

# Lucy-n: a n-Synchronous Extension of Lustre <sup>★</sup>

Louis Mandel <sup>1</sup>, Florence Plateau <sup>1</sup> and Marc Pouzet <sup>1 2</sup>

<sup>1</sup> LRI, Univ. Paris-Sud 11, Orsay, France and INRIA Saclay

<sup>2</sup> Institut Universitaire de France

**Abstract.** Synchronous functional languages such as Lustre or Lucid Synchrone define a restricted class of Kahn Process Networks which can be executed with no buffer. Every expression is associated to a clock indicating the instants when a value is present. A dedicated type system, the clock calculus, checks that the actual clock of a stream equals its expected clock and thus does not need to be buffered. The n-synchrony relaxes synchrony by allowing the communication through bounded buffers whose size is computed at compile-time. It is obtained by extending the clock calculus with a subtyping rule which defines buffering points.

This paper presents the first implementation of the n-synchronous model inside a Lustre-like language called Lucy-n. The language extends Lustre with an explicit `buffer` construct whose size is automatically computed during the clock calculus. This clock calculus is defined as an inference type system and is parametrized by the clock language and the algorithm used to solve subtyping constraints. We detail here one algorithm based on the abstraction of clocks, an idea originally introduced in [5]. The paper presents a simpler, yet more precise, clock abstraction for which the main algebraic properties have been proved in Coq. Finally, we illustrate the language on various examples including a video application.

**Key words:** Process networks, Synchronous model, Type systems.

## 1 Introduction

This paper focuses on programming models and languages for implementing real time streaming applications as found in video systems. These applications transform infinite streams of pixels through successive *filters* and are thus naturally expressed as Kahn Process Networks [8]. In this model, processes execute concurrently and communicate through unbounded FIFO buffers with blocking reads when the buffer is empty and non blocking writes. The model is deterministic (a network defines a stream function) and is delay insensitive (computation and communication time do not change the network semantics). Kahn networks with bounded buffers can be implemented by adding a *back pressure* mechanism in order to avoid writes into a full buffer. Nonetheless, this may introduce artificial blocking when the size of buffers have been underestimated. The size of buffers can be increased dynamically [11] but this solution cannot be used for real time applications where execution in bounded memory must be guaranteed at compile time.

---

<sup>★</sup> This work is funded by the French “Action d’envergure” INRIA Synchronics.

To know whether a Kahn network is deadlock free or can be executed in bounded memory is undecidable in the general case [1]. *Synchronous Data Flow* (or SDF) [9] and its variants (*Cyclo Static Data Flow* [10] among others) are restricted classes of networks where every node consumes and produces a fixed number of tokens at every step. The size of buffers can be computed at compile time and a periodic static schedule can be generated. This makes SDF a good candidate for modeling and programming video intensive applications with periodic behavior [14].

Synchronous languages such as Lustre [2] or Lucid Synchrone [13] also define a restricted class of Kahn Networks [3] as they can be executed without any implicit buffering (i.e., synchronously). They are not limited to periodic behavior and ensure strong safety properties at compile-time such as determinism and absence of deadlock. Moreover, they can be compiled into statically scheduled executable code. Nonetheless, they do not offer the same flexibility as SDF-like tools do. Buffers have to be inserted manually and their size computed adequately which is both difficult and error-prone. We thus want to extend synchronous languages with conveniences to communicate through bounded buffers, like in SDF.

In synchronous languages, time is defined as the succession of discrete instants. In a data-flow framework, every stream  $s$  is associated to a boolean sequence or *clock* with value 1 at instants at which  $s$  is present and 0 otherwise. Two streams can be composed (e.g., added) without any buffer when their clocks are equal. The purpose of the *clock calculus* is to give sufficient condition for a system to be executed synchronously. This is essentially a typing problem [3,6]. Every expression is given a clock type (or simply type) and must satisfy a typing rule such as:

$$\frac{H \vdash e_1 : ck_1 \mid C_1 \quad H \vdash e_2 : ck_2 \mid C_2}{H \vdash e_1 + e_2 : ck_3 \mid \{ck_1 = ck_2 = ck_3\} \cup C_1 \cup C_2}$$

This rule states that under the typing environment  $H$ , if  $e_1$  has type  $ck_1$  under the constraints  $C_1$  and if  $e_2$  has type  $ck_2$  under the constraints  $C_2$ , then  $e_1 + e_2$  has type  $ck_3$  under the constraint that  $ck_1 = ck_2 = ck_3$  and the constraints  $C_1$  and  $C_2$ . Equality of types ensures equality of clocks. Hence, the composition of two flows of same type can be done without buffer. Synchronous languages only consider equality constraints. The n-synchrony [4] relaxes these constraints by allowing to compose streams whose type are not equal but can be synchronized through the introduction of a bounded buffer. If a stream  $x$  with type  $ck$  can be consumed later with type  $ck'$  using a bounded buffer, we shall say that  $ck$  is a subtype of  $ck'$  and we write  $ck <: ck'$ . We extend the language with a **buffer** construct which indicates the points where the subtyping rule should be applied.


$$\frac{H \vdash e : ck \mid C}{H \vdash \mathbf{buffer} e : ck' \mid \{ck <: ck'\} \cup C}$$

In terms of sequences of values, `buffer e` is equivalent to  $e$  but it may delay its input using a bounded buffer. The `buffer` construct gives more freedom to the designer while preserving an execution in bounded memory.

The purpose of the extended clock calculus is to check that bounds exist for buffer sizes and to compute them. To this aim, subtyping constraints have to be solved. In order to reduce the algorithmic complexity of constraints resolution, an abstraction of clocks has been introduced in [5]. It consists in reasoning on sets of clocks (or *envelopes*) defined by an asymptotic rate and two shifts bounding the potential delay with respect to this rate. Then, subtyping constraints can be replaced by linear constraints on those rates and shifts and solved with a tool such as Glpk. On several examples such as the *Picture in Picture* given at the end of the paper, the over-estimation due to the abstraction is small with respect to the exact solution.

**Contribution and Organization of the Paper.** This paper presents the design and implementation of an n-synchronous extension of Lustre called Lucy-n. The clock calculus is generic in the sense that it is parametrized by the clock language and the algorithm used to solve subtyping constraints. In this paper, we present an algorithm using clock abstraction. The abstraction presented in [5] has been improved in various ways: the formulae are simpler; the abstraction is more precise and no restrictions are imposed anymore on clocks when computing their abstraction. Moreover, the precision of abstract operators has been studied. Finally, the main algebraic properties and the correctness of the abstraction have been proved in Coq (1800 lines of specification and 7000 lines of proof).

The paper is organized in the following way. The language is presented in Section 2. Some algebraic properties on boolean sequences are stated in Section 3. We present the basics of the clock calculus in Section 4. We then introduce the improved version of clock abstraction in Section 5 followed by the constraint solving algorithm in Section 6. The implementation is discussed in Section 7. Finally, we illustrate the use of the language on a video application in Section 8.

All examples presented in the paper have been programmed in Lucy-n and buffer sizes have been computed automatically. The prototype is available at <http://www.lri.fr/~plateau/mpc10>. Definitions and properties that have been proved in Coq are marked with  which is a link to the corresponding code. We give a proof sketch for each property. Full proofs on paper are only available in a French document [12].

