keyword `run` states that its first argument `f` is a stateful function and thus has a type of the form $t_1 \Rightarrow t_2$ instead of $t_1 \rightarrow t_2$.

Using the primitive `condact`, it is possible to program a classical operator both available in the SCADE library and in the *digital* library of SIMULINK. Its graphical representation in SCADE is given in figure 1. This operator detects a rising edge (false to true transition). The output is true when a transition has been detected and is sustained for `numb_of_cycle` cycles. The output is initially false and a rising edge arriving while the output is true is still detected.

```
let node count_down (res, n) = cpt where
  rec cpt = if res then n else (n -> pre (cpt - 1))

let node rising_edge_retrigger rer_input numb_of_cycle = rer_output
  where
    rec rer_output = (0 < v) & (c or count)
    and v =
          condact count_down clk 0 (count, numb_of_cycle)
    and c = false fby rer_output
    and clock clk = c or count
    and count = false -> (rer_input & pre (not rer_input))
```

Higher-order is useful for the compiler writer as a way to reduce the number of basic primitives in the language. For the programmer, it brings the possibility to define operators and to build generic libraries. In LUCID SYNCHRONE, all the static analysis (types, clocks, etc.) are compatible with higher-order. Note that the higher-order features we have considered in the above examples, through it increase the modularity can still be statically expanded in order to get a first-order program.

2.6 Datatypes and Pattern Matching

Until now, we have only considered basic types and tuples. The language also supports structured datatypes. It is for example possible to define the type `number` whose value is either an integer or a float. The type `circle` defines a circle by its coordinates, its center and its radius.

```
type number = Int of int | Float of float
type circle = { center: float * float; radius: float }
```

Programs can be defined by pattern matching according to the structure of a value. Let us illustrate this on a wheel for which we want to detect the rotation. The wheel is composed of three colored sections with color blue (`Blue`), red (`Red`) and green (`Green`). A sensor observes the successive colors and must determine if the wheel is immobile and otherwise, its direction.

The direction is direct (`Direct`) for a sequence of `Red`, `Green`, `Blue`... and indirect (`Indirect`) for the opposite sequence. The direction may also be undetermined (`Undetermined`) or the wheel may be immobile (`Immobile`).

```
type color = Blue | Red | Green
type dir = Direct | Indirect | Undetermined | Immobile

let node direction i = d where
  rec pi = i fby i
  and ppi = i fby pi
  and d = match ppi, pi, i with
        (Red, Red, Red) | (Blue, Blue, Blue)
      | (Green, Green, Green) -> Immobile
```

```
| (_, Blue, Red) | (_, Red, Green)
| (_, Green, Blue) -> Direct

| (_, Red, Blue) | (_, Blue, Green)
| (_, Green, Red) -> Indirect

| _ -> Undetermined
end
```

The behavior is defined by pattern matching on three successive values of the input `i`. Each case (possibly) defines a set of equations. At every instant, the construction `match/with` selects the first pattern (from top to bottom) that matches the value of `(pii, pi, i)` and executes the corresponding branch. Only one branch is executed during a reaction.

This example illustrates the interest of the pattern matching construct in ML language: a compiler is able to check its exhaustiveness and completeness. In the LUCID SYNCHRONE compiler, this analysis strictly follows the one of OCAML [36].

2.7 A Programming Construct to Share the Memory

The pattern-matching construct is a control structure whose behavior is very different from the one of the `if/then/else` construct. During a reaction, only one branch is active. On the contrary, `if/then/else` is a strict operator and all its argument execute on the same rate. The pattern-matching construct corresponds to a `merge` which combines streams with complementary clocks.

With control structures, comes the problem of communicating between the branches of a control structure. This is a classical problem when dealing with several running modes in a graphical tool like SCADE or SIMULINK. Each mode is defined by a block diagram and modes are activated in an exclusive manner. When two modes communicate through the use of some memory, this memory must be declared on the outside of the block in order to contain the last computed value. To ease this communication, we introduce constructions to declare, initialize, and access a *shared memory* [4]. The last computed value of a shared variable `o` can be accessed by writing `last o`. Consider the following system. It has two modes, the mode *up* increment a shared variable `o` whereas the mode *down* decrements it.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last o + step done
      | false -> do o = last o - step done
      end
  and last o = 0
```

In the above program, the `match/with` construction combines equations where as it was applied to expressions in the previous example. The ability for a control-structure to combine equations eases the partial definition of some shared variables which may not have an explicit definition in one handler.

The equation `last o = 0` defines a shared variable `o` with initial value `0`. The communication between the two modes is made through the construction `last o` which contains the last computed value of `o`.

step	1	1	1	1	1	1	1	1	1	1	1	...
m	t	t	t	f	f	t	t	t	f	f	t	...
last o	0	1	2	3	2	1	2	3	4	3	2	...
o	1	2	3	2	1	2	3	4	3	2	3	...

[4] The SIMULINK tool provides a mechanism that ease the communication between several block diagram. This is mainly based on the use of imperative variables which can be read or written in different places.

This program has the same meaning as the following [5]. Here, the computation of the shared variable `last o` is done outside of the two branches thus en-lighting the fact that it contains the last computed value of `o`.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last_o + step done
      | false -> do o = last_o - step done
      end
  and last_o = 0 -> pre o
```

