Static Scheduling of Latency Insensitive Designs with Lucy-n

Louis Mandel, Florence Plateau, Marc Pouzet

Static Scheduling of Latency Insensitive Designs with Lucy-n

Louis Mandel
LRI, Université Paris-Sud 11
INRIA Paris-Rocquencourt

Florence Plateau
LRI, Université Paris-Sud 11
Presently at Prove & Run

Marc Pouzet
DI, École Normale Supérieure
INRIA Paris-Rocquencourt

***Abstract*—Lucy-n is a data-flow programming language similar to Lustre extended with a buffer operator. It is based on the n-synchronous model which was initially introduced for programming multimedia streaming applications. In this article, we show that Lucy-n is also applicable to model Latency Insensitive Designs (LID). In order to model latency introduced by wires, we add a delay operator. Thanks to this new operator, a LID can be described by a Lucy-n program. Then, the Lucy-n compiler automatically provides static schedules for computation nodes and buffer sizes needed in shell wrappers.**

I. Introduction

The theory of Latency Insensitive Design ([1], [2]) was introduced to cope with the problem of long wires in Systems on Chips (SoC). Due to the length of wires, data can take more than a single clock cycle to go from one computation node (also known as Intellectual-Property or IP) to another. It raises the issues of activating IPs only when all inputs are available and storing the inputs awaiting to be processed. They are treated by encapsulating each computation node in a process, called a *shell wrapper*, that is used as a communication interface. In order to synchronize all the inputs, the shell wrappers have a buffer on each input wire. To respect the desired clock period, long wires are split into shorter segments by inserting relay nodes, called *relay stations*.

Different dynamic scheduling protocols for the shell wrappers have been proposed [2], [3]. Those protocols use back pressure mechanisms to inform the producer node that the consumer node has no more room to keep the inputs waiting to be processed. When a producer node is not executed, the shell wrapper has to inform the consumer node through a control channel that no valid inputs have been sent.

To avoid the communication overhead introduced by these dynamic protocols, static scheduling methods have been proposed [4], [5], [6], [7], [8]. Schedules are represented as ultimately periodic binary words indicating the instants where nodes have to be executed.

The schedules of [4], [5] and [6] fire the execution of the nodes As-Soon-As-Possible (ASAP). While in [7] a well-balanced schedule is sought in order to minimize buffer sizes for a fixed rate. This method is semi-automatic: prefixes of schedules must be found by hand. The approach of [8] is to search for schedules that can be shared between several IPs. The advantage of this method is that it reduces the complexity of the algorithm which finds the schedules and simplifies the circuits needed to generate them.

The approaches for finding schedules are either analytic [4], [7] or based on simulation [6], [8].

This paper presents a novel analytic way to compute static schedules for Latency Insensitive Designs by encoding LID models in the n-synchronous language Lucy-n [9], [10]. The schedules obtained by the Lucy-n compiler are not yet as good as the ones obtained by previous techniques. However, the advantage of our technique is to be able to compose modularly IPs that have already been statically scheduled. This can be useful for example when IP blocks are provided as black boxes. These Statically Scheduled IPs (SSIPs) have the particularity of not reading all their inputs nor writing all their outputs at each activation.

The paper is organized as follows. Section II presents the Lucy-n language. Section III explains how to encode LID models in this language and presents the new `delay` operator that is mandatory for the encoding. Section IV defines the typing of our new operator and presents a way of solving the typing constraints. Section V discuss the composition of SSIPs. Section VI presents experimental results. Finally, section VII concludes.

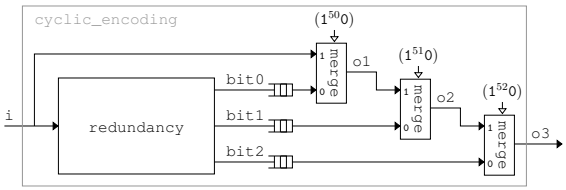
The Lucy-n compiler, the complete code of the paper examples and additional materials are available at http://www.lri.fr/~mandel/lucy-n/fmcad11.

II. A brief overview of Lucy-n

Lucy-n [10] is a programming language similar to the synchronous data-flow language Lustre [11] extended with a build-in buffer operator. The goal of this language is to relax synchrony constraints by inserting buffers without abandoning the guaranties given by synchrony, namely, determinism and execution in bounded time and memory. To this end, the compiler must compute both static schedules so that executions can be performed with finite buffer sizes and the buffer sizes themselves.