## 2 The Language

We consider a first-order synchronous dataflow language reminiscent of Lustre but extended with an explicit buffering operator. The syntax is given in Figure 1. A program ( $d$ ) is a sequence of definitions of stream functions called *nodes* and definitions of clock names ( $c$ ). The inputs of a node are described by a pattern ( $pat$ ) and its body is an expression ( $e$ ). The operators are the basic ones of Lucid Synchrones and their intuitive semantics is detailed later.  $e_1 \text{ op } e_2$  denotes

$d$	$::=$	<b>let node</b> $f$ $pat = e$	node definition
		<b>let clock</b> $c = ce$	clock definition
		$d$	sequence of definitions
$pat$	$::=$	$x$	pattern
		$(pat, \dots, pat)$	
$e$	$::=$	$i$	constant flow
		$x$	flow variable
		$(e, \dots, e)$	tuple
		$e$ $op$ $e$	imported operator
		<b>if</b> $e$ <b>then</b> $e$ <b>else</b> $e$	mux operator
		$f$ $e$	node application
		$e$ <b>where rec</b> $eqs$	local definitions
		$e$ <b>fb</b> y $e$	initialized delay
		$e$ <b>when</b> $ce$   $e$ <b>whenot</b> $ce$	sampling
		<b>merge</b> $ce$ $e$ $e$	merging
		<b>buffer</b> $e$	buffering
$eqs$	$::=$	$pat = e$   $eqs$ <b>and</b> $eqs$	mutually recursive equations

**Fig. 1.** Language kernel.

the point-wise application of a binary operator; **if**  $e_1$  **then**  $e_2$  **else**  $e_3$  is the point-wise application of a conditional;  $f$   $e$  is the application of a node  $f$  to an expression  $e$ ;  $e_1$  **fb**y  $e_2$  conses the head of  $e_1$  to  $e_2$  (and thus corresponds to an initialized delay);  $e$  **when**  $ce$  samples a stream  $e$  according to a clock expression  $ce$  whereas **merge**  $ce$   $e_1$   $e_2$  merges two streams with complementary clocks. Finally **buffer**  $e$  buffers  $e$ . We write  $e$  **where rec**  $eqs$  for an expression defined by a collection of mutually recursive equations ( $eqs$ ). In this paper, we restrict the clock language  $ce$  to define ultimately periodic boolean sequences only:

$$\begin{aligned}
 ce &::= c \mid u(v) \\
 u &::= \varepsilon \mid 0.u \mid 1.u \\
 v &::= 0 \mid 1 \mid 0.v \mid 1.v
 \end{aligned}$$

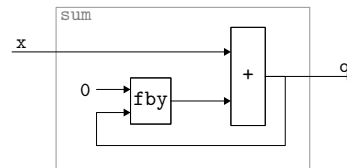
It can be a variable name ( $c$ ) or a periodic word ( $u(v)$ ) made of a finite prefix ( $u$ ) followed by the infinite repetition of a binary word ( $v$ ). For example, (10) defines the half sequence 101010...

**A First Program.** Let us write a node that sums the values taken by its input. We depict the corresponding block diagram on the right and give the clock type signature produced by the compiler.

```

let node sum x = o where
  rec o = x + (0 fby o)
val sum :: forall 'a. 'a -> 'a

```

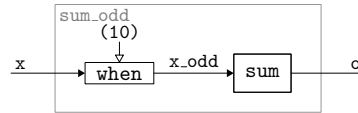


At the first instant,  $0 \text{ fby } o$  is equal to 0 then it is equal to the previous value of  $o$ . The type of `sum` means that for any clock  $c$  given to the input  $x$ , then the output of `sum x` has the same clock  $c$ . Here is an example of the execution of `sum x` on the input sequence  $x = 5, 7, 3, \dots$  set on clock (1):

flow	values	clock
$x$	5 7 3 6 2 8 ...	(1)
$0 \text{ fby } o$	0 5 12 15 21 23 ...	(1)
$o$	5 12 15 21 23 31 ...	(1)

**Sampling and Merging Streams.** We now introduce two special operators to remove and add values on a stream. The expression  $e \text{ when } ce$  returns a subsequence of  $e$  keeping the values of  $e$  at the instants where  $ce$  equals 1. For example, the sum of elements of odd index is:

```
let node sum_odd x = o where
  rec x_odd = x when (10)
  and o = sum(x_odd)
```



```
val sum_odd :: forall 'a. 'a -> 'a on (10)
```

On the input sequence of the previous example, we get:

flow	values	clock
$x$	5 7 3 6 2 8 ...	(1)
(10)	1 0 1 0 1 0 ...	(1)
$x\_odd$	5 3 2 ...	(10)
$o$	5 8 10 ...	(10)

We can observe that  $x$  is present at each instant and that  $x\_odd$  and  $o$  are defined one instant over two. Thus, the clock of  $x$  is (1) and the one of  $x\_odd$  and  $o$  is (10). The clock of  $x\_odd$  is the clock of  $x \text{ when } (10)$ , it can be computed from the clock of the flow  $x$  and the sampling condition (10). This is the result of the operation (1) `on` (10) defined below.

**Definition 1 (on Operator).** ❀

$$\begin{aligned}
 0.w_1 \text{ on } w_2 &\stackrel{def}{=} 0.(w_1 \text{ on } w_2) \\
 1.w_1 \text{ on } 1.w_2 &\stackrel{def}{=} 1.(w_1 \text{ on } w_2) \\
 1.w_1 \text{ on } 0.w_2 &\stackrel{def}{=} 0.(w_1 \text{ on } w_2)
 \end{aligned}$$

In the previous example, we have considered an input flow on the base clock (1) (true all the time) but this is not necessarily the case. Indeed,  $x$  could in particular be the result of a sampling and be on clock (110) for example. In that case, the clock of  $x\_odd$  would be (110) `on` (10) which is equal to (100) according

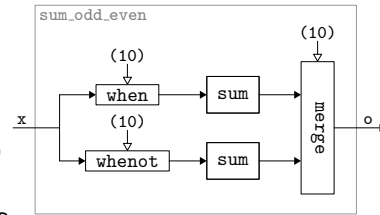
to the definition of *on*. The diagram below illustrates what happens when several sampling are composed.

flow	values	clock
<i>x</i>	5 7 3 6 ...	(110)
<i>x when</i> (10)	5            3            ...	(110) <i>on</i> (10)

The type signature for `sum_odd`, that is,  $\forall \alpha. \alpha \rightarrow \alpha \text{ on } (10)$  reflects the fact that the clock of the output stream is a sub-clock of the input stream. It states that for any clock *c*, if *x* has clock *c* then the result has clock *c on* (10). To avoid any possible confusion, we write *on* purpose *on* for the type constructor whereas *on* stands for its interpretation on boolean streams.

As opposed to sampling, the expression `merge ce e1 e2` allows to combine two streams *e1* and *e2* on complementary clocks. When *ce* is true, the output of the merge is the current value of *e1* while *e2* is not consumed; otherwise, the current value of *e2* is produced and *e1* does not progress. For example, the following node splits *x* in two subsequences, instantiates `sum` on each and finally merges them.

```
let node sum_odd_even x = o where
  rec x_odd = x when (10)
  and x_even = x whennot (10)
  and o =
    merge (10) (sum x_odd) (sum x_even)
```



```
val sum_odd_even :: forall 'a. 'a -> 'a
```

We get the following chronogram:<sup>3</sup>

flow	values	clock
<i>x</i>	5 7 3 6 2 8 ...	(1)
(10)	1 0 1 0 1 0 ...	(1)
<i>x_odd</i>	5 3 2 ...	(1) <i>on</i> (10) = (10)
<i>x_even</i>	7 6 8 ...	(1) <i>on not</i> (10) = (01)
sum <i>x_odd</i>	5 8 10 ...	(10)
sum <i>x_even</i>	7 13 21 ...	(01)
<i>o</i>	5 7 8 13 10 21 ...	(1)

**n-Synchronous Communication.** As in Lustre, communication is synchronous. As a consequence, the following program is rejected:

```
let node bad x = x + (x when (10))
```

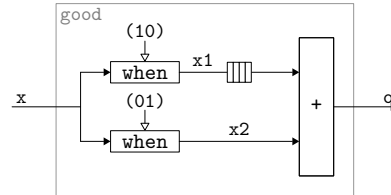
File "bad.ls", line 1, characters 17-33:  
 Cannot unify clock 'a2 on (10) with clock 'a2

<sup>3</sup> *not ce* is the point-wise application of the negation operator to *ce*.

