Lustre, Fast First and Fresh

Timothy Bourke, Marc Pouzet

Lustre, fast first and fresh

Timothy Bourke and Marc Pouzet

***Abstract*—The rate-synchronous model formalizes an industrial approach for composing Lustre nodes that execute at different rates. Such programs are compiled to cyclic sequential code in two steps. First, an Integer Linear Program is solved to assign each component to a phase relative to its period. Second, the corresponding step functions are ordered for execution within a cycle of the generated code. By default, programs are deterministic: for any valid schedule, the generated code calculates the values decreed by the source dataflow semantics at the specified rates. In practice, though, specifying precise values in the source program is sometimes unnecessary, impracticable, and overly constraining. In this case, the ILP constraints can be relaxed, though not necessarily completely, and their solution decides which dataflow semantics applies. Care is still required to ensure that code generation remains deterministic.**

***Index Terms*—Time-Centric Reactive Software, Dataflow Synchronous Programming.**

I. Introduction

A Lustre [1] program comprises a hierarchy of *nodes*, which are functions that map streams of input values to streams of output values. Within a node, a set of equations defines the values of local and output signals using a simple expression language. Equivalently, a node is specified by a dataflow graph whose directed arcs are labelled by signal names and whose vertexes specify computations. Nodes are given a synchronous semantics so that they can be compiled to code that executes in bounded time and memory. This is usually taken to mean that (i) all streams advance at the same rate so that one cycle of the generated code samples one value of each input and calculates one value of each output, and (ii) all calculations complete within a cycle, so that if the worst-case execution time (WCET) of the generated code is less than its execution period then correct timing behavior is ensured. Within a node, it is possible to sample a signal `x` on an arbitrary boolean signal `b`, written `x when b`, and thereby to specify slower computations. For example, given `y = f(x when b)`, the node `f` is applied to a filtered substream, which is realized in the generated code by introducing nested conditionals so that `y` is only updated in cycles where `b` is true: `if (b) { y = f(x); }`. A type system [2] ensures that variables are only read in cycles where they are written, and thus also that buffering is unnecessary.

A recent article [3] proposed a variation of Lustre that formalizes a long-standing practice at Airbus. After recalling the details of the proposed *rate-synchronous language*, we will focus on its nondeterministic aspects and make some new observations on the treatment of cycles during scheduling and compilation.

T. Bourke is with Inria and ENS, PSL University, CNRS, Paris, France.
M. Pouzet is with ENS, PSL University, CNRS, Paris, France and Inria.
Manuscript received TODO, 2024; revised TODO, 2024.

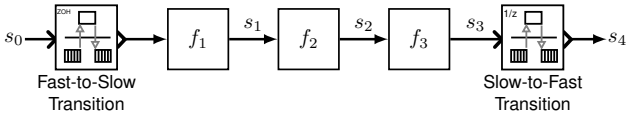


Fig. 1. Block diagram of 3-function pipeline.

II. A Deterministic Rate-Synchronous Language

The rate-synchronous language was originally motivated by a flight control and guidance system comprising approximately 5000 nodes and over 120 000 named signals. The language's goal was to allow the nodes to be instantiated within a single `main` node by providing constructions for specifying execution rates and resource constraints. The Prelude language [4], [5] was developed with the same goal and targets a set of tasks for execution by a real-time scheduler. In contrast, rate-synchrony abstracts from WCET and signal phases, and focuses on statically-scheduled sequential code generation.

a) Example: To illustrate the language, consider the simple system sketched in figure 1 using a syntax inspired by Simulink.[1] An input signal s_0 is resampled at a slower rate, and then processed by three successive filters f_1, f_2, and f_3, before being buffered as the output signal s_4. Suppose that the filters are to run three times slower than the base rate of the system. The fast-to-slow rate transition must then choose one of every three values to pass to f_1. Similarly, the slow-to-fast transition must propagate an initial value zero, one, or two times before repeating each of its input values three times. In our extension of Lustre, we could write

```
s1 = f1(s0 when (0 % 3));
s2 = f2(s1);
s3 = f3(s2);
s4 = current(s3, (2 % 3));
```

where we have chosen to sample the first of every three values of s_0; and to produce s_4 by propagating an initial value two times and then repeating every value of s_3 three times.

