Scheduling and Buffer Sizing of n-Synchronous Systems

Typing of Ultimately Periodic Clocks in Lucy-n
Extended Version

Louis Mandel Florence Plateau [*]

Université Paris-Sud 11
INRIA Paris-Rocquencourt
{mandel,plateau}@lri.fr

Abstract

Lucy-n is a language for programming networks of processes communicating through *bounded buffers*. A dedicated type system, termed a clock calculus, automatically computes static schedules of the processes and the sizes of the buffers between them.

In this article, we present a new algorithm which solves the subtyping constraints generated by the clock calculus. The advantage of this algorithm is that it does not overestimate the buffer sizes needed. Moreover, it provides a way to favor either system throughput or buffer sizes minimization.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Data-flow languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Algorithms

Keywords Synchronous Languages, Typing, Constraints Solving

1. Introduction

The *n-synchronous model* [8] is a data-flow programming model. It describes networks of processes that are executed concurrently and that communicate through buffers of bounded size. It combines concurrency, determinism and flexible communications. These properties are especially useful for programming multimedia applications.

A language called Lucy-n [16] has been proposed to program in the n-synchronous model. It is essentially Lustre [6] extended with a buffer operator. Lucy-n provides a static analysis that infers the activation conditions of computation nodes and the related sizes of buffers. This analysis is in the tradition of the *clock calculus* of the synchronous data-flow languages. A clock calculus is a dedicated type system that ensures that a network of processes can be executed in bounded memory. The original clock calculus ensures that a network can be executed without buffering [5]. In the synchronous languages, each flow is associated to a clock that defines the instants where data is present. The clocks are infinite binary words where the occurrence of a `1` indicates the presence of a value on the flow and the occurrence of a `0` indicates the absence of value. Here is an example of a flow `x` and its clock:

`x`	`2`	`5`		`3`		`7`	`9`	`4`			`6` ...
clock(`x`)	`1`	`1`	`0`	`1`	`0`	`1`	`1`	`1`	`0`	`0`	`1` ...

The clock calculus forces each expression to satisfy a typing constraint similar to the following:

$$\frac{H \vdash e_1 : ct_1 \mid C_1 \qquad H \vdash e_2 : ct_2 \mid C_2}{H \vdash e_1 + e_2 : ct_3 \mid \{ct_1 = ct_2 = ct_3\} \cup C_1 \cup C_2}$$

This rule establishes that in the typing environment H, the expression $e_1 + e_2$ has a clock of type ct_3 if e_1 has a clock of type ct_1, e_2 a clock of type ct_2 and if the constraint $ct_1 = ct_2 = ct_3$ is satisfied.[1] Type equality ensures clock equality. Thus two processes producing flows of the same type can be composed without buffers.

The traditional clock calculus of synchronous languages only considers equality constraints on types, but adapting the clock calculus to the n-synchronous model requires the introduction of a subtyping rule for the buffer primitive. If a flow whose clock is of type ct can be used later on a clock of type ct' after having been stored in a buffer of bounded size, we say that ct is a subtype of ct', noted $ct <: ct'$:

$$\frac{H \vdash e : ct \mid C}{H \vdash \texttt{buffer}(e) : ct' \mid \{ct <: ct'\} \cup C}$$

The clock calculus of Lucy-n considers both equality constraints and subtyping constraints.

To solve such constraints, we have to be able to unify types ($ct_1 = ct_2$) and to verify the subtyping relation ($ct_1 <: ct_2$). These two operations depend very much on the clock language. Ultimately periodic binary words, *i.e.*, words that are composed of a finite prefix followed by an infinite repetition of a finite pattern, is an especially interesting and useful clock language. In this case, equality and subtyping are decidable. An algorithm to solve constraints on the types of ultimately periodic clocks is proposed in [16]. The algorithm exploits clock abstraction [9] where the exact "shape" of clocks are forgotten in favor of simpler specifications of the presence instants of the flows.

Type constraints on abstract clocks can be solved simply and efficiently. But, the loss of precise information leads to over approximations of buffer sizes. Moreover, even if a constraint system has a solution, the resolution algorithm can fail to find it because of

[*] Presently at Prove & Run.

[1] The sets C_1 and C_2 contain the constraints collected during the typing of the expressions e_1 and e_2.