Indeed,  $x$  and  $x$  when (10) have respectively type  $\alpha$  and  $\alpha$  on (10) whereas  $+$  expects its two arguments to have the same type.

When the `buffer` primitive is used, communication is n-synchronous. That means that it can be made synchronous through the insertion of a bounded buffer which size is computed automatically. Even using this `buffer` construct, the previous example cannot be accepted since it would need an infinite buffer to synchronize  $x$  and  $x$  when (10). Here is an example of a perfectly valid n-synchronous program.

```
let node good x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = (buffer x1) + x2
```



The compiler outputs the type and the buffer size needed:

```
val good :: forall 'a. 'a -> 'a on (01)
Buffer line 4, characters 10-20: size = 1
```

As an example, we get:

flow	values	clock
$x$	5 7 3 6 2 8 ...	(1)
$x1$	5 3 2 ...	(10)
<code>buffer(x1)</code>	5 3 2 ...	(01)
$x2$	7 6 8 ...	(01)
<code>buffer(x1) + x2</code>	12 9 10 ...	(01)

Semantically, `buffer` is the identity function, it only delays its input. The use of a buffer is accepted provided the input clock of the buffer is *adaptable* to the output clock. The next section defines the adaptability relation between clocks.

### 3 Clock Adaptability

Here is the intuition of adaptability: *a clock  $w_1$  is adaptable to clock  $w_2$  if any stream with clock  $w_1$  can be consumed with clock  $w_2$  up to the insertion of a bounded buffer.*

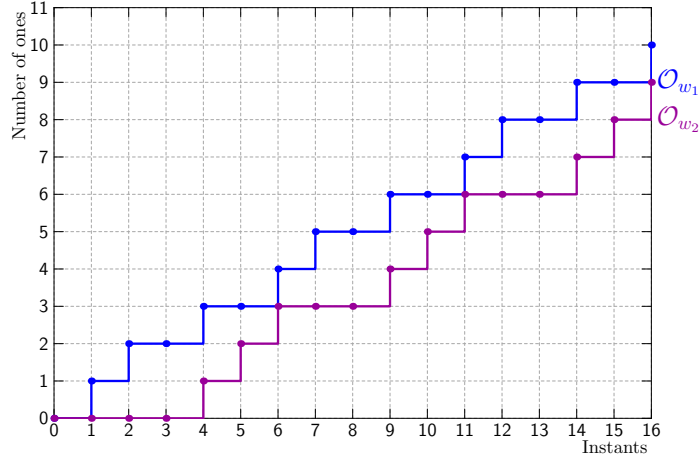
To properly define this relation, we introduce the *cumulative function* of a binary word: for any binary word  $w$ ,  $\mathcal{O}_w(i)$  counts the number of 1s up to the index  $i$ . Figure 2 shows the cumulative functions of  $w_1 = (11010)$  and  $w_2 = 0(00111)$ .

**Definition 2 (Elements and Cumulative Function of  $w$ ).** ❀

Let  $w = b.w'$  with  $b \in \{0, 1\}$ . We write  $w[i]$  for the  $i$ -th element of  $w$ :

$$w[1] \stackrel{def}{=} b$$

$$\forall i > 1, w[i] \stackrel{def}{=} w'[i - 1]$$



**Fig. 2.** Cumulative functions for  $w_1 = (11010)$  and  $w_2 = 0(00111)$ .

We write  $\mathcal{O}_w$  for the cumulative function of  $w$ :

$$\mathcal{O}_w(0) \stackrel{\text{def}}{=} 0$$

$$\forall i \geq 1, \mathcal{O}_w(i) \stackrel{\text{def}}{=} \begin{cases} \mathcal{O}_w(i-1) & \text{if } w[i] = 0 \\ \mathcal{O}_w(i-1) + 1 & \text{if } w[i] = 1 \end{cases}$$

Adaptability is the conjunction of two relations: *precedence* and *synchronizability*. Precedence ensures that there is no read in an empty buffer, that is at each instant, more values have been written than read in the buffer. Synchronizability ensures that the number of values present in the buffer during the execution is bounded.

**Definition 3 (Synchronizability  $\bowtie$ , Precedence  $\preceq$ , Adaptability  $<:$ ).**

$$w_1 \bowtie w_2 \stackrel{\text{def}}{\iff} \exists b_1, b_2 \in \mathbb{Z}, \forall i \geq 0, b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2 \quad \color{magenta}{\spadesuit}$$

$$w_1 \preceq w_2 \stackrel{\text{def}}{\iff} \forall i > 0, \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i) \quad \color{magenta}{\spadesuit}$$

$$w_1 <: w_2 \stackrel{\text{def}}{\iff} w_1 \preceq w_2 \wedge w_1 \bowtie w_2 \quad \color{magenta}{\spadesuit}$$

In Figure 2,  $w_1 \bowtie w_2$  since the vertical distance between the two curves is bounded and  $w_1 \preceq w_2$  since the curve  $\mathcal{O}_{w_1}$  is always above the one of  $\mathcal{O}_{w_2}$ .

**Buffer Size.** Consider a buffer with an input clock  $w_1$  and output clock  $w_2$ . For every instant  $i$ , the number of elements present in the buffer is:

$$size_i(w_1, w_2) = \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \quad \color{magenta}{\spadesuit}$$



A negative value means that there were more reads than writes and this case should not appear. A sufficient size for the buffer is the maximal number of values present in the buffer during the execution:

$$size(w_1, w_2) = \max_{i \geq 1} (\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)) \quad \spadesuit$$

Thus, if  $w_1$  is adaptable to  $w_2$ , a stream with clock  $w_1$  can be safely consumed on the clock  $w_2$  by insertion of a bounded buffer. Otherwise, the size of the buffer may be infinite.

**Theorem 1 (Communication Through a Buffer).**  $\spadesuit$

$$w_1 <: w_2 \Rightarrow \exists b, \forall i, 0 \leq size_i(w_1, w_2) \leq b$$

**Proof :** By definition of the synchronizability relation and the formula giving the number of elements present in the buffer at each instant, we know that there exists a value  $b$  such that  $\forall i, size_i(w_1, w_2) \leq b$ . By definition of the precedence relation, we have  $\forall i, size_i(w_1, w_2) \geq 0$ .  $\square$

As ultimately periodic words have a repetitive behavior after a certain index, checking adaptability relation and computing buffer sizes can be done statically [4].

## 4 Relaxed Clock Calculus

The purpose here is to check that programs can be evaluated using bounded buffers and to find those bounds. As seen in Section 1, it is done using typing techniques where types represents clocks. In this paper, we consider the following type language:

$$\begin{aligned} \sigma & ::= \forall \alpha_1, \dots, \alpha_n. (ck \times \dots \times ck) \rightarrow (ck \times \dots \times ck) \\ ck & ::= \alpha \mid ck \text{ on } ce \mid ck \text{ on not } ce \end{aligned}$$

A type scheme ( $\sigma$ ) represents the type of a node. It describes types of output streams with respect to types of input streams. Types of streams ( $ck$ ) can be either a type variable ( $\alpha$ ) or the type of a sampled stream ( $ck \text{ on } ce, ck \text{ on not } ce$ ).