If we set $s_0 = 1, 2, \ldots$ and $f_i(x) = x + 10$, and declare s_3 with an *initial last value* of 0, then the behavior of the system can be visualized by the grid below.

s_0	1	2	3	4	5	6	7	8	9	$\cdots$
s_1	11			14			17			$\cdots$
s_2	21			24			27			$\cdots$
s_3	31			34			37			$\cdots$
s_4	0	0	31	31	31	34	34	34	37	$\cdots$

Significantly, the width of the columns in the grid varies with the execution rate of the corresponding signal. Each value of s_1, s_2, and s_3 is conceptually synchronous with three values

[1] https://www.mathworks.com/help/simulink/slref/ratetransition.html

of s_0 and s_4. This is the defining characteristic of the rate-synchronous model. In a normal Lustre program, all columns would have the same width and the slower values would be aligned, using empty cells or absence markers, with specific instants of the base clock. Specifying distinct alignments for the slower calculations would require explicit buffering between the f_i, as in an earlier proposition [6, §4.2] and the n-synchronous model [7, §4.3.1]. This idea translates directly to more complicated programs and to a formal semantics [3, Figure 2(a)] that is especially simple because the `when` is productive (all streams are infinite) and absence is not made explicit. For instance, the streams ($\mathbb{N}_0 \to \mathbb{V}$) associated with the sampling operators are defined

$$[\![x \texttt{ when } (s \texttt{ \% } n)]\!](i) = [\![x]\!](n \cdot i + s)$$

$$[\![\texttt{current}(x, (s \texttt{ \% } n))]\!](i) = \begin{cases} x_{\text{-1}} & \text{if } i < s \\ [\![x]\!]\left(\left\lfloor \frac{i-s}{n} \right\rfloor\right) & \text{otherwise,} \end{cases}$$

where $x_{\text{-1}}$ represents the initial last value for the variable x.

b) Syntax: The syntax of the language is presented in figure 2, except for the resource constraints which are detailed elsewhere [3, §§3.3 and 3.4]. It is a normalized kernel where equations *eq* are either basic expressions or function instances. Expressions e are formed from constants, variables, operators (unary, binary, conditional), `last` operators, and sampling operators. The `last` operator [8] refers to the previous or initial last value of a variable. It plays the same role as Lustre's `pre` operator but is more convenient to work with in this context. There are two forms of *sampling choice* s: the first was explained above; the second is presented in the next section. Equations and resource constraints are given within the context of a node declaration that names the node and lists its input, output, and local variables. Each variable is declared with a type, a clock type, and an initial last value, which is obligatory if the variable is used with `last` or `current`. A clock type is of the form $^1/_n$ and specifies the rate of a variable as a fraction of the rate of the enclosing node. The clock type of the node base rate ($^1/_1$) is written as 1 or elided. The clock typing rules are straightforward [3, Figure 2(b)]. For instance, here are the ones for binary operators and `when`:

$$\frac{e_1 :: {}^1/_n \qquad e_2 :: {}^1/_n}{e_1 \oplus e_2 :: {}^1/_n} \qquad \frac{x :: {}^1/_m}{x \texttt{ when } (\cdot \texttt{ \% } n) :: {}^1/_{mn}}$$

Apart from nodes, it is also possible to declare (i) external nodes by giving just an interface (name, inputs, and outputs) and a list of resource requirements, and (ii) possible resources by giving their type. Each equation may be preceded by pragmas that fix its label and phase. By default, a label is chosen automatically and a phase is fixed during scheduling.