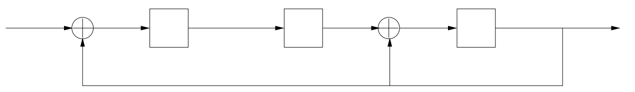


Figure 1. Circuit for division [18] by X^3+X+1. The input flow is the sequence of fifty coefficients of the polynomial to divide. After consuming the fiftieth input bit, all the coefficients of the quotient polynomial have been produced at the output and the registers contain the coefficients of the reminder polynomial.

the abstraction. Therefore, when the clocks are simple, we prefer to find buffer sizes precisely, rather than quickly.

In this article, we present an algorithm to solve the constraints without clock abstraction. This problem is difficult for two reasons. First, such an algorithm must consider all the information present in the clocks. If the prefixes and periodic patterns of the words that describe the clocks are long, there may be combinatorial explosions. Second, the handling of the initial behaviors (described by the prefixes of the words) is always delicate [2] and not always addressed [1]. Dealing with the initial and periodic behaviors simultaneously is a source of complexity but, to the best of our knowledge, there is no approach that manages to treat them in separate phases.

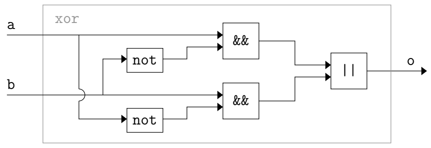
We present the Lucy-n language in Section 2 and its clock calculus in Section 3 through an example. Section 4 introduces the properties used in Section 5. Section 5 presents an algorithm for resolving constraints. Section 6 presents two variants of the resolution algorithm. The first one is a semi-decidable algorithm which is proved to find a solution if it exists. The second variant is an algorithm which has a polynomial time complexity but that can handle only a subset of systems. Section 7 discusses results obtained on examples and compares them with previous resolution algorithms. Finally, Section 8 concludes the article.

2. The Lucy-n Language

In this section, we present the Lucy-n language through the programming of a GSM voice encoder component.[2] This component is a cyclic encoder that takes as input a flow of bits representing voice samples and that produces an output flow that contains 3 new redundancy bits after every 50 data bits.

The principle of the cyclic encoder is to consider the 50 bits to encode as the coefficients of a polynomial p of degree 49. The redundancy bits are the coefficients of the reminder of the division of p by a polynomial peculiar to the encoder, here $X^3 + X + 1$. The classical circuit to divide a polynomial is shown in Figure 1; the operator $\oplus$ represents the exclusive-or and boxes represent registers initialized to `false`.

The exclusive-or operator can be programmed as follows in Lucy-n (we depict the block diagram above the corresponding code):



```
let node xor (a, b) = o where
  rec o = (a && (not b)) || (b && (not a))
val xor : (bool * bool) -> bool
val xor :: forall 'a. ('a * 'a) -> 'a
```

[2] The code of the example and the compiler are available at `http://www.lri.fr/~mandel/popl12`

The node `xor` takes as input two flows `a` and `b` and computes the value of the output flow `o`. The value of `o` is defined by the equation `o = (a && (not b)) || (b && (not a))` where the scalar operators `&&`, `||` and `not` are applied point wise to their input flows. Hence, if we apply the node `xor` to two flows `x` and `y`, we obtain a new flow `xor(x,y)`:

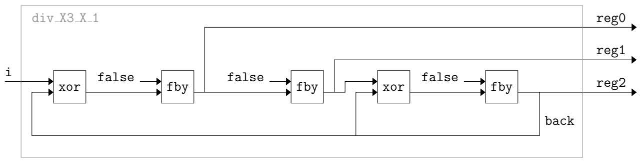
x	true	false	true	false	false	...
y	false	false	true	true	false	...
xor(x,y)	true	false	false	true	false	...

The definition of the `xor` node is followed by two facts automatically inferred by the Lucy-n compiler:

- the data type (*val xor : (bool * bool) -> bool*)
- and the clock type (*val xor :: forall 'a.('a * 'a) -> 'a*).

In the clock type, the variable *'a* represents the activation condition of the node. The type *'a * 'a -> 'a* means that at each activation, the two inputs are consumed (thus, they must be present) and the output is produced instantaneously. Since *'a* is a polymorphic variable, this type indicates that the node can be applied to any input flows that have the same clock as each other, whatever that clock is, and that it will have to be activated according instants defined by this clock.