We present the language through the example of a cyclic encoder that takes as input a stream of bits and returns as output the same stream where after every 50 bits are added 3 redundancy bits. The program is given Figure 1.

The input flow `i` goes into a `redundancy` node that computes three flows of redundancy bits (line 2): `bit0`, `bit1`




```
let node cyclic_encoding i = o3 where
  rec (bit0, bit1, bit2) = redundancy i
  and o1 = merge (1^50 0) i (buffer bit0)
  and o2 = merge (1^51 0) o1 (buffer bit1)
  and o3 = merge (1^52 0) o2 (buffer bit2)
```

Fig. 1. Graphical and textual representation of a cyclic encoder in Lucy-n.

and `bit2` each producing 1 redundancy bit every 50 bits. The implementation of this node is based on a classical division circuit [12].

In parallel, the input flow `i` is merged with the first flow of redundancy bits `bit0`, following the condition $(1^{50}0)$ (line 3). It means that periodically 50 bits of `i` are read as input and transmitted as output in `o1`, then 1 bit of `bit0` is read and transmitted. Following the same principle, periodically, 51 bits of `o1` are merged with 1 bit of `bit1` to produce `o2` (line 4) and 52 bits of `o2` are merged with 1 bit of `bit2` to produce `o3` (line 5). The three flows `bit0`, `bit1` and `bit2` are bufferized such that their values are stored until the instant they are needed.

In synchronous languages, each flow is associated to a clock indicating the instants where a value is present. Clocks are infinite binary words where `1` represent the presence of a value on a flow and `0` the absence of value. Dedicated types, named clock types, specify this information. For example, the clock type of the node `redundancy` is:[1]

$$\forall\alpha.\,\alpha \texttt{ on } (1^{50}0) \rightarrow \alpha \texttt{ on } (0^{50}1) \times \alpha \texttt{ on } (0^{50}1) \times \alpha \texttt{ on } (0^{50}1)$$

Here, all the input and output types are expressed relative to a type variable α which represents the activation rhythm of the node `redundancy` and thus defines its notion of instant. The input type α `on` $(1^{50}0)$ means that whatever the base rhythm α, periodically, the input must be present for 50 instants of α, then absent for 1 instant. Each of the three outputs of the `redundancy` node emits a value on the last instant of every cycle of 51 instants.

Communication between two nodes is synchronous, *i.e.* it can be done without a buffer, if the flow is produced on the wire at the same clock that it is consumed. Equality of clocks is ensured by equality of types. So, such synchrony is guaranteed at compile time by a type system, called a clock calculus. For example, let us consider the typing rule for the `merge` operator where H is a typing environment which associates types to variables:

$$\frac{H \vdash ce : ct \quad H \vdash e_1 : ct \texttt{ on } ce \quad H \vdash e_2 : ct \texttt{ on not } ce}{H \vdash \texttt{merge } ce\ e_1\ e_2 : ct}$$

[1]In the rest of the article, we use the term *type* instead of *clock type* since data types are not considered here.

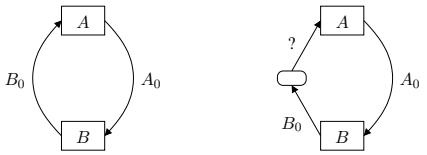


Fig. 2. First example of synchronous circuit and LID of [4].

This rule indicates that in the typing environment H, the expression `merge` $ce\ e_1\ e_2$ has type ct if: (1) the merging condition ce has type ct, (2) the expression e_1 has type ct `on` ce and (3) the expression e_2 has type ct `on not` ce. It expresses the synchronous semantics of the `merge` operator: with respect to a rhythm ct, the presence instants of the flow e_2 (*i.e.* `not` ce) are the complement of the presence instants of the flow e_1 (*i.e.* ce), and the output flow of the `merge` is present at each instant of the reference rhythm ct.

In the n-synchronous language Lucy-n, the `buffer` operator relaxes the synchronous hypothesis by introducing of a subtyping rule to the clock calculus.

$$\frac{H \vdash e : ct \qquad ct \ <: \ ct'}{H \vdash \texttt{buffer } e : ct'}$$

This typing rule means that if an expression e has type ct and if clocks represented by ct are *adaptable* to clocks represented by ct', then we can use the results of e on the type ct' provided we store them in a buffer. The adaptability relation ensures that such buffers are bounded. It is denoted $w_1 <: w_2$ where w_1 is the clock of writes into the buffer and w_2 is the clock of reads, and it is defined as the conjunction of two conditions. First, no reads may occur on an empty buffer, *i.e.* the jth reading in a buffer must always occur after the jth writing. Second, there must be a bounded number of values in the buffer, *i.e.* the difference between the number of writes and reads since the beginning of an execution must be bounded. Theses conditions can be checked at compile time provided clocks are periodic.