The full program for the example is shown in figure 3a. We have declared a `cpu` resource to use as a proxy for the WCET of the external nodes and requested that scheduling balance its sum across cycles at the base rate. Other uses for resources include limiting the number of reads or writes on a data bus. The `main` node also includes a constraint requiring that the latency along the chain from s_1, through s_2, to s_3 not exceed 1 cycle at the base rate. Such constraints are respected

eq ::= $x = e \mid x^* = f\,(e^*)$

e ::= $c \mid x \mid \diamond\, e \mid e \oplus e \mid$ `if` e `then` e `else` $e \mid$ `last` x
$\mid\ x$ `when` $s \mid$ `(last` x`)` `when` $s \mid$ `current(`x`,` s`)`

s ::= `(`c `%` c`)` $\mid$ `(?` `%` c`)`

d ::= `node` f `(` $(x$ `:` ty `::` ck `[last =` c`]` `;`$)^*$ `)`
`returns (` $(x$ `:` ty `::` ck `[last =` c`]` `;`$)^*$ `)`
`var` $(x$ `:` ty `::` ck `[last =` c`]` `;`$)^*$
`let` $(($($pragmas$ eq`)` $\mid$ cst`)` `;`$)^+$ `tel`
$\mid$ `node` f `(` $(x$ `:` ty `;`$)^*$ `)` `returns` `(` $(x$ `:` ty `;`$)^*$ `)`
`requires (` $(x = c$ `;`$)^*$ `)`
$\mid$ `resource` x `:` ty

ty ::= `bool` $\mid$ `int` $\mid$ `float`

ck ::= `1 /` $n \mid$ `1`

pragmas ::= `[label(`x`)]` `[phase(`c `%` c`)]`

p ::= $(d$ `;`$)^*$

Fig. 2. Syntax (see [3] for the definition of constraints *cst*).

by scheduling but do not affect the semantics of deterministic source programs.

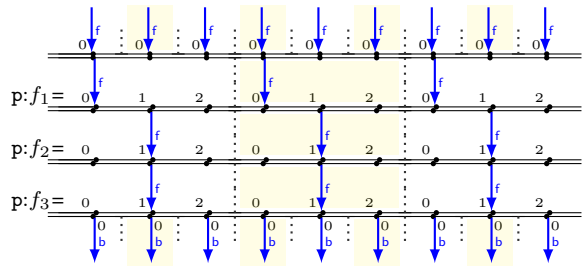
c) Code Generation: The compiler transforms a source program into sequential code in two steps. The first generates an Integer Linear Programming (ILP) problem which is solved by a tool like glpk, IBM cplex, or Gurobi Optimizer to determine the phase of each equation. Some details are given in the next section. In the second step, equations that may be executed together in a cycle are ordered based on their *concomitance*. If an equation labelled w has *forward concomitance* with one labelled r, denoted $w \xrightarrow{\text{f}} r$, the code generated for equation w is executed before the code generated for equation r. In this way, the variables defined by w are written before being read in r. With *backward concomitance*, $w \xrightarrow{\text{b}} r$, the code generated for equation r is executed first so that it reads the variables defined by w before they are updated, that is, it reads the previous values. This is useful, notably, for implementing the `last` operator.

Figure 3b shows a possible schedule for the example: f_1 is assigned to phase 0, f_2 and f_3 are assigned to phase 1, and nothing is assigned to phase 2. In this way, the latency constraint is respected and the `cpu` resource is as balanced as possible ($\sum \texttt{cpu} = [5, 4, 0]$).

Sequential code is generated [3, §4] using a variation of the standard clock-directed modular compilation scheme [9]. Figure 3c shows the result for the example. A counter `c` is introduced and incremented in each cycle modulo the hyperperiod (the least common multiple of the denominators of all rates). A `switch` statement on `c` executes the code for each equation based on its assigned phase. Note that the calculation of `s1` samples `s0` as required by the `when` in the dataflow source. The resulting value is stored in a `static` variable because it is not read until the subsequent cycle. On the other hand, the value calculated for `s2` is stored in an automatic variable since it is read directly by the call to `f3`'s step function. The value of `s3` is stored in a static variable

```
resource cpu : int

node f1(x : int) returns (y : int)
requires (cpu = 5);

node f2(x : int) returns (y : int)
requires (cpu = 2);

node f3(x : int) returns (y : int)
requires (cpu = 2);

node main(s0 : int) returns (s4 : int)
var s1, s2 : int :: 1/3;
    s3 : int :: 1/3 last = 0;
let
  s1 = f1(s0 when (0 % 3));
  s2 = f2(s1);
  s3 = f3(s2);
  s4 = current(s3, (2 % 3));

  resource balance cpu;
  latency_chain forward <= 1 (s1 -> s2 -> s3);
tel
```