Typing judgments are of the form:  $H \vdash e : ck \mid C$  meaning that under type environment  $H$ , the expression  $e$  has the type  $ck$  provided type constraint  $C$  are satisfied. An environment  $H$  associates type schemes, types, and clock expressions respectively to node names, stream variables and clock variables:

$$H ::= ([f_1 : \sigma_1, \dots, f_m : \sigma_m], [x_1 : ck_1, \dots, x_p : ck_p], [c_1 : ce_1, \dots, c_n : ce_n])$$

A set  $C$  of typing constraints contains equality constraints and subtyping constraints. It is of the form:

$$C ::= [ck_1 = ck'_1, \dots, ck_n = ck'_n, ck_{n+1} <: ck'_{n+1}, \dots, ck_m <: ck'_m]$$

Typing constraints are gathered progressively during typing and this raises no particular difficulty. We simply illustrate what is done on the example good given in the previous section.

**An example.** The node `good` is rewritten as follows to make the typing derivation smaller.

let node `good` `x = buffer (x when (10)) + x when (01)`

We need to use the following typing rules in addition to the ones given in Section 1:

$$\frac{H(x) = ck}{H \vdash x : ck \mid \emptyset} \quad \frac{H \vdash e : ck \mid C}{H \vdash e \text{ when } ce : ck \text{ on } ce \mid C}$$

Thereby, if we associate the type variable  $\alpha_x$  to the input of node `good`, the typing of the left and the right branches of the `+` operator gives the following two derivations:

$$A : \frac{\frac{x : \alpha_x \vdash x : \alpha_x \mid \emptyset}{x : \alpha_x \vdash x \text{ when } (10) : \alpha_x \text{ on } (10) \mid \emptyset}}{x : \alpha_x \vdash \text{buffer } (x \text{ when } (10)) : \alpha_b \mid \{\alpha_x \text{ on } (10) <: \alpha_b\}}$$

$$B : \frac{x : \alpha_x \vdash x : \alpha_x \mid \emptyset}{x : \alpha_x \vdash x \text{ when } (01) : \alpha_x \text{ on } (01) \mid \emptyset}$$

Now, using the typing rule of `+`, the body of `good` gives the following derivation:

$$\frac{A \quad B}{x : \alpha_x \vdash \text{buffer } (x \text{ when } (10)) + x \text{ when } (01) : \alpha_o \mid \{\alpha_b = \alpha_x \text{ on } (01) = \alpha_o, \alpha_x \text{ on } (10) <: \alpha_b\}}$$

The type of `good` is  $\alpha_x \rightarrow \alpha_o$  provided the following system of constraints is satisfied:

$$\{\alpha_b = \alpha_x \text{ on } (01) = \alpha_o, \alpha_x \text{ on } (10) <: \alpha_b\}$$

**Constraint solving.** There are two kinds of constraints in the system: equality and subtyping constraints. In order to solve equality constraints, synchronous languages such as Lustre or Lucid Sychrone use structural unification over clock types. It means that two types of the form  $ck_1 \text{ on } ce_1$  and  $ck_2 \text{ on } ce_2$  can be unified if and only if  $ce_1$  is equal to  $ce_2$  and if  $ck_1$  can be unified with  $ck_2$ . Here, since we use only ultimately periodic clocks, the `on` operator can be interpreted and we can use a unification algorithm such as the one presented in [4]. None of these two unification techniques is complete, and they may fail on different cases. So, to be conservative over Lustre but more expressive, it is possible to use interpreted unification only after structural simplification of the constraints. We call this technique semi-interpreted unification.

In the example, if we choose the instantiation  $\alpha_x = \alpha$ ,  $\alpha_b = \alpha \text{ on } (10)$  and  $\alpha_o = \alpha \text{ on } (10)$ , then the constraint  $\alpha_b = \alpha_x \text{ on } (01) = \alpha_o$  is always satisfied and the set of constraints reduces to  $\{\alpha \text{ on } (10) <: \alpha \text{ on } (10)\}$ .

In order to solve subtyping constraints, we must find instantiations of type variables such that constraints take the form  $\alpha \text{ on } w_1 <: \alpha \text{ on } w_2$  with  $w_1 <: w_2$ . This is simple to achieve with the example `good` since the constraint is  $\alpha \text{ on } (10) <: \alpha \text{ on } (10)$  and then, already in this form. If this is not the case, finding such a solution is computationally expensive. A first solution was experimented so as to convert adaptability constraints into a system of linear inequations. Nonetheless, the number of linear inequations is proportional to the number of 1s for each adaptability constraint. In order to overcome this complexity, we propose in this paper not to consider exact periodic clocks but their abstraction. The basic principles and algebraic properties of clock abstractions have been introduced in [5]. In this paper, we present an improved version. Moreover, its algebraic properties have been proved in Coq.

## 5 Abstraction of Binary Words

The idea behind abstraction is to reason on sets of binary words. An abstraction bounds the cumulative function of a set of words by two linear curves with the same slope. Thus, the abstraction of an infinite binary word  $w$  keeps only the asymptotic proportion  $r$  of 1s in  $w$  and two values  $b^0$  and  $b^1$  which give the minimum and maximum shift of 1s in  $w$  compared to  $r$ . This abstract information is called an *envelope* and noted  $\langle b^0, b^1 \rangle (r)$ .

**Definition 4 (Concretization).** ❀

$$\text{concr} (\langle b^0, b^1 \rangle (r)) \stackrel{\text{def}}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{l} w[i] = 1 \Rightarrow \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = 0 \Rightarrow \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

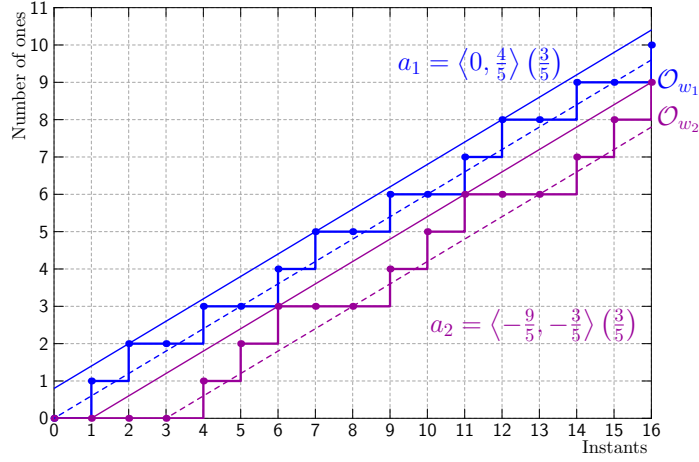
with  $b^0, b^1, r \in \mathbb{Q}$  and  $0 \leq r \leq 1$ .

The words  $w_1 = (11010)$  and  $w_2 = 0(00111)$  seen previously are respectively in envelopes  $a_1 = \langle 0, \frac{4}{5} \rangle (\frac{3}{5})$  and  $a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle (\frac{3}{5})$  shown in Figure 3. In chronograms, an abstract value  $\langle b^0, b^1 \rangle (r)$  is represented by two lines  $\Delta^1 : r \times i + b^1$  and  $\Delta^0 : r \times i + b^0$  that bound the cumulative functions of a set of binary words. The definition states that any rising edge must be below the line  $\Delta^1$  (solid line) and any absence of a rising edge must be above the line  $\Delta^0$  (dashed line).