Using this new node and the initialized register primitive of Lucy-n, `fby` (followed by), we can program the circuit of Figure 1.



```
let node div_X3_X_1 i = (reg0,reg1,reg2) where
  rec reg0 = false fby (xor(i, back))
  and reg1 = false fby (xor(reg0, back))
  and reg2 = false fby reg1
  and back = reg2
val div_X3_X_1 : bool -> (bool * bool * bool)
val div_X3_X_1 :: forall 'a. 'a -> ('a * 'a * 'a)
```

The equation `reg2 = false fby reg1` means that `reg2` is equal to `false` at the first instant and to the preceding value of `reg1` at the following instants. Note that the definitions of flows `reg0`, `reg1`, `reg2` and `back` are mutually recursive.

In order to divide a flow of polynomials, the `div_X3_X_1` node must be modified. After the arrival of the coefficients of each polynomial, that is after every 50 input bits, the three registers must be reset to `false`. Since, the content of the registers is the result of an exclusive-or between the feedback edge `back` and the preceding register (or the input flow for the first register), to reset the registers to `false`, we have to introduce three `false` values as input and three `false` values on the feedback wire, every 50 input bits.[3]

The clock type of the node `div_X3_X_1` modified accordingly is:[4]

```
val div_X3_X_1 ::
  forall 'a. 'a on (1^50 0^3) -> ('a * 'a * 'a)
```

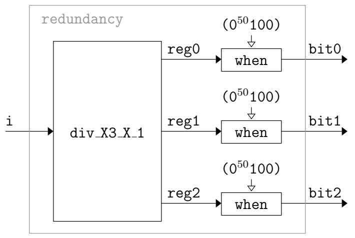
The notation *(1^50 0^3)* represents the infinite repetition of the binary word $1^{50}0^3$ where 1^{50} is the concatenation of fifty 1s and 0^3 the concatenation of three 0s. To understand the type of `div_X3_X_1`, notice that *'a* (the activation rhythm of the node)

[3] In the following, it will be implicit that all the behaviors described are repeated periodically.

[4] The source code of the modified node is available at `http://www.lri.fr/~mandel/popl12/gsm.ls.html`

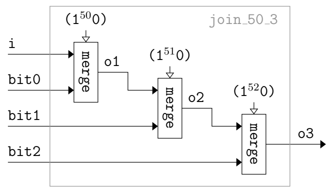
defines the notion of instants for the equations of the node. The clock type of the input flow is *'a on (1^50 0^3)*. It means that the input flow has to be present during the first 50 instants, then absent for 3 instants (during which the registers are reset). Therefore, this node can compute one division every 53 instants of the rhythm *'a*. The values of the registers are produced at each instant.

Now, to define a node `redundancy` which computes only the redundancy bits corresponding to a flow of polynomials, we sample the output of the node `div_X3_X_1`. In our implementation of the node `div_X3_X_1`, the reminder of the division is contained in the registers after the 50th input bit and output at the 51st instant. Thus, the `redundancy` node has to sample the output of `div_X3_X_1` at the 51st instant. For this, we use the `when` operator. It is parameterized by a flow and a sampling condition, and it samples the value of the flow only when the condition is equal to 1. To keep only the 51st element of a sequence of 53 bits, we use the sampling condition $(\mathtt{0}^{50}\mathtt{100})$:



```
let node redundancy i = (bit0,bit1,bit2) where
  rec (reg0,reg1,reg2) = div_X3_X_1 i
  and bit0 = reg0  when (0^50 100)
  and bit1 = reg1  when (0^50 100)
  and bit2 = reg2  when (0^50 100)
val redundancy : bool -> (bool * bool * bool)
val redundancy ::
 forall 'a. 'a on (1^50 0^3) ->
  ('a on (0^50 100) * 'a on (0^50 100) * 'a on (0^50 100))
```

To append 3 redundancy bits after 50 data bits, we use the `merge` operator. Its parameters are a merging condition and two flows; `merge` ce e_1 e_2 outputs the value of e_1 when ce is equal to 1 and the value of e_2 when ce is equal to 0. The flows e_1 and e_2 must be present on disjoint instants of the clock of ce: when ce is equal to 1, e_1 must be present and e_2 absent and vice versa when ce is equal to 0. Thus, to incorporate the first redundancy bit (`bit0`) after 50 input bits, we use the merging condition $(\mathtt{1}^{50}\mathtt{0})$. We obtain a flow of 51 bits. Then, we use the condition $(\mathtt{1}^{51}\mathtt{0})$ to incorporate the second redundancy bit, and finally the condition $(\mathtt{1}^{52}\mathtt{0})$ for the third redundancy bit.