`last o` is another way to refer to the previous value of a stream and is thus similar to `pre o`. There is however a fundamental difference between the two. This difference is a matter of instant of observation. In a dataflow diagram, `pre` (e) denotes a local memory containing the value of its argument on the last time it has been observed. If `pre` (e) appears in a block that is executed from time to time, say on a clock c, this means that the argument e is computed and memorized only when the clock c is true. On the other hand, `last` o denotes the last value of the variable o at the clock where o is defined. `last` o is only defined on variables, not on expressions. `last` o is a way to communicate a value between two modes and this is why we call it a shared memory. The semantics of the previous program is precisely defined as:

```
let node up_down m step = o where
  rec o = merge c ((last_o when c) + (step when c))
                  ((last_o whenot c) - (step whenot c))
  and clock c = m
  and last_o = 0 -> pre o
```

In a graphical language, shared variables can be depicted differently to minimize the possible confusion between `last` o and `pre` o.

As a final remark, the introduction of shared memories allows to implicitly complement streams with their last values. Omitting an equation for some variable `x` is equivalent to adding an equation `x = last x`. This is useful in practice when modes defines several streams, you only need to define streams that are changing.

2.8 Signals and Signal Patterns

One difference between Lustre and Esterel is that the former manages streams whereas the later manages signals. Streams and signals differ in the way they are accessed and produced. Esterel distinguishes two kinds of signals: pure and valued. A pure signal is essentially a boolean stream, true when the signal is present and false otherwise. A valued signal is a more complex object: it can be either present of absent and when present, it carries a value. Signals exhibit an interesting feature for the programmer: they only exists when explicitly emitted. A stream on the other hand, must be defined in every instant. This feature leads to a more natural description of control-dominated systems.

Esterel-like signals are a particular case of clocked streams and can thus be simulated into a data-flow calculus without any penalty in term of static analysis or code-generation [16]. Signals are build and accessed through the use of two programming constructs, `emit` and `present`. A valued signal is simply a pair made of (1) a stream sampled on that condition c packed with (2) a boolean stream c — its clock — giving the instant where the signal is present [6]. The clock signature of a signal x is a dependent pair $\Sigma(c : \alpha).\alpha$ `on` c made of a boolean c with clock type α and a stream containing a value and whose clock type is α `on` c. We write α `sig` as a short-cut for the clock signature $\Sigma(c : \alpha).\alpha$ `on` c.

[5] This is precisely how the Lucid Synchrone compiler translates the first program.

[6] In circuit terminology, a signal is made of a value and an *enable* which indicate the instant where the value is valid [47].

2.8.1 Signals as Clock Abstractions

A signal can be built from a sampled stream by abstracting its internal clock.

```
let node within min max x = o where
  rec clock c = (min <= x) & (x <= max)
  and emit o = x when c

val within :  'a -> 'a -> 'a => int sig
val within :: 'a -> 'a -> 'a -> 'a sig
```

This function computes a condition `c` and a sampled stream `x when c`. The equation `emit o = x when c` defines a signal `o` present and equal to `x` when `c` is true. The construction `emit` encapsulates the value with its clock, corresponding to a limited form of existential quantification. This enters exactly in the existing [19] clock calculus based on the Laufer & Odersky type-system.

2.8.2 Testing Presence and Pattern Matching over Signals

The presence of a signal `x` can be tested using the boolean expression `?`x. For example, the following program counts the number of occurrences of the signal `x`.

```
let node count x = cpt where
  rec cpt = if ?x then 1 -> pre cpt + 1 else 0 -> pre cpt

val count :  'a sig => int
val count :: 'a sig -> 'a
```

The language offers a more general mechanism to test the presence of several signals and access their value. This mechanism is similar to pattern matching and is reminiscent to join-patterns in the Join-calculus [24].