The clock calculus automatically infers the type of the flows. For example, the type inferred for the entire `cyclic_encoding` node is: $\forall\alpha.\alpha$ `on` $(1^{50}0^3) \rightarrow \alpha$.[2] From such types, it is possible to build a static schedule for the program and to compute the buffer sizes needed. Here, the compiler finds that the node `cyclic_encoding` can be executed without buffering `bit0` and with buffers of size one for `bit1` and `bit2`. For more information about Lucy-n and its type system refer to [13], [10], [14].

III. LID Encoding in Lucy-n

To illustrate the encoding of Latency Insensitive Designs in Lucy-n, we use the first example given in [4]. We first model the synchronous circuit of Figure 2(a), and then we model the variation of Figure 2(b) where a relay station is added to model a communication latency from B to A.

[2]This type shows that the input must be absent during the insertion of the redundancy bits into the output.

A. Encoding of Computation Nodes

In the theory of LID, each computation node or IP reads all its input and produces all its outputs at each activation. Hence, IPs are encoded as Lucy-n nodes of clock $\forall\alpha.\,\alpha \times \cdots \times \alpha \rightarrow \alpha \times \cdots \times \alpha$. Since we do not have to know the behavior of the IPs to compute the scheduling of the system, the IPs can be represented as dummy nodes with the correct clock type. For example, the IPs of Figure 2 can be modeled as:

```
let node ip_A x = x
let node ip_B x = x
```

B. Encoding of Wires

In the synchronous circuits described in [4], data takes one clock cycle to go from one IP to another. Hence, a wire cannot be represented as a Lucy-n variable, because variables in synchronous languages model instantaneous communication channels. We thus model wires with a new operator, called `delay`, that transmits its input to its output with one instant of delay.[3] With this operator, a wire from a source `src` to a destination `dst` is written:

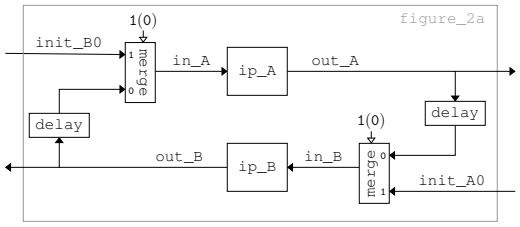
```
dst = delay src
```

To put an initial value on a wire as A_0 and B_0 in the example of Figure 2(a), we use the `merge` operator with the condition `1(0)`.

```
dst = merge 1(0) init (delay src)
```

The condition `1(0)` equals 1 at the first instant then 0 forever. So, at the first instant, the value `init` is transmitted to the destination `dst`. Then, thereafter, the values coming from the source `src` are transmitted.

Now, following the encoding of computation nodes and wires, we can program the example of Figure 2(a):




```
let node figure_2a (init_A0, init_B0) =
  (out_A, out_B) where
  rec out_A = ip_A in_A
  and out_B = ip_B in_B
  and in_A = merge 1(0) init_B0 (delay out_B)
  and in_B = merge 1(0) init_A0 (delay out_A)
```

In this node, `out_A` and `out_B` are the flows of values computed by the IPs A and B. The definitions of `in_A` and `in_B` describe the wires between A and B. The node `figure_2a` takes as inputs `init_A0` and `init_B0`, the initial values on the wires, and returns as outputs `out_A` and `out_B`, the values computed by the two IPs.

The Lucy-n compiler infers the following type for the node `figure_2a`:

$$\forall\alpha.\,\alpha \texttt{ on } 1(0) \times \alpha \texttt{ on } 1(0) \rightarrow \alpha \times \alpha$$

It means that the initial values need only be present at the first instant and that the two IPs produce some outputs at each instant. Since the presence instants of IPs outputs correspond to their activation instants, output types give the activation instants of the IPs, *i.e.* their static schedules. Here, we can verify that in the case of the synchronous circuits, all the IPs must be executed at the same instants.