For the set of words defined by an envelope to be non-empty, the line  $\Delta^1$  must be above the line  $\Delta^0$ . At each instant, there must be a discrete value between the two lines. It is the case if the distance between them respects the following constraint.

**Proposition 1 (Non-empty envelope).** ❀ ❀

$$\forall a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left( \frac{n}{\ell} \right), \quad \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow \text{concr} (a) \neq \emptyset$$



**Fig. 3.** Envelopes of  $w_1$  and  $w_2$ .

**Proof :** The proof is based on the use of the words  $early_a$  and  $late_a$  related to the envelope  $a = \langle b^0, b^1 \rangle (r)$  and defined by:

$$\begin{aligned}
 early_a[i] &\stackrel{def}{=} \begin{cases} 1 & \text{if } \mathcal{O}_{early_a}(i-1) + 1 \leq r \times i + b^1 \\ 0 & \text{otherwise} \end{cases} \quad \spadesuit \\
 late_a[i] &\stackrel{def}{=} \begin{cases} 0 & \text{if } \mathcal{O}_{late_a}(i-1) + 0 \geq r \times i + b^0 \\ 1 & \text{otherwise} \end{cases} \quad \spadesuit
 \end{aligned}$$

These words are such that  $concr(a) = \{w \mid early_a \preceq w \preceq late_a\}$ . They are used to prove most of the properties.

To prove that an envelope is non-empty, we have to prove that  $early_a \preceq late_a$ .  $\spadesuit$  This can be done by arithmetic manipulation once we have noticed that:

$$\begin{aligned}
 \forall i, \quad \mathcal{O}_{early_a}(i) &= \max(0, \min(i, \lfloor r \times i + b^1 \rfloor)) \quad \spadesuit \spadesuit \\
 \mathcal{O}_{late_a}(i) &= \max(0, \min(i, \lceil r \times i + b^0 \rceil)) \quad \spadesuit \spadesuit
 \end{aligned}$$

□

The abstraction of a periodic binary word can be computed automatically.

**Definition 5 (Abstraction of a Periodic Word).**

Let  $p = u(v)$  a periodic binary word. We define  $abs(p) \stackrel{def}{=} \langle b^0, b^1 \rangle (r)$  with:

$$\begin{aligned}
 r &= rate(p) = \frac{|v|_1}{|v|} \\
 b^0 &= \min_{i=1..|u|+|v| \text{ with } p[i]=0} (\mathcal{O}_p(i) - r \times i) \\
 b^1 &= \max_{i=1..|u|+|v| \text{ with } p[i]=1} (\mathcal{O}_p(i) - r \times i)
 \end{aligned}$$

where  $|u|$  is the length of  $u$  and  $|u|_1$  its number of 1s.

The asymptotic rate  $r$  corresponds to the ratio between the number of 1s in the periodic pattern and its length. To compute  $b^0$  and  $b^1$ , the word must be traversed. The shift  $b^0$  is the minimum difference when a 0 occurs between the number of 1s seen at instant  $i$  and the ideal value  $r \times i$ . The shift  $b^1$  is the maximal difference between these values when a 1 occurs.

### 5.1 Abstract Operations and Relations.

The interest of the abstraction is to reduce the complexity of exact computations and decisions on binary words by transforming them into arithmetic manipulations on rational numbers. For example, the computation of  $\mathit{on}$  on envelopes only needs three multiplications and two additions:

**Definition 6 ( $\mathit{on}^\sim$  Operator).**  $\spadesuit$  *Let  $b^0_1 \leq 0$  and  $b^0_2 \leq 0$ .<sup>4</sup> We define:*

$$\begin{aligned} \mathit{on}^\sim & \left\langle \begin{array}{cc} b^0_1 & , & b^1_1 \\ b^0_2 & , & b^1_2 \end{array} \right\rangle \left( \begin{array}{c} r_1 \\ r_2 \end{array} \right) \\ \stackrel{\text{def}}{=} & \left\langle b^0_1 \times r_2 + b^0_2, b^1_1 \times r_2 + b^1_2 \right\rangle (r_1 \times r_2) \end{aligned}$$

The elements of  $w_1 \mathit{on} w_2$  are the elements of  $w_1$  filtered by the elements of  $w_2$ . The rate of 1 in  $w_1 \mathit{on} w_2$  is thus the product of the rate of  $w_1$  and the one of  $w_2$ . When  $w_1$  is sampled by  $w_2$ , its shifts are multiplied by  $r_2$ . The shifts of  $w_2$  are added to those of  $w_1$ . Consequently, this  $\mathit{on}^\sim$  operator is correct:

**Proposition 2 (Correctness of  $\mathit{on}^\sim$ ).**  $\spadesuit$  *The following property holds:*

$$\forall w_1 \in \mathit{concr}(a_1), \forall w_2 \in \mathit{concr}(a_2), w_1 \mathit{on} w_2 \in \mathit{concr}(a_1 \mathit{on}^\sim a_2)$$

**Proof :** Based on the computations of  $\mathit{early}_{a_1} \mathit{on} \mathit{early}_{a_2}$ ,  $\mathit{late}_{a_1} \mathit{on} \mathit{late}_{a_2}$  and  $\mathit{early}_{(a_1 \mathit{on}^\sim a_2)}$ ,  $\mathit{late}_{(a_1 \mathit{on}^\sim a_2)}$ .  $\square$

The negation of binary words can also be computed on envelopes.

**Definition 7 (The Operator  $\mathit{not}^\sim$ ).**  $\spadesuit$  *The following property holds:*

$$\mathit{not}^\sim \langle b^0, b^1 \rangle (r) \stackrel{\text{def}}{=} \langle -b^1, -b^0 \rangle (1 - r)$$

**Proposition 3 (Correctness of  $\mathit{not}^\sim$ ).**  $\spadesuit$  *The following property holds:*

$$\forall w \in \mathit{concr}(a), \mathit{not} w \in \mathit{concr}(\mathit{not}^\sim a)$$

**Proof :** By definition 4 and by noticing that  $\mathcal{O}_{\mathit{not} w}(i) = i - \mathcal{O}_w(i)$ .  $\square$

Now, we formulate definitions of Section 3 in the abstract domain. A relation is satisfied on envelopes if it is satisfied on all couple of words of their respective concretization sets.

<sup>4</sup> We can always lose precision on the envelopes to satisfy this condition. More details are given in [12]. The chapter about clock abstraction will be translated in English.

**Definition 8 (Abstract Synchronizability  $\bowtie^\sim$ , Precedence  $\preceq^\sim$ , Adaptability  $<:\sim$ ).**

$$\begin{aligned}
a_1 \bowtie^\sim a_2 &\stackrel{\text{def}}{\iff} \forall w_1 \in \text{concr}(a_1), w_2 \in \text{concr}(a_2), w_1 \bowtie w_2 \quad \spadesuit \\
a_1 \preceq^\sim a_2 &\stackrel{\text{def}}{\iff} \forall w_1 \in \text{concr}(a_1), w_2 \in \text{concr}(a_2), w_1 \preceq w_2 \quad \spadesuit \\
a_1 <:\sim a_2 &\stackrel{\text{def}}{\iff} \forall w_1 \in \text{concr}(a_1), w_2 \in \text{concr}(a_2), w_1 <: w_2 \quad \spadesuit
\end{aligned}$$

These relations can be tested by arithmetic comparisons on rates and shifting.