The following program expects two input signals `x` and `y` and returns the sum of `x` and `y` when both signals are emitted. It returns the value of `x` when only `x` is emitted, the value of `y` when only `y` is emitted and `0` otherwise.

```
let node sum x y = o where
  present
    x(v) & y(w) -> do o = v + w done
  | x(v1) -> do o = v1 done
  | y(v2) -> do o = v2 done
  | _ -> do o = 0 done
  end

val sum :  int sig -> int sig => int
val sum :: 'a sig -> 'a sig -> 'a
```

Each handler is made of a filter and a set of equations. Filters are treated sequentially. The filter `x(v) & y(w)` is verified when both signals `x` and `y` are present. In that case, `v` and `w` are bounded to the values of the signals. If this filter is not satisfied, the second one is considered, etc. The last filter `_` defines the default case. Note that `x` and `y` are signal expressions whereas `v` and `w` are patterns. Thus, a filter `x(4) & y(w)` is read: await the instant where `x` is present and carry the value `4` and `y` is present.

Using signals, it is possible to mimic the `default` construction of SIGNAL. The expression `default x y` emits the value of `x` when `x` is present, the value of `y` when `x` is absent and `y` is present. `o` being a signal, it does not keep implicitly its last value in the remaining case (`x` and `y` absent) and it is considered absent.

```
let node default x y = o where
  present
    x(v) -> do emit o = v done
  | y(v) -> do emit o = v done
```

```
  end

val default :  'a -> 'a => 'a
val default :: 'a sig -> 'a sig -> 'a sig
```

This is only a simulation since the clock information — the precise instant where `x`, `y` and `default x y` are emitted — is hidden. The compiler is not able to state that `o` is emitted only when `x` or `y` are present.

The use of signals comes naturally when considering control dominated systems. Moreover, pattern matching over signals is safe in the sense that it is possible to access the value of a signal only when the signal is emitted. This is an important difference with ESTEREL where the value of a signal is implicitly sustained and can be accessed even when the signal is not emitted, thus raising initialization issues.

2.9 State Machines and Mixed Designs

In order to define control dominated systems, it is also possible to directly define finite state machines. These state machines can be composed with dataflow equations as well as other state machines and can be arbitrarily nested [17].

An automaton is a set of states and transitions. A state is made of a set of equations and transitions. Two types of transitions, *weak* and *strong* can be fired in a state and for each of them, the target state can be either entered by *history* or simply *reset*.

2.9.1 Weak and Strong Preemption

In a synchronous reaction, we can consider two types of transitions from a state: a transition is *weak* when it is inspected at the end of the reaction or *strong* when it is made immediately, at the beginning of the reaction. In the former case, the condition determines the active state of the next reaction, in the later it determines the current active set of equations to be executed.

Here is an example of a two state automaton with weak transitions:

```
let node weak_switch on_off = o where
  automaton
    Off -> do o = false until on_off then On
  | On -> do o = true until on_off then Off
  end
```

A state is made of several states (the initial one being the first in the list) and each state defines a set of shared variables. This automaton has two states `Off` and `On` each of them defining the current value of `o`. The keyword `until` indicates that `o` is false until the time (included) where `on_off` is true. Then, at the next instant, the active state becomes `On`. An automaton with weak transitions corresponds to a Moore automaton. On the contrary, the following function returns the value true as soon as the input `on_off` is true. Here, the value of `o` is defined by the equation `o = false` unless the condition `on_off` is verified. The condition is thus tested before executing the definitions of the state: it is called a strong transition.

```
let node strong_switch on_off = o where
  automaton
    Off -> do o = false unless on_off then On
  | On -> do o = true unless on_off then Off
  end
```

The graphical representation of these two automata is given in figure 7. We borrow the notation introduced by Jean-Louis Colaço and inspired by the SYNCCHARTS [1]: a strong transition is represented by an arrow starting with a circle which indicates that the condition is evaluated at the same instant as the target state. Reciprocally, a weak transition is represented by an arrow terminated by a circle, indicating that the condition is evaluated a the same instant as the source state.

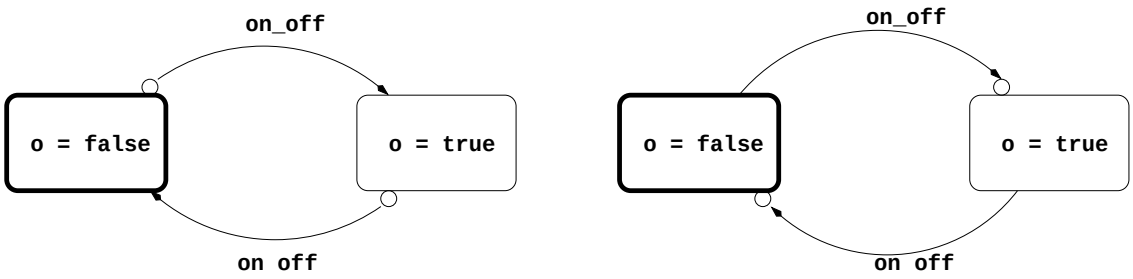


Figure 7: Weak and strong transitions in an automaton

on_off	f	f	t	f	f	t	f	t
weak_switch on_off	f	f	f	t	t	t	f	f
strong_switch on_off	f	f	t	t	t	f	f	t

We can notice that for any boolean stream `on_off`, `weak_switch on_off` produces the same sequence as `strong_switch (false -> pre on_off)`.