C. Encoding of Shell Wrappers

In latency insensitive designs, each computation node is enclosed inside a shell wrapper that controls the activation of the node and bufferizes the inputs until this activation. To model this behavior, we need only put a buffer in front of each input and the activation condition will be automatically computed by the clock calculus. Hence, if we want to put the `ip_A` of the previous example in a shell wrapper, we only have to write:

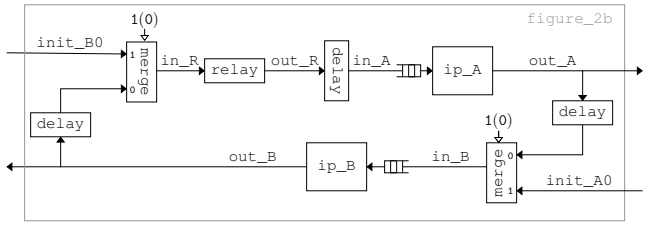
```
out_A = ip_A (buffer in_A)
```

D. Encoding of Relay Stations

A relay station is just a dummy node that splits a wire into two segments and thus introduces a delay.

```
let node relay x = x
```

Now that we have presented the encoding of computation nodes, wires, shell wrappers and relay stations, we can model the circuit of Figure 2(b): [4]




```
let node figure_2b (init_A0, init_B0) =
  (out_A, out_B) where
  rec out_A = ip_A (buffer in_A)
  and out_B = ip_B (buffer in_B)
  and out_R = relay in_R
  and in_A = delay out_R
  and in_B = merge 1(0) init_A0 (delay out_A)
  and in_R = merge 1(0) init_B0 (delay out_B)
```

The type of the node computed by the compiler is equal to $\forall\alpha.\,\alpha \texttt{ on } 1(0) \times \alpha \texttt{ on } 1(0) \rightarrow \alpha \texttt{ on } (01) \times \alpha \texttt{ on } (01)$. It means that the initial values are used only at the first instant and that the nodes A and B have to be executed every two instants from the second instant.

[3] Even if the `pre` operator of Lustre introduces a delay, it does not have exactly the same semantics as the one of `delay`. If we consider a flow `x` of clock (100), the values of `x` are output by `pre x` on the clock $000(100)$ whereas they are output by `delay x` on the clock $0(100)$. The `pre` operator outputs a value only when a new input arrives.

[4] Notice that there is no initial value on the wire between the relay station and the `ip_A` (line 6).

IV. Typing the delay operator

The typing rule for the `delay` operator is similar to the one for `buffer`. It introduces a constraint between the types of its input and output:

$$\frac{H \vdash e : ct \qquad ct' = \texttt{shiftr}\ ct}{H \vdash \texttt{delay}\ e : ct'}$$

The constraint $ct' = \texttt{shiftr}\ ct$ means that the clocks represented by ct' must be the same as the ones represented by ct delayed by one instant with respect to the activation rhythm of the node.

During typing, all the subtyping constraints introduced by the buffers and all the equality constraints introduced by the delays are collected. For example, the constraints generated by typing the node `figure_2b` are:

$$\left\{\begin{array}{lcl} \alpha_{in_A}\ \texttt{on}\ (1) & <: & \alpha_{out_A}\ \texttt{on}\ (1) \\ \alpha_{in_B}\ \texttt{on}\ (1) & <: & \alpha_{out_B}\ \texttt{on}\ (1) \\ \alpha_{in_A}\ \texttt{on}\ (1) & = & \texttt{shiftr}\ (\alpha_R\ \texttt{on}\ (1)) \\ \alpha_{in_B}\ \texttt{on}\ 0(1) & = & \texttt{shiftr}\ (\alpha_{out_A}\ \texttt{on}\ (1)) \\ \alpha_R\ \texttt{on}\ 0(1) & = & \texttt{shiftr}\ (\alpha_{out_B}\ \texttt{on}\ (1)) \end{array}\right\}$$

The collection and resolution of subtyping constraints is explained in [14]. Here, we extend this technique to cope with the constraints introduced by delays.

The resolution algorithm has the following structure:

1) Express all the constraints with respect to the same type variable. This variable represents the reference rhythm of the node. In our example, we state that $\alpha_{in_A} = \alpha\ \texttt{on}\ c_{in_A}$, $\alpha_{out_A} = \alpha\ \texttt{on}\ c_{out_A}$, $\alpha_{in_B} = \alpha\ \texttt{on}\ c_{in_B}$, $\alpha_{out_B} = \alpha\ \texttt{on}\ c_{out_B}$ and $\alpha_R = \alpha\ \texttt{on}\ c_R$ where c_{in_A}, c_{out_A}, c_{in_B}, c_{out_B} and c_R are unknown infinite binary words.
2) Since all constraints are now expressed with respect to the same type variable, simplify them into constraints over binary words. Here, we get:

$$\left\{\begin{array}{lcl} c_{in_A}\ \textit{on}\ (1) & <: & c_{out_A}\ \textit{on}\ (1) \\ c_{in_B}\ \textit{on}\ (1) & <: & c_{out_B}\ \textit{on}\ (1) \\ c_{in_A}\ \textit{on}\ (1) & = & 0(1)\ \textit{on}\ c_R\ \textit{on}\ (1) \\ c_{in_B}\ \textit{on}\ 0(1) & = & 0(1)\ \textit{on}\ c_{out_A}\ \textit{on}\ (1) \\ c_R\ \textit{on}\ 0(1) & = & 0(1)\ \textit{on}\ c_{out_B}\ \textit{on}\ (1) \end{array}\right\}$$

   and the variables c_{in_A}, c_{out_A}, c_{in_B}, c_{out_B} and c_R become the new unknowns of the system.
3) Translate constraints on words into linear constraints on integers representing the index of the 1s of the unknown words of the system.
4) Solve these constraints using standard techniques.

The main difference with the former resolution algorithm is the presence of the `shiftr` operator. It does not affect step 1 of the algorithm. You can notice that at step 2, typing constraints of the form $\alpha_x\ \texttt{on}\ p_x = \texttt{shiftr}\ (\alpha_y\ \texttt{on}\ p_x)$ have become constraints on words of the form $c_x\ \textit{on}\ p_x = 0(1)\ \textit{on}\ c_y\ \textit{on}\ p_y$, where the *on* operator is defined as follows:

$$\begin{array}{lcl} 0.w_1\ \textit{on}\ w_2 & \stackrel{def}{=} & 0.(w_1\ \textit{on}\ w_2) \\ 1.w_1\ \textit{on}\ 1.w_2 & \stackrel{def}{=} & 1.(w_1\ \textit{on}\ w_2) \\ 1.w_1\ \textit{on}\ 0.w_2 & \stackrel{def}{=} & 0.(w_1\ \textit{on}\ w_2) \end{array}$$

If we consider w_1 and w_2 as clocks, the intuitive semantics of this operator is that $w_1\ \textit{on}\ w_2$ is the clock w_2 executed on the rhythm w_1. Therefore, $0(1)\ \textit{on}\ w$ is the clock w executed from the second instant. It corresponds to the clock w shifted by one instant.

Thanks to this translation, delays generate constraints that have the same nature as constraints coming from buffers and thus, steps 3 and 4 can be done similarly as before.[5]

V. Composition of Statically Scheduled IPs

To the best of our knowledge, the only work on the composition of Statically Scheduled IPs (SSIPs) is [15], where different composition techniques are proposed depending on the nature of SSIPs involved.

For SSIPs where the same number of values are consumed and produced on all ports, they propose using dynamic scheduling protocols like those used in the dynamic scheduling of LIDs. If it is not the case, they propose encoding the composition of SSIPs with Synchronous Data-Flow graphs (SDF) [16]. This encoding is similar to the encoding of Cyclo-Static SDF graphs [17] into SDF. It raises two problems. First, the initial phases of the schedules cannot be encoded. Second, the encoding is an abstraction where some information is lost and thus correct networks with cycles may be rejected.

In the context of Lucy-n, the composition of SSIPs is classical node composition. Indeed, in Lucy-n nodes, consumption and production patterns are arbitrarily complex ultimately periodic binary words, so SSIPs are Lucy-n nodes. We think that the main strength of our method is that it treats with IPs and SSIPs uniformly.

VI. Experiment

One of the LIDs that we have encoded is shown in Figure 3. It is the MPEG-2 video encoder from [2] where the relay stations are at the same places as in [4]. This program is compiled by the Lucy-n compiler in less than 0.1 seconds and has the following type:

$$\texttt{mpeg} :: \forall\alpha.\alpha\ \texttt{on}\ (10) \rightarrow \alpha\ \texttt{on}\ 0(10)$$

The throughput of the solution computed with our algorithm is $1/2$ whereas the one of the solution computed in [4] is $2/3$. Nevertheless, the programmer can give a hint to the compiler with an option `-nbones 2` asking it to seek a solution with twice as many 1s in the periodic pattern of the schedule. With this option, the new solution is:[6]

$$\texttt{mpeg} :: \forall\alpha.\alpha\ \texttt{on}\ (110) \rightarrow \alpha\ \texttt{on}\ 00(101)$$

with a throughput of $2/3$.

[5] As $<:$ is antisymmetric, equality constraints can be translated into two inverse adaptability constraints.