**Proposition 4 (Synchronizability, Precedence, Adaptability Tests).**

Let the two envelopes  $a_1 = \langle b^0_1, b^1_1 \rangle(r_1)$  and  $a_2 = \langle b^0_2, b^1_2 \rangle(r_2)$ . We have:

$$\begin{aligned}
r_1 = r_2 &\iff a_1 \bowtie^\sim a_2 \quad \spadesuit \spadesuit \\
b^1_2 - b^0_1 < 1 &\implies a_1 \preceq^\sim a_2 \quad \text{if } r_1 = r_2 \quad \spadesuit \spadesuit \\
a_1 \bowtie^\sim a_2 \wedge a_1 \preceq^\sim a_2 &\iff a_1 <:\sim a_2 \quad \spadesuit \spadesuit
\end{aligned}$$

**Proof :** By the use of  $early_{a_i}$  and  $late_{a_i}$ . □

As shown in Figure 3, words in envelopes  $a_1$  and  $a_2$  navigate between their two respective lines. If the lines have the same slope, all words stay at a bounded distance from each other. They are thus synchronizable. If moreover the overlapping between the envelopes is small enough, the words of the first envelope are always above the ones of the second one. Hence, the precedence relation is satisfied. In Figure 3, the envelope  $a_1$  is adaptable to  $a_2$ .

**Definition 9 (Buffer Size).** ✦

Let  $a_1 = \langle b^0_1, b^1_1 \rangle(r_1)$  and  $a_2 = \langle b^0_2, b^1_2 \rangle(r_2)$  two envelopes such that  $a_1 <:\sim a_2$ .

$$size^\sim(a_1, a_2) = \lfloor b^1_1 - b^0_2 \rfloor$$

The size of the buffer to communicate between an element of  $a_1$  and an element of  $a_2$  is the size necessary to communicate between the earlier element of  $a_1$  and the latest element of  $a_2$ . That size can be over approximated by the distance between the upper line of  $a_1$  and the lower line of  $a_2$ . The floor function is used since a buffer has an integral number of elements.

**Proposition 5 (Correctness of  $size^\sim$ ).** ✦ The following property holds:

$$\forall w_1 \in \text{concr}(a_1), \forall w_2 \in \text{concr}(a_2), size(w_1, w_2) \leq size^\sim(a_1, a_2)$$

**Proof :** By the computation of  $size(early_{a_1}, late_{a_2})$ . □

## 5.2 Precision of Abstract Operators and Tests

We characterize here the precision of abstract operators and tests on envelopes that are in the normal form defined below.

**Definition 10 (Normal Form).**

Let  $a = \langle b^0, b^1 \rangle (r)$ . Its normal form is  $\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \rangle (\frac{n}{\ell})$  with

$$\frac{n}{\ell} = r \quad \gcd(n, \ell) = 1 \quad k^0 = \lceil b^0 \times \ell \rceil \quad k^1 = \lfloor b^1 \times \ell \rfloor$$

Intuitively, putting an envelope in normal form consists in moving the lines as close as possible without changing the concretization set. To achieve this, if  $\ell$  is the denominator of the reduced form of the rational slope,  $b^1$  is decreased to the biggest rational number with denominator  $\ell$  and  $b^0$  is increased to the smallest rational number with denominator  $\ell$ . Note that the movements are always strictly less than  $\frac{1}{\ell}$ .

If  $a$  is in normal form, the test given in proposition 1 to check whether  $a$  is empty or not is not only correct, but also complete. It means that it succeeds for all non-empty envelopes in normal form. The precedence test given in proposition 4 is also correct and complete on envelopes in normal form. Finally, as the synchronizability test is always correct and complete, the adaptability test is correct and complete on envelopes in normal form.

The most precise result for  $a_1 \text{ on } a_2$  is the smallest envelope that contains the following set of words:

$$W = \{w \mid w = w_1 \text{ on } w_2 \wedge w_1 \in \text{concr}(a_1) \wedge w_2 \in \text{concr}(a_2)\}$$

The formula that we proposed in definition 6 is not the most precise. Nonetheless, if  $a_1$  and  $a_2$  are not empty, and such that  $b^0_1 \leq 0$ ,  $b^0_2 \leq 0$ ,  $r_1 \neq 0$  and  $a_1$  is in normal form, we are able to quantify its imprecision: the line  $\Delta^1$  (resp.  $\Delta^0$ ) that we compute can be slightly above (resp. below) the most precise line, but at a distance of less than one.

Finally, the most precise correct buffer size computed on envelopes  $a_1$  and  $a_2$  is the minimal buffer size sufficient to communicate from every word of  $a_1$  to every word of  $a_2$ . The formula given in definition 9 doesn't give the most precise result, but overestimates it by at most one.

### 5.3 Comparison with previous abstractions

There are several important differences between the abstraction presented here and the one introduced in [5]. Contrary to the previous abstraction, the new one is able to consider binary words with null rates (i.e., composed by a prefix followed by the infinite repetition of 0). The new abstraction is more precise on binary words with a prefix starting by a bunch of 1s. The formula for abstraction of the operator *not* is far simpler and also treats the case of words with rate 0 or 1. The precision of the abstraction has been established. Finally, the main properties have been formally proved in Coq.

## 6 Solving subtyping Constraints

A subtyping constraint is introduced for every expression of the form `buffer e`. subtyping constraints are all gathered and solved for every node definition. For example, we have seen in Section 4 that the type of `good` is  $\alpha \rightarrow \alpha$  on (10) with the constraints system  $\{\alpha$  on (10)  $<: \alpha$  on (01) $\}$  and that the constraint is verified if and only if (10)  $<: (01)$ . To solve it, we can go into the abstract domain:

$$(10) <: (01) \iff \text{abs}((10)) <:\sim \text{abs}((01))$$

The abstractions of (10) and (01) are respectively  $\langle 0, \frac{1}{2} \rangle (\frac{1}{2})$  and  $\langle -\frac{1}{2}, 0 \rangle (\frac{1}{2})$ . Thus, by application of proposition 4, we get:

$$(10) <: (01) \iff \left( \frac{1}{2} = \frac{1}{2} \quad \wedge \quad 0 - 0 < 1 \right)$$

The subtyping constraint for the node `good` is always verified. Its type scheme is thus:  $\forall \alpha. \alpha \rightarrow \alpha$  on (10). On this example, the abstract method find the same solution as the method on ultimately periodic words.

This example was particularly simple because the constraint was on the very same variable  $\alpha$ . This is not always the case as in:

```
let node f (x, y, z) =
  buffer (x when (11010))
  + y when 0(00111)
  + buffer (z when (01))
  + buffer (z when 0(1100))
```

The type of the node is  $\alpha_1 \times \alpha_2 \times \alpha_3 \rightarrow \alpha_2$  on 0(00111) with the following constraints system  $C$ :

$$C = \left\{ \begin{array}{ll} \alpha_1 \text{ on } (11010) & <: \alpha_2 \text{ on } 0(00111) \\ \alpha_3 \text{ on } (01) & <: \alpha_2 \text{ on } 0(00111) \\ \alpha_3 \text{ on } 0(1100) & <: \alpha_2 \text{ on } 0(00111) \end{array} \right\}$$