2.9.2 ABRO and Modular Reset

Adding automata to a dataflow language gives ESTEREL-like features. We illustrate it on the so-called ABRO example which en-lights the interest of synchronous composition and preemption [6]. Its specification is the following:

> "Await for the presence of events `a` and `b` and emit `o` at the precise instant where the two events have been received. Reset this behavior at every occurrence of `r`".

We first define a node `expect` that awaits for the presence of an event.

```
let node expect a = o where
  automaton
    S1 -> do o = false unless a then S2
  | S2 -> do o = true done
  end

let node abo a b = (expect a) & (expect b)
```

The node `abo` returns the value true and sustains it as soon as the inputs `a` and `b` is true. This is the parallel composition of two automata composed with an and gate. The node `abro` is obtained by making a new state automaton with a strong transition on the condition `r`. The target state is reset: every stream or automaton restarts in its initial configuration. This reset is indicated by the keyword `then`.

```
let node abro a b r = o where
  automaton
    S -> do o = abo a b unless r then S
  end
```

The construction `reset/every` is a short-cut for such an automaton.

```
let node abro a b r = o where
  reset
    o = abo a b
  every r
```

2.9.3 Local Definitions to a State

Each state in an automaton is made of a set of equations defining shared and local variables. Local variables can be used to compute the values of shared variables and weak transitions only. They cannot be used to compute strong transitions since strong transitions are tested at the beginning of the reaction. This is checked automatically by the compiler.

The following program sustains the value true for a duration `d1`, the value false for a duration `d2`, then restarts.

```
let node alternate d1 d2 = status where
  automaton
    True ->
      let rec c = 1 -> pre c + 1 in
      do status = true
      until (c = d1) then False
  | False ->
      let rec c = 1 -> pre c + 1 in
      do status = false
      until (c = d2) then True
  end
```

The state `True` defines a local variable `c` that is used to compute the weak transition `c = d1`.

2.9.4 Communication between States and Shared Memory

In the above example, there is no communication between the values computed in each state. The need for such communications appears naturally when considering several running modes. Consider the following three states automaton.

```
let node up_down go = o where
  rec automaton
        Init ->
           do o = 0 until go then Up
      | Up ->
          do o = last o + 1
          until (o >= 5) then Down
      | Down ->
          do o = last o - 1
          until (o <= - 5) then Up
      end
```

This program computes the sequence:

go	*f*	*f*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*
o	0	0	0	1	2	3	4	5	4	3	2	1	0

As the initial state is only weakly preempted, the value of `last o` is defined when entering the `Up` state. `last o` always contain the last computed value of `o`.

Every state defines the current value of a shared variable. In practice, it is heavy to define the value of a shared variable in every state. We need a mechanism to implicitly give a default value to those variables. A natural choice is to maintain the last computed value. Let us consider the example of a button to adjust an integer value from two boolean values [7]:

```
let node adjust p m = o where
  rec last o = 0
  and automaton
        Idle ->
           do unless p then Incr unless m then Decr
```

[7] This example is due to Jean-Louis Colaço and Bruno Pagano.

```
    | Incr ->
        do o = last o + 1 unless (not p) then Idle
    | Decr ->
        do o = last o - 1 unless (not m) then Idle
    end
```

The absence of equations in the initial state `Idle` means that `o` keeps its previous value. Said differently, an equation `o = last o` is implicitly added to the state.

2.9.5 Resume or Reset a State

When a transition is fired, it is possible to specify whether the target state is reset (and this is what has been considered in the previous examples) or not. The language allows to continue (or resume) the execution of a state in the configuration it had at its previous execution. For example:

```
let node up_down () = o where
  rec automaton
        Up -> do o = 0 -> last o + 1 until (o >= 2)
              continue Down
    | Down -> do o = last o - 1 until (o <= -2)
                continue Up
    end
```

The chronogram is now:

o	0	1	2	1	0	-1	-2	-1	0	1	2	-1

Let us notice that `last o` is necessarily defined when entering the state `Down` since the state is only left with a weak transition. Replacing it by a strong transition (`unless`) raises an initialization error.