[6] The compilation line is: `lucync -nbones 2 -obj r mpeg.ls`.

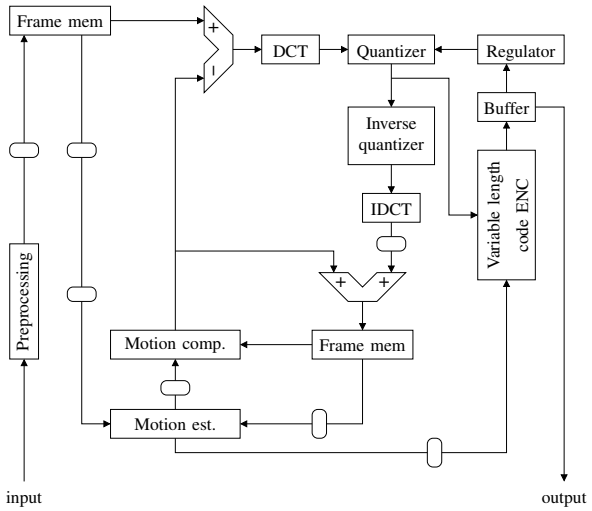


Fig. 3. MPEG-2 video encoder as found in [2], [4].

For the moment, only a few small-scale experiments have been performed, but thanks to our experiments on other Lucy-n programs [14], we think that it is possible to schedule systems with hundreds of elements.

VII. Conclusion

The contribution of this paper is the introduction of a delay operator to the language Lucy-n and a demonstration of its utility for modeling Latency Insensitive Designs. The benefit of modeling LIDs in Lucy-n is that it gives a new algorithm to automatically compute static schedules for the designs. The advantage of this method is that it allows designs that have already been scheduled to be incorporated in compositions.

We are working to improve the quality of the schedules. We can already influence the resolution algorithm by choosing the number of `1`s in the sought solution and by giving different objective functions during step 4 of the typing algorithm, for instance to privilege buffer sizes or throughput. Nevertheless, we do not yet have optimality results. Thus, we hope to adapt the results of the works cited in this article. Note, however, that our problem is more difficult because the consumption and production patterns are more complex. But, this is precisely why we can deal with IPs and SSIPs uniformly.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful remarks and Timothy Bourke for its careful reading of the paper.

References

[1] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of Latency-Insensitive Design," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[2] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with Latency in SOC Design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, 2002.

[3] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *IEEE Trans. on Computer-Aided Design*, vol. 28, no. 10, 2009.

[4] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proc. of the Design Automation Conference*, 2004.

[5] J. Boucaron, "Modélisation formelle de systèmes insensibles à la latence et ordonnancement," Ph.D. dissertation, Univ. de Nice Sophia-Antipolis, 2007.

[6] J. Boucaron, R. de Simone, and J.-V. Millo, "Formal Methods for Scheduling of Latency-Insensitive Designs," *EURASIP Journal on Embedded Systems*, vol. 2007 , Issue 1, Jan. 2007.

[7] J.-V. Millo, "Ordonnancements Périodiques dans les Réseaux de Processus : Application à la Conception Insensible aux Latences," Ph.D. dissertation, Univ. de Nice Sophia-Antipolis, 2008.

[8] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky, "Scheduling synchronous elastic designs," in *Int. Conf. on Application of Concurrency to System Design*, Jun. 2009.

[9] 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 *ACM Int. Conf. on Principles of Programming Languages*, 2006.

[10] L. Mandel, F. Plateau, and M. Pouzet, "Lucy-n: a n-synchronous extension of Lustre," in *Tenth Int. Conf. on Mathematics of Program Construction*, 2010.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[12] W. W. Peterson, *Error-Correcting Codes*. The M.I.T. Press, 1961.

[13] F. Plateau, "Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée," Ph.D. dissertation, Univ. Paris-Sud 11, 2010.

[14] L. Mandel and F. Plateau, "Typage des horloges périodiques en Lucy-n," in *Journées Francophones des Langages Applicatifs*, 2011, English extended version available at http://www.lri.fr/~mandel/lucy-n/fmcad11.

[15] J. Boucaron and J.-V. Millo, "Compositionality of Statically Scheduled IP," *ENTCS*, vol. 200, no. 1, pp. 71–87, May 2007.

[16] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE Trans. on Computers*, vol. 75, no. 9, Sep. 1987.

[17] T. M. Parks, J. L. Pino, and E. A. Lee, "A Comparison of Synchronous and Cyclo-Static Dataflow," in *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, 1995.