(a) Source program



(b) Schedule (three iterations)

```
static int c = 0;
static int s1;
static int s3 = 0;

extern int s0, s4;
```

```
void step0() {
  int s2;
  s4 = s3;
  switch (c) {
  case 0: s1 = f1(s0);
          break;
  case 1: s2 = f2(s1);
          s3 = f3(s2);
          break;
  }
  c = (c + 1) % 3;
}
```

(c) Generated code

Fig. 3. Simple example: 3-function pipeline.

because it is read through a `current` in the dataflow source. The backward concomitance from `s3` to `s4` ensures that the latter is updated before the former.

Were the third equation changed to `s3 = f3(last s2)`, the equation defining `s2` would be backward concomitant with the one defining `s3`. The same phases could be assigned and the two commands under the `case 1` would be swapped.

d) Scheduling: The two scheduling steps must agree on the concomitance of each pair of communicating equations, otherwise the generated code could violate the dataflow semantics or resource constraints. For instance, given the schedule of figure 3b, the latency along $s_1 \xrightarrow{\mathsf{f}} s_2 \xrightarrow{\mathsf{f}} s_3$ is 1 cycle, but were code generated for $s_1 \xrightarrow{\mathsf{f}} s_2 \xrightarrow{\mathsf{b}} s_3$, then f_3 would be executed before f_2 and the latency would increase to 4.

In the original proposition [3, §3], the concomitance is fixed prior to generating the ILP constraints and maintained through to code generation. An alternative is to introduce 0-1 variables and to let the solver choose certain concomitances. For instance, $\mathtt{cc}_{w,r} = 1$ indicates $w \xrightarrow{\mathsf{f}} r$, and $\mathtt{cc}_{w,r} = 0$ indicates $w \xrightarrow{\mathsf{b}} r$.

The dependency [3, §3.2] and latency constraints [3, §3.4] must be updated to factor in the variable concomitance. The reasoning relies on the following two simple properties.

$$\forall x, y \in \mathbb{Z}, (x < y + 1) \iff (x \leq y) \tag{1}$$
$$\forall x, y \in \mathbb{Z}, (x + 1 \leq y) \iff (x < y) \tag{2}$$

For an equation of the form $r = w$, the dependency constraint is $\mathtt{p}{:}w < \mathtt{p}{:}r + \mathtt{cc}_{w,r}$. For $\mathtt{cc}_{w,r} = 1$, applying equation (1) gives $\mathtt{p}{:}w \leq \mathtt{p}{:}r$, and for $\mathtt{cc}_{w,r} = 0$, $\mathtt{p}{:}w < \mathtt{p}{:}r$. The equations can only be scheduled in the same phase when the value is propagated forward from the writer to the reader.

Similarly, for an equation $r = \mathtt{last}\ w$, the constraint is $\mathtt{p}{:}r + \mathtt{cc}_{w,r} \leq \mathtt{p}{:}w$. For $\mathtt{cc}_{w,r} = 1$, applying equation (2) gives $\mathtt{p}{:}r < \mathtt{p}{:}w$, and for $\mathtt{cc}_{w,r} = 0$, $\mathtt{p}{:}r \leq \mathtt{p}{:}w$. The equations can only be scheduled in the same phase when the reader executes before the writer.