Depending on the instantiation of type variables  $\alpha_1$ ,  $\alpha_2$  and  $\alpha_3$ , these constraints can be verified or not. Solving  $C$  consists in finding a substitution for those variables such that the constraints are always satisfied. For that, we have to express all related constraints according to the same type variable. Let  $\alpha_1 = \alpha$  on  $c_1$ ,  $\alpha_2 = \alpha$  on  $c_2$ ,  $\alpha_3 = \alpha$  on  $c_3$  where  $c_1$ ,  $c_2$  and  $c_3$  fresh unknown variables such that the following system is satisfied:

$$C \Leftrightarrow \left\{ \begin{array}{ll} c_1 \text{ on } (11010) & <: c_2 \text{ on } 0(00111) \\ c_3 \text{ on } (01) & <: c_2 \text{ on } 0(00111) \\ c_3 \text{ on } 0(1100) & <: c_2 \text{ on } 0(00111) \end{array} \right\}$$

We have translated the subtyping constraints into adaptability constraints. In order to solve them, we look for a solution in the abstract domain:

$$C \Leftrightarrow \left\{ \begin{array}{ll} \text{abs}(c_1) \text{ on}\sim \text{abs}((11010)) & <:\sim \text{abs}(c_2) \text{ on}\sim \text{abs}(0(00111)) \\ \text{abs}(c_3) \text{ on}\sim \text{abs}((01)) & <:\sim \text{abs}(c_2) \text{ on}\sim \text{abs}(0(00111)) \\ \text{abs}(c_3) \text{ on}\sim \text{abs}(0(1100)) & <:\sim \text{abs}(c_2) \text{ on}\sim \text{abs}(0(00111)) \end{array} \right\}$$



Consider the second constraint. Let  $abs(c_2) = \langle b^0_2, b^1_2 \rangle (r_2)$  and  $abs(c_3) = \langle b^0_3, b^1_3 \rangle (r_3)$ . We compute values of the abstractions  $abs(0(01)) = \langle -\frac{1}{2}, 0 \rangle (\frac{1}{2})$  and  $abs(0(00111)) = \langle -\frac{9}{5}, -\frac{3}{5} \rangle (\frac{3}{5})$ . Then, by definition of  $on^\sim$ , the constraint can be rewritten into:<sup>5</sup>

$$\langle b^0_3 \times \frac{1}{2} - \frac{1}{2}, b^1_3 \times \frac{1}{2} + 0 \rangle (r_3 \times \frac{1}{2}) <:\sim \langle b^0_2 \times \frac{3}{5} - \frac{9}{5}, b^1_2 \times \frac{3}{5} - \frac{3}{5} \rangle (r_2 \times \frac{3}{5})$$

Then, by proposition 4, it can be decomposed into a synchronizability constraint ( $r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5}$ ) and a precedence constraint ( $(b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1$ ). If we apply these transformations to the other constraints, we can transform abstract adaptability constraints into a set of synchronizability constraints:

$$\left\{ \begin{array}{l} r_1 \times \frac{3}{5} = r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5} \end{array} \right\}$$

and a set of precedence constraints:

$$\left\{ \begin{array}{l} (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_1 \times \frac{3}{5} + 0) < 1 \\ (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1 \\ (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1 \end{array} \right\}$$

In the synchronizability constraints, we look for correct rates. The  $r_i$  must be between 0 and 1 according to definition 4. To solve the system, we rewrite every constraint  $r_i \times q_i = r_j \times q_j$  into  $r_i = \frac{q_j}{q_i} \times r_j$  with  $\frac{q_j}{q_i} \leq 1$ . Then, we saturate the system by expressing all constraints in terms of a common variable. Finally, we choose the rate 1 for this variable so as to maximize the throughput of the system. In the example above, we obtain  $r_1 = \frac{5}{6}$ ,  $r_2 = \frac{5}{6}$  and  $r_3 = 1$ .

The precedence constraints allow to determine values of  $b^0_i$  and  $b^1_i$ . In order to find non-empty envelopes, we add non-vacuity constraints as defined in proposition 1. Since we have computed the  $on^\sim$  operator, we also have to add the constraints that  $b^0_i \leq 0$ . Thus, the system reduces to a set of linear inequations which can be solved using a standard tool such as Glpk [7]. We find the following solution to the abstract adaptability constraints:

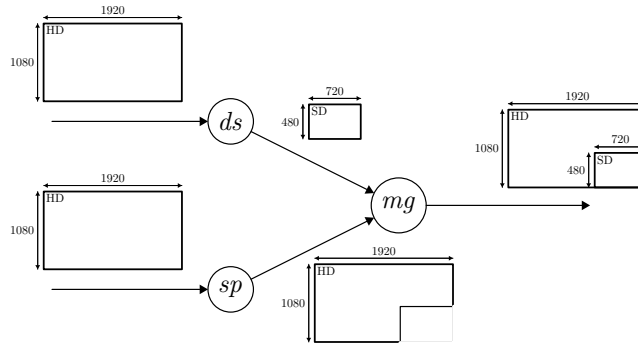
$$abs(c_1) = \langle -\frac{5}{6}, 0 \rangle (\frac{5}{6}) \quad abs(c_2) = \langle 0, \frac{10}{6} \rangle (\frac{5}{6}) \quad abs(c_3) = \langle 0, 0 \rangle (1)$$

By definition of the relation  $<:\sim$ , every word in the computed envelope is a solution of the original adaptability constraints system. Thus, we can take, for example  $c_1 = (011111)$ ,  $c_2 = (111110)$  and  $c_3 = (1)$  as a solution. Applying the substitution:  $\{\alpha_1 \leftarrow \alpha \text{ on } c_1, \alpha_2 \leftarrow \alpha \text{ on } c_2, \alpha_3 \leftarrow \alpha \text{ on } c_3\}$  in the type of  $\mathbf{f}$ , we get a correct type for any instantiation of  $\alpha$ . The final type signature for  $\mathbf{f}$  is:

$$\mathbf{f} : \forall \alpha. \alpha \text{ on } (011111) \times \alpha \text{ on } (111110) \times \alpha \rightarrow \alpha \text{ on } (111110) \text{ on } 0(00111)$$

Once the system of constraints is solved, we know input and output clocks of the buffers. Hence, we can compute their size. Concerning node  $\mathbf{f}$ , buffers on line 2, 4 and 5 are respectively of size 2, 1 and 2.

<sup>5</sup> To apply the definition, we have to suppose that  $b^0_2 \leq 0$  and  $b^0_3 \leq 0$ .



**Fig. 4.** *Picture in Picture.* The inputs are two flows of pixels representing High Definition videos. The size of the first video is reduced using a *Downscaler*. Some pixels from the second image are eliminated by sampling. Then the two images are merged.

## 7 Implementation

All the presented material has been implemented in OCaml. A distinctive feature of the implementation is to be generic: the clock calculus is a functor parametrized by the basic clock language (the one defining  $ce$ ). It can accept any clock language provided the following functions are given:

**Equality:** A function `equal` to test the equality of two clocks and a function `unify` that takes as input two clock expressions  $ce_1$  and  $ce_2$  and returns two expressions  $ce'_1$  and  $ce'_2$  such that  $ce'_1$  *on*  $ce_1 = ce'_2$  *on*  $ce_2$ .

**Adaptability:** A function `adaptable` to test the adaptability of two clocks and a function `solve` that returns an instantiation of variables that satisfies an adaptability constraints system.

**Buffer size:** A function `buffer_size` to compute the size of a buffer provided its input and output clocks.

In the Lucy-n compiler, the clock expressions, unification algorithm and constraints solving algorithm can be chosen using respectively the options `-ce`, `-unif` and `-solver`. The examples given in the present paper have been typed using the following command line:

```
lucync -ce pbw -unif semii -solver abs file.ls
```

It means that clock expressions are made of periodic binary words (`pbw`), that unification is semi-interpreted (`semii`) and that the solver of subtyping constraints uses abstraction (`abs`).