2.10 Parameterized State Machines

We can now consider a more general class of automata where states are parameterized by some initial computed during the transition. This is useful to reduce the number of states and the communicate values between a source state and a target state. It also allows to express in a uniform manner special treatments when entering a state (e.g., *transitions on entry* of STATEFLOW). The following program counts occurrences of `x`.

```
let node count x = o where
  automaton
    Zero -> do o = 0 until x then Succ(1)
  | Succ(v) -> do o = v until x then Succ(v+1)
  end
```

Consider now a mouse controller whose specification is the following:

> "Produce the event `double` when the two events `click` have been received in less than four `top`. Emit `simple` if only one event `click` has been received"

This corresponds to a three states automaton:

```
let node counting e = cpt where
  rec cpt = if e then 1 -> pre cpt + 1 else 0 -> pre cpt

let node controller click top = (simple, double) where
  automaton
    Await ->
     do simple = false and double = false
     until click then One
```

```
  | One ->
     do simple = false and double = false
     unless click then Emit(false, true)
     unless (counting top = 4) then Emit(true, false)
  | Emit(x1, x2) ->
     do simple = x1 and double = x2
     until true then Await
  end
```

The controller awaits for the first occurrence of `click` then it enters in state `One` and counts the number of `top`. This state is strongly preempted when a second `click` is received or that the condition `counting top = 4` is true. For example, if `click` is true, the control goes immediately in state `Emit(false, true)`, giving the initial values `false` and `true` to the parameters `x1` and `x2`. Thus, at this instant, `simple = false` and `double = true`. At the next instant, the control goes to the initial state `Await`.

This example illustrates an important feature of state machines in Lucid Synchrone: only one set of equations is executed during a reaction. Nonetheless, it is possible to combines (at most) one strong transition followed by a weak transition and this is exactly what has been illustrated above. As opposed to other formalisms such as the StateCharts [30] or the SyncCharts [1], it is impossible to cross an arbitrary number of states during a reaction leading to simpler design and debugging.

2.11 Combining State Machines and Signals

Weak or strong transitions are not only made of boolean conditions, they can also test for the presence of signals as it was done with the `present` construct. We illustrate it on a system with two input signals `low` and `high` and an output stream `o`.

```
let node switch low high = o where
  rec automaton
        Init -> do o = 0 then Up(1)
      | Up(u) ->
          do o = last o + u
          unless low(v) then Down(v)
      | Down(v) ->
          do o = last o - v
          unless high(w) then Up(w)
      end

val switch :  'a sig -> 'a sig => 'a
val switch :: 'a sig -> 'a sig -> 'a
```

The condition `unless low(v) then Down(v)` is read: "goes in the parameter state `Down(v)` when the signal `low` is present and has the value v". The construct `do o = 0 then Up(1)` is a short-cut for `do o = 0 until true then Up(1)`.

high		3								2		
low				1		9			2	4		
o	0	1	2	1	0	-1	-2	-3	-4	-2	0	2

We can thus rewrite the specification of the mouse controller:

```
type e = Simple | Double

let node counting e = o where
  rec o = if ?e then 1 -> pre o + 1 else 0 -> pre o

let node controller click top = e where
```

```
  automaton
    Await ->
     do until click(_) then One
  | One ->
     do unless click(_) then Emit Double
     unless (counting top = 4) then Emit Simple
  | Emit(x) ->
     do emit e = x then Await
  end

val controller :  'a sig -> 'b sig => 'c sig
val controller :: 'a sig -> 'a sig -> 'a sig
```

Note that no value is computed in states `Await` and `One`. When writing `emit o = x`, the programmer states that `o` is a signal and thus, does not have to be defined in every state (or to implicitly complement its current value with `last o`). The signal `o` is only emitted in the state `Emit` and is absent otherwise.

The join use of signals and data-types exhibits an extra property with respect to the use of boolean to represent events: here, the output `o` has only three possible values (`Simple`, `Double` or absent) whereas the boolean encoding give four values (with one meaningless).

2.12 Recursion and non Real-time Features

Lucid Synchrone also provides a way to define recursive functions thus leading to possible non real-time executions. Nonetheless, this feature can be turned-off through a compilation frag, restricting the type system to only allow recursion on values with a statically bounded size.

A typical example of functional recursion is the sieve of *Eratosthene* as described in the seminal paper of Kahn [32] whose synchronous version has been given in [12]. These programs are still synchronous — parallel composition is the synchronous composition based on a global time scale — and streams can be efficiently compiled into scalar variables. Functional recursion models the dynamic creation of process and thus execution in bounded time and space is lost. Recursion can also be used through a static argument (as this is typically done in hardware functional languages such as Lava [7]). In that case, the program executes in bounded time and memory. Nonetheless, the current version of the compiler does not distinguishes them from the general case and they are thus rejected when the compilation flag is on. Several examples of recursive functions are available in the distribution [42].

2.13 Two Classical Examples

We end this presentation with two classical examples. The first one is an *inverted pendulum* programmed in a purely dataflow style. The second one is a simple controller for a heater and illustrates the combination of dataflow and automata.

2.13.1 The Inverted Pendulum

Consider an inverted pendulum with length l, its bottom part with coordinates (x_0, y_0) being manually controlled. θ is the angle of the pendulum. The physical low of the pendulum is given in figure 8.