An equation $r = w\ \mathtt{when}\ (i \,\%\, n)$ induces two bounds:

$$i \cdot \mathrm{period}(w) < \mathtt{p}{:}r - \mathtt{p}{:}w + \mathtt{cc}_{w,r} \leq (i+1) \cdot \mathrm{period}(w),$$

as does an equation $r = \mathtt{current}(w, (i \,\%\, n))$:

$$(i-1) \cdot \mathrm{period}(r) \leq \mathtt{p}{:}w - \mathtt{p}{:}r - \mathtt{cc}_{w,r} < i \cdot \mathrm{period}(r).$$

In both cases, rearranging the bounds, reasoning by cases on $\mathtt{cc}_{w,r}$, and applying equations (1) and (2) gives the expected constraints, either allowing writers and readers to be scheduled in the same phase or not depending on the dataflow semantics and the expected execution order.

Similar reasoning holds for end-to-end latency bounds, which are adapted by replacing the constraints on $\mathtt{lat}_{w,r}$ from [3, §3.4.2] with $0 < \mathtt{lat}_{w,r} + \mathtt{cc}_{w,r} \leq L$, where for forward chains $L = \mathrm{period}(r)$, and for backward chains $L = \mathrm{period}(w)$.

Rather than adding concomitance variables for every pair of communicating equations, it may only be appropriate to introduce them for pairs that participate in latency chains. The potential cost of the additional variables would then at least provide some extra flexibility in highly-constrained systems.

e) Cycles: To reject ill-founded Lustre programs, compilers construct a graph of the instantaneous static dependencies between variables and reject those that contain cycles. For a (deterministic) cyclic rate-synchronous program, the generated ILP problem will simply not have a solution. However, it may still be useful to analyze the dependency graph

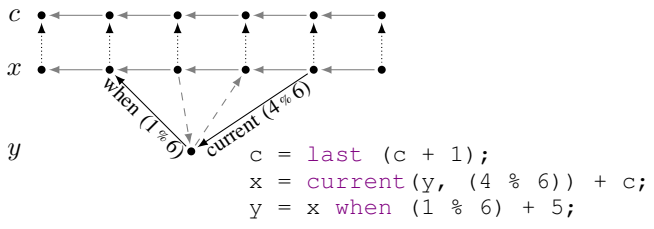


Fig. 4. Dataflow dependencies versus scheduling dependencies

to provide precise error messages. The dependency graph should be unrolled across a hyperperiod, as for the example in figure 4. This program is well defined since the dataflow graph is free of cycles, but it still cannot be scheduled because the `when` requires that y be updated after x_1 and before x_2, while the `current` requires that it be updated after x_3 and before x_4. Adding these extra dependencies to the graph (the dashed grey lines) creates a cycle.

III. Nondeterministic Sample Choices

For the flight control and guidance application, the precision and determinism of the `when` and `current` operators with sample choices of the form $(i \,\%\, n)$, is unnecessary and impracticable. How should tens-of-thousands of different choices be made and maintained? This is why the language also provides sample choices of the form $(? \,\%\, n)$, which indicate that the designer does not care which value is chosen. In practice, the generated code will simply sample the *freshest* value.

For an equation $r = w$ `when` $(? \,\%\, n)$, the associated scheduling constraint only requires that r be updated after the first w in every group of n: $0 < \mathrm{p}{:}r - \mathrm{p}{:}w + \mathtt{cc}_{w,r}$. For an equation $r = \mathtt{current}(w, (? \,\%\, n))$, the scheduling constraint only requires that w be updated before the last r in every group of n: $\mathrm{p}{:}w < (n-1) \cdot \mathrm{period}(r) + \mathrm{p}{:}r + \mathtt{cc}_{w,r}$.

After phase and concomitance values have been chosen by solving the ILP constraints, the ?'s in the source program are replaced by explicit values. Code is thus still generated from a deterministic dataflow program. The source with determinized choices may be analyzed, manipulated, or rescheduled.