## 8 Application: the *Picture in Picture*

We illustrate the language on the example of a *Picture in Picture*. It is depicted in Figure 4 and is programmed in the following way:

```
let clock encrust_end =
  (0^(1920 * (1080 - 480)) {0^1200 1^720}^480)

let node pip_end (p1, p2) = o where
  rec small = buffer(downscaler p1)
  and big = (p2 whenot encrust_end)
  and o = merge encrust_end small big
```

The boolean sequence `encrust_end` controls the merge operation: when it is true, the small image is emitted, otherwise, the big one is emitted. The notation  $\{0^{1200} 1^{720}\}^{480}$  is a shortcut for repeating 480 times the pattern  $0^{1200}1^{720}$ . The small image is obtained by application of the node `downscaler`. It consists of an horizontal and a vertical filter. The horizontal filter applies a convolution and a sampling to reduce the size of lines from 1920 to 720. Similarly, the vertical filter applies a convolution and a sampling to reduce the size of columns from 1080 to 480. For the convolution, the vertical filter needs the pixels above and below every pixel. This means that the first output can be produced only after the consumption of one line of input (720 pixels). The signature inferred for node `downscaler` is:

```
val downscaler :: forall 'a. 'a -> 'a on hf on 0^720 (1) on vf
```

Since `downscaler` output is connected to the `merge` through a buffer, the type of node `pip_end` is  $(\alpha_6 \times \alpha_{10}) \rightarrow \alpha_{10}$  with the following constraint (as emitted by the compiler):

```
'a6 on hf on 0^720 (1) on vf <: 'a10 on encrust_end
```

It remains to find envelopes  $abs(c_6)$  and  $abs(c_{10})$  which satisfy the following constraint:

$$abs(c_6) \text{ on } \langle -720, \frac{481}{3} \rangle (\frac{1}{6}) <: \sim abs(c_{10}) \text{ on } \langle -192200, 0 \rangle (\frac{1}{6})$$

The values computed for  $abs(c_6)$  and  $abs(c_{10})$  are the envelopes  $\langle 0, 0 \rangle (1)$  and  $\langle -4315, -4315 \rangle (1)$ . As their concretization sets contain respectively the words  $(1)$  and  $0^{4315}(1)$ , the inferred type is:

```
val pip_end ::
  forall 'a. ('a * 'a on 0^4315 (1)) -> 'a on 0^4315 (1)
Buffer line 56, characters 13-34: size = 192240
```

This means that if the input image to be reduced arrives at the very first instant, the big image should arrive after a delay of 4315 cycles and the resulting image is produced after this delay which is approximately two lines of high definition images.

We saw that the node `downscaler` introduces a delay while the `merge` doesn't. The inferred type is correct but overestimates by one line the necessary delay before the production starts. This is due to the abstraction. Nonetheless, we believe it to be reasonable considering that the existing resolution method with no abstraction needs one day of computation. Finally, the size of the buffer is also satisfactory: 192 240 instead of 191 970 for the value with no abstraction, that is, less than one extra line of the small image.

## 9 Conclusion

This paper has presented the implementation of the n-synchronous model inside a Lustre-like language. It is achieved by defining an extended clock calculus and introducing a subtyping rule. The implementation is modular in the sense that it can be instantiated for any clock language  $ce$  for which  $ce_1 = ce_2$ ,  $ce_1 <: ce_2$  and  $size(ce_1, ce_2)$  are defined.

The language is conservative with respect to Lustre in the sense that if the program is synchronous and no `buffer e` construct is used, then the program is accepted by the clock calculus. Though the size of buffers is computed automatically, the places at which to insert them is still explicit.

## Acknowledgments

We specially thank the reviewers for their careful reading and numerous comments which helped us improving the paper and Stéphane Lescuyer for his essential help in the Coq formalisation and proofs.

## References

1. Buck, J.T.: Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Ph.D. thesis, EECS Department, University of California, Berkeley (1993)
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 178–188. ACM, New York, NY, USA (1987)
3. Caspi, P., Pouzet, M.: Synchronous kahn networks. In: ACM SIGPLAN International Conference on Functional Programming. pp. 226–238. ACM Press, New York, Philadelphia, Pennsylvania (1996)
4. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In: ACM International Conference on Principles of Programming Languages (POPL'06). pp. 180–193. ACM Press, New York (2006)
5. Cohen, A., Mandel, L., Plateau, F., Pouzet, M.: Abstraction of clocks in synchronous data-flow systems. In: Ramalingam, G. (ed.) APLAS. Lecture Notes in Computer Science, vol. 5356, pp. 237–254. Springer (2008)

6. Colaço, J.L., Pouzet, M.: Clocks as First Class Abstract Types. In: Third International Conference on Embedded Software (EMSOFT'03). pp. 134–155. Philadelphia, Pennsylvania, USA (Oct 2003)
7. Glpk: Gnu linear programming kit
8. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information processing. pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden (Aug 1974)
9. Lee, E., Messerschmitt, D.: Synchronous dataflow. *IEEE Trans. Comput.* 75(9), 1235–1245 (1987)
10. Parks, T.M., Pino, J.L., Lee, E.A.: A comparison of synchronous and cycle-static dataflow. In: ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set). p. 204. IEEE Computer Society, Washington, DC, USA (1995)
11. Parks, T.M.: Bounded scheduling of process networks. Ph.D. thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA (1995)
12. Plateau, F.: Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée. Ph.D. thesis, Université Paris-Sud 11 (Jan 2010)
13. Pouzet, M.: Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI (April 2006)
14. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction. pp. 179–196. Springer-Verlag, London, UK (2002)

## A Downscaler Code

```
1  (* convolutions *)
2  let node convo (c0, c1, c2) = (c0 + c1 + c2) / 3
3
4  let node convolution (p0, p1, p2) = p where
5    rec p = (r,g,b)
6      and r = convo (p0r, p1r, p2r)
7      and g = convo (p0g, p1g, p2g)
8      and b = convo (p0b, p1b, p2b)
9      and p0r, p0g, p0b = p0
10     and p1r, p1g, p1b = p1
11     and p2r, p2g, p2b = p2
12
13  (* horizontal filter *)
14  let clock hf = (10100100)
15
16  let node horizontal_filter p = o where
17    rec p0 = p fby p1
18      and p1 = p fby p2
19      and p2 = p
20      and o = (convolution (p0, p1, p2)) when hf
21
22  (* vertical sliding_window *)
23  let clock first_sd_line = 1^720 (0)
24  let clock first_line_of_img = (1^720 0^(720*1079))
25  let clock last_line_of_img = (0^(720*1079) 1^720)
26
27  let node my_fby_sd_line (p1,p2) =
28    merge first_sd_line (p1 when first_sd_line) (buffer(p2))
29
30  let node reorder p = ((p0,p1,p2)::'a) where
31    rec p0 =
32      merge first_line_of_img
33        (p1 when first_line_of_img)
34        ((my_fby_sd_line (p1, p1)) whennot first_line_of_img)
35    and p1 = buffer(p)
36    and p2 =
37      merge last_line_of_img
38        (p1 when last_line_of_img)
39        ((p whennot first_sd_line) whennot last_line_of_img)
40
41  (* vertical filter *)
42  let clock vf = (1^720 0^720 1^720 0^720 0^720 1^720 0^720 0^720 1^720)
43
44  let node vertical_filter p = o where
45    rec (p0,p1,p2) = reorder p
46    and o = (convolution (p0, p1, p2)) when vf
47
48  (* downscaler *)
49  let node downscaler p = vertical_filter (horizontal_filter p)
```