We first define a module `Misc` to build an integrator and derivative (`*.` stands for the floating point multiplication).

```
(* module Misc *)
let node integr t x' =
  let rec x = 0.0 -> t *. x' +. pre x in x
let node deriv t x = 0.0 -> (x -.(pre x))/. t
```

The main module is written:

$$l\frac{d^2\theta}{dt^2} = (sin(\theta)(\frac{d^2y_0}{dt^2} + g)) - (cos(\theta)\frac{d^2x_0}{dt^2})$$
$$x = x_0 + lsin(\theta)$$
$$y = y_0 + lcos(\theta)$$

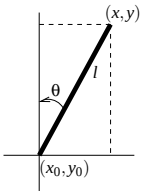


Figure 8: The Inverted Pendulum

```
(* module Pendulum *)
let static dt = 0.05 (* step *)
let static l = 10.0  (* length *)
let static g = 9.81  (* acceleration *)

let node integr x' = Misc.integr dt x'
let node deriv x = Misc.deriv dt x

(* the equation of the pendulum *)
let node equation x0'' y0'' = theta where
  rec theta =
    integr (integr ((sin thetap) *. (y0'' +. g)
                    -. (cos thetap) *. x0'') /. l)
  and thetap = 0.0 fby theta

let node position x0 y0 =
  let x0'' = deriv (deriv x0)  in
  let y0'' = deriv (deriv y0)  in

  let theta = equation x0'' y0'' in

  let x = x0 +. l *. (sin theta)  in
  let y = y0 +. l *. (cos theta)  in
  Draw.make_pend x0 y0 x y

let node main () =
  let x0,y0 = Draw.mouse_pos () in
  let p = Draw.position x0 y0 in
  Draw.clear_pendulum (p fby p);
  Draw.draw_pendulum p;;
```

The dot notation `Misc.integr` denotes the function `integr` from the module `Misc`.

This example also illustrates the communication between Lucid Synchrone and the host language (here OCaml) in which auxiliary functions are written. Here, the module `Draw` exports several functions (for example, `make_pend` creates a pendulum, `mouse_pos` returns the current position of the mouse).

2.13.2 A Heater

The second example is a controller for a gas heater depicted in figure 9.

The front of the heater has a green light indicating a normal functioning whereas a red light indicates that some problem has occurred (security stop). In case or problem, the heater is stopped. It can be restarted by pressing a restart button. Finally, it is possible to set the desired water temperature.

The controller has the following inputs: the stream `res` is used to restart the heater; `expected_temp` is the expected temperature; `actual_temp` is the actual water temperature; `light_on` indicates that the gas is burning; `dsecond` is a boolean stream giving the base rate of the system. The

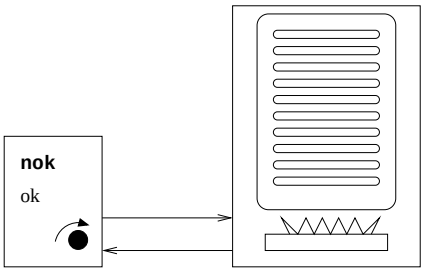


Figure 9: The Heater

output of the controller are the following: `open_light` lights on the gas; `open_gas` opens the gas valve; `ok` is true for a normal functioning whereas `nok` indicates a problem.

The purpose of the controller is to keep the water temperature close to the expected temperature. When the water needs to be heated, the controller turns on the gas and light for at most 500 milliseconds. When the light is on, only the gas valve is maintained open. If there is no light after 500 millisecond, it stops for 100 milliseconds and starts again. If after three tests there is still no light, the heater is blocked on a security stop. Only pushing the `res` button restarts the process.

```
let static low = 4
let static high = 4
let static delay_on = 500 (* in milliseconds *)
let static delay_off = 100

let node count d t = ok where
  rec ok = cpt = 0
  and cpt = (d -> pre cpt - 1) mod d

let node edge x = false -> not (pre x) & x
```

The following node decides whether the heater must be turned on. To avoid oscillations, we introduce an hysteresis mechanism . `low` and `high` are two threshold. The first version is purely dataflow, the second, while equivalent uses the automaton construction.

```
let node heat expected_temp actual_temp = on_heat where
  rec on_heat =
    if actual_temp <= expected_temp - low then true
    else if actual_temp >= expected_temp + high
         then false
         else false -> pre on_heat

let node heat expected_temp actual_temp = on_heat where
  rec automaton
        False ->
          do on_heat = false
          unless (actual_temp <= expected_temp - low)
          then True
      | True ->
          do on_heat = true
          unless (actual_temp >= expected_temp + high)
          then False
      end
```