A nondeterministic version of the `last` operator, sometimes called a "partial delay" [10], is also possible: `last? x` would be replaced, depending on scheduling, by `x` or `last x`. The constraints presented above distinguish the sampling choice made by `when` and `current` from the effect of delays introduced by `last` or `last?`. An alternative is to provide a family of different operators [6, §5.1].

a) Cycles: For a set of cyclic equations at a single rate, for example, `s2 = f2(s3)` and `s3 = f3(s2)`, any ILP solution must necessarily assign all equations to the same phase and the code generator will then be unable to sequence them. It may sometimes be useful to automatically break such cycles by implicitly inserting `last?`'s or by calculating a minimum feedback arc set and flipping the associated concomitances [3, §3.1.1.1]. In any case, this is a specification issue.

Another problem may occur in nondeterministic programs with inter-rate cycles. Such a cycle invariably involves at least one `when` and one `current`. Suppose that its sampling operators all contain choices of the form $(? \,\%\, n)$. If the ILP solution requires that the constituent equations sometimes execute in the same cycle, and all links have the same concomitance, then the code generator will fail even though the choices could have been resolved differently by choosing different phases or concomitances.

A tempting solution is to add an additional ILP constraint to ensure that, for any cycle, either one of its equations never executes in the same cycle as the others, or that not all of its concomitance variables have the same value. Unfortunately, this problem is equivalent to finding a minimum feedback arc set. It means either encoding a linear ordering as a graph which requires $O(n^2)$ 0-1 variables and $O(n^3)$ constraints [11], or applying techniques [12], [13] that involve enumerating elementary cycles [14] with a worst case proportional to $O(2^n)$.

A simpler alternative, used industrially, called *fast first*, requires ordering equations by decreasing rate within a cycle. Then, `current` samplings necessarily have backward concomitance, since they read a slower value, and `when` samplings necessarily have forward concomitance, since they read a faster value. Even if the equations of an inter-rate cycle execute in the same cycle, they can thus always be sequenced.

IV. Conclusion

We recalled the principles of the rate-synchronous model proposed in [3], extended it to decide concomitance during ILP scheduling, and discussed how cycles may be introduced when scheduling nondeterministic dataflow programs.

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[2] J.-L. Colaço and M. Pouzet, "Clocks as first class abstract types," in *EMSOFT*, ser. LNCS, vol. 2855, Oct. 2003, pp. 134–155.

[3] T. Bourke, V. Bregeon, and M. Pouzet, "Scheduling and compiling rate-synchronous programs with end-to-end latency constraints," in *ECRTS*, ser. LIPIcs, vol. 262, Jul. 2023, pp. 1:1–1:22.

[4] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *RTAS*. IEEE, Apr. 2010, pp. 301–310.

[5] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A real-time architecture design language for multi-rate embedded control systems," in *SAC*, Mar. 2010, pp. 527–534.

[6] G. Iooss, A. Cohen, D. Potop-Butucaru, M. Pouzet, V. Bregeon, J. Souyris, and P. Baufreton, "Polyhedral scheduling and relaxation of synchronous reactive systems," in *IMPACT*. HiPEAC, Jun. 2022.

[7] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems," in *POPL*, Jan. 2006, pp. 180–193.

[8] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Uni. Paris-Sud, Apr. 2006.

[9] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," in *LCTES*, Jun. 2008, pp. 121–130.

[10] R. Wyss, F. Boniol, J. Forget, and C. Pagetti, "A synchronous language with partial delay specification for real-time systems programming," in *APLAS*, ser. LNCS, vol. 7705, Dec. 2012, pp. 223–238.

[11] M. Grötschel, M. Jünger, and G. Reinelt, "A cutting plane algorithm for the linear ordering problem," *Operations Research*, vol. 32, no. 6, pp. 1195–1220, Nov.–Dec. 1984.

[12] A. Baharev, H. Schichl, A. Neumaier, and T. Achterberg, "An exact method for the minimum feedback arc set problem," *ACM J. Experimental Algorithmics*, vol. 26, no. 1, p. article 4, Apr. 2021.

[13] M. Grötschel, M. Jünger, and G. Reinelt, "Comments on "an exact method for the minimum feedback arc set problem"," *ACM J. Experimental Algorithmics*, vol. 27, no. 1, p. article 3, Jul. 2022.

[14] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Computing*, vol. 4, no. 1, pp. 77–84, Mar. 1975.