Now, we define a node that turns on the light and gas for 500 millisecond then turn them off for 100 milliseconds and restarts:

```
let node command dsecond = (open_light, open_gas) where
  rec automaton
        Open ->
          do open_light = true
          and open_gas = true
          until (count delay_on dsecond) then Silent
      | Silent ->
          do open_light = false
          and open_gas = false
          until (count delay_off dsecond) then Open
      end
```

The program that control the aperture of the light and gas is written below:

```
let node light dsecond on_heat light_on =
                    (open_light, open_gas, nok) where
  rec automaton
        Light_off ->
          do nok = false
          and open_light = false
          and open_gas = false
          until on_heat then Try
      | Light_on ->
          do nok = false
          and open_light = false
          and open_gas = true
          until (not on_heat) then Light_off
      | Try ->
          do
            (open_light, open_gas) = command dsecond
          until light_on then Light_on
          until (count 4 (edge (not open_light)))
          then Failure
      | Failure ->
          do nok = true
          and open_light = false
          and open_gas = false
          done
      end
```

Finally, the main function connect the two components.

```
let node main
      dsecond res expected_temp actual_temp light_on =
                  (open_light, open_gas, ok, nok) where
  rec reset
        on_heat = heat expected_temp actual_temp
      and
        (open_light, open_gas, nok) =
           light dsecond on_heat light_on
      and
        ok = not nok
      every res
```

In all the above examples, we only describe the reactive kernel of the application. From these definitions, the Lucid Synchrone compiler produces a transition function written in OCaml which can in turned be linked to any other OCaml program.

3 Discussion

In this section we discuss related works and in particular the various embedding of circuit description languages or reactive languages inside general purpose functional languages. Then, we give an historical perspective on Lucid Synchrone and its use as a prototyping language for various extension of the industrial tool Scade.

3.1 Functional Reactive Programming and Circuit Description Languages

The interest of using a lazy functional language for describing synchronous circuits has been identified in the early eighties by Mary Sheeran in μFP [45]. Since then, various languages or libraries have been embedded inside the language Haskell for describing circuits (Hydra [40], Lava [7]), the architecture of processors (for example, Hawk [39]), reactive systems (Fran [23], frp [49]). Functional languages dedicated to circuit description have also been proposed (for example, Jazz [47], ReFleCT [33]). Circuits and dynamical systems can be modeled directly in Haskell, using module defining basic operations (for example, registers, logical operations, stream transformers) in a way very similar to what synchronous languages such as Lustre or Lucid Synchrone offer. The embedding of these domain specific languages inside Haskell benefits from the expressive power of the host language (typing, data and control structures). The class mechanism of Haskell [27] can also be used to easily change the representation of streams in order to obtain the verification of a property, a simulation or a compilation. *Multi-stage* [46] techniques are another way to describe a domain specific language. This approach through an embedding inside a general purpose language is well adapted to circuit description language where the compilation result is essentially a net-list of boolean operators and registers. This is nonetheless limited when a compilation to software code is expected (as it is mainly the case for Scade/Lustre). Even if a net-list can be compiled into sequential code, the obtained code is very inefficient due to code size increase. Moreover, when non length preserving functions are considered, as is the case in Fran or frp, real-time (execution in bounded time and memory) can not be statically guaranteed [8]. These works do not provide a notion of clock, control structures mixing dataflow systems and automata nor compilation techniques with dedicated type-systems. The choice we have made in Lucid Synchrone was to reject more programs in order to obtain more guarantees at compile time (e.g., synchronous execution, absence of deadlocks). Moreover, only software compilation has been considered in our work.

3.2 Lucid Synchrone as a Prototyping Language

The development of Lucid Synchrone started around 1995 so as to serve as a laboratory for experimenting various extensions of synchronous languages. The firsts works showed that it was possible to define a functional extension of Lustre, combining the expressiveness of functional programming using lazy lists, with the synchronous efficiency [11]. The clock calculus was defined as a dependent type system and generalized to higher-order [12]. The next question was to understand how to extend compilation techniques. This was answered by using co-algebraic techniques to formalize and to generalize the techniques already used to compile Scade [13]. These results combined together lead to the first implementation of a compiler (V1) and had important practical consequences. Indeed, the definition of the clock calculus as a type system is the key to clock inference, which makes them much easier to use. The introduction of polymorphism is an important aspect of code reusability [15]. Automatic inference mechanisms are essential in a graphical tool like Scade, in which programs are mainly drawn. They can also be found, but hidden, in tools like Simulink [10].

A partnership started during year 2000 with the Scade development team at Esterel-Technologies (Telelogic at the time), to write a new Scade compiler. This compiler, ReLuC

[8] Consider, for example, the program in figure 5.

(for *Retargetable Lustre Compiler*), uses the results already applied in Lucid Synchrone, the clock calculus through typing, as well as new constructions of the language.

The basis of the language being there, work then turned to the design of modular type-based analysis: causality loops analysis [22] and initialization analysis [18, 20]. The initialization analysis has been developed in collaboration with Jean-Louis Colaço at Esterel-Technologies and used at the same time in the Lucid Synchrone compiler and the ReLuC compiler. On real-size examples (more than 50000 lines of codes), it proved to be really fast and precise, reducing the number of wrong alarms. Work on the clock calculus also continued. As it was expressed as a dependent type system, it was natural to embed it in the Coq [21] proof assistant, thus getting both a correction proof [8] and some introspection on the semantics of the language. By looking at Scade designs, we observed that this calculus could be also turned into an ML-like type system basing it on the Laüfer & Odersky extension [34]. Although less expressive than the dependent type one, it is expressive enough in practice and can be implemented much more efficiently [19]. It is used in the current version of the compiler (V3) and replace the dependent-type based clock calculus of ReLuC. At the same time, several language extensions were also studied: a modular reinitialization construct [29], the addition of sum types, and a pattern matching operation [28]. These works show the interest of the clock mechanism of synchronous languages, present since the beginning in both Lustre and Signal. Clocks provide a simple and precise semantics for control structures that can be translated into the code language, thus reusing the existing code generator. This work was pursued in collaboration with Esterel-Technologies and leaded to the proposition of an extension of Lustre with hierarchical automata [17, 16] in the spirit of the *Mode-automata* of Maraninchi and Rémond [37]. This extension is also based on the clock mechanism and automata are translated into the core language. This extension is implemented in Lucid Synchrone (V3) and in the ReLuC compiler at Esterel-Technologies. All these developments are integrated in Scade 6, the next release of Scade.

4 Conclusion

This paper has presented the actual development of Lucid Synchrone. Based on the synchronous model of Lustre but reformulated in the framework of typed functional languages, it offers higher-order features, automatic type and clock inference and the ability to describe, in a uniform way, data and control dominated systems.

The Lucid Synchrone experiment illustrates the various extensions that can be done in the field of synchronous languages to increase their expressive power and safety while retaining their basic properties for describing real-time systems. Two natural directions can be drawn from this work. One is about the link with formal verification and proof systems (such as Coq) with certified compilation and proof of programs in mind. The other concerns the integration of some of the principles of synchronous programming as a general model of concurrency (not limited to real-time) inside a general purpose language.

5 Acknowledgment

The work on Lucid Synchrone has been mainly developed at the Université Pierre et Marie Curie and greatly benefited from discussions with Thérèse Hardin. It is also the result of a long collaboration with Jean-Louis Colaço from Esterel-Technologies. We thank them warmly.

References

[1] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at: `www-mips.unice.fr/~andre/syncharts.html`.

[2] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. A.P.I.C. Studies in Data Processing, Academic Press, 1985.

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

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

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] Gérard Berry. The esterel v5 language primer, version 5.21 release 2.0. Draft book, 1999.

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP)*. ACM, 1998.

[8] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

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

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

[11] Paul Caspi and Marc Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[12] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[13] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[14] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[15] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[16] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

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

[18] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[19] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[20] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):245–255, August 2004.

[21] The coq proof assistant, 2007. http://coq.inria.fr.

[22] Pascal Cuoq and Marc Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[23] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May-June 1999.

[24] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[26] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[27] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[28] Grégoire Hamon. Synchronous Data-flow Pattern Matching. In *Synchronous Languages, Applications, and Programming*. Electronic Notes in Theoretical Computer Science, 2004.

[29] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

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

[31] David Harel and Amir Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Verlag, 1985.

[32] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[33] Sava Krstic and John Matthews. Semantics of the reFLect Language. In *PPDP*. ACM, 2004.

[34] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[35] Xavier Leroy. The Objective Caml system release 3.10. Documentation and user's manual. Technical report, INRIA, 2007.

[36] Luc Maranget. Les avertissements du filtrage. In *Actes des Journées Francophones des Langages Applicatifs*. Inria éditions, 2003.

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

[38] The Mathworks. *Stateflow and Stateflow Coder, User's Guide*, release 13sp1 edition, September 2003.

[39] J. Matthews, J. Launchbury, and B. Cook. Specifying microprocessors in hawk. In *International Conference on Computer Languages*. IEEE, 1998.

[40] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. In Springer-Verlag, editor, *Functional Programming Languages in Education*, pages 195–214, 1995.

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[42] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.

[43] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[44] SCADE. `http://www.esterel-technologies.com/scade/`, 2007.

[45] Mary Sheeran. mufp, a language for vlsi design. In *ACM Conference on LISP and Functional Programming*, pages 104–112, Austin, Texas, 1984.

[46] Walid Taha. Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999.

[47] Jean Vuillemin. On Circuits and Numbers. Technical report, Digital, Paris Research Laboratory, 1993.

[48] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[49] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.