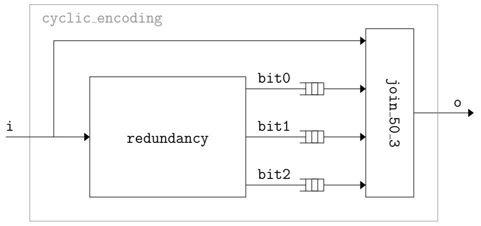
```
let node join_50_3 (i, bit0, bit1, bit2) = o3 where
  rec o1 = merge (1^50 0) i  bit0
  and o2 = merge (1^51 0) o1 bit1
  and o3 = merge (1^52 0) o2 bit2
val join_50_3 : forall 'x. ('x * 'x * 'x * 'x) -> 'x
val join_50_3 ::
  forall 'a. ('a on (1^52 0) on (1^51 0) on (1^50 0) *
             'a on (1^52 0) on (1^51 0) on not (1^50 0) *
             'a on (1^52 0) on not (1^51 0) *
             'a on not (1^52 0)) -> 'a
```

We will see in section 4 that the clock type of `join_50_3` is equivalent to: `join_50_3` :: $\forall\alpha.\ (\alpha\ \mathtt{on}\ (\mathtt{1}^{50}\mathtt{000})*$

$$
\begin{aligned}
&\alpha\ \mathtt{on}\ (\mathtt{0}^{50}\mathtt{100})*\\
&\alpha\ \mathtt{on}\ (\mathtt{0}^{50}\mathtt{010})*\\
&\alpha\ \mathtt{on}\ (\mathtt{0}^{50}\mathtt{001})) \rightarrow \alpha
\end{aligned}
$$

This type expresses that the flow containing data must be present for the first 50 instants, and then absent for the following 3 instants. The flows containing the first, second and third redundancy bit must arrive at the 51st, 52nd, and 53rd instants respectively.

To program the node that corresponds to the cyclic encoder, we must now use the node `redundancy` to compute the three redundancy bits and the node `join_50_3` to incorporate them into the input flow. But the redundancy bits are produced too early (at the instant 51) compared to when they are expected by the node `join_50_3` (instants 51, 52 and 53, successively). They must thus be stored using the `buffer` operator:



```
39 let node cyclic_encoding i = o where
40   rec (bit0, bit1, bit2) = redundancy i
41   and o = join_50_3 (i, buffer bit0,
42                         buffer bit1,
43                         buffer bit2)
```

```
val cyclic_encoding : bool -> bool
val cyclic_encoding ::
  forall 'a. 'a on (1^50 0^3) -> 'a
Buffer line 41, characters 24-35: size = 0
Buffer line 42, characters 24-35: size = 1
Buffer line 43, characters 24-35: size = 1
```

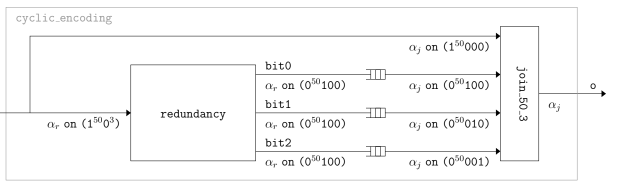
The compiler automatically computes the buffer sizes required. We can see that the buffer at line 41 is useless, the inferred size is 0. This buffer is used for the communication of the first redundancy bit (`bit0`) between the `redundancy` node and the `join_50_3` node. This bit is produced at the 51st instant and consumed immediately. The two other redundancy bits (`bit1` and `bit2`) are also produced at the 51st instant, but they are consumed later. Thus the second bit has to be stored in a buffer of size 1 for 1 instant and the third bit has to be stored in a buffer of size 1 for 2 instants.

Notice that before calculating the buffer sizes, the compiler must infer the activation rhythm of each node. When the output of one node is consumed directly by another, *i.e.*, when there is no buffer between them, the nodes must be activated such that outputs of the first node are produced at the same instants as inputs of the second node are consumed. When the output of one node is consumed by another through a buffer, the nodes must be activated such that the buffer is not read when it is empty and such that there is no infinite accumulation of data in the buffer.

3. Clock Calculus

We have seen in Section 1 that each expression in a program must satisfy a type constraint.[5] To illustrate the typing inference algorithm which collects the constraints, we return to the `cyclic_encoding` node of the previous section.

[5] Rules of the clock calculus are detailed in [19].


cyclic_encoding
i
α_i
α_r on ($1^{50}0^3$)
redundancy
bit0
α_r on ($0^{50}100$)
bit1
α_r on ($0^{50}100$)
bit2
α_r on ($0^{50}100$)
α_j on ($1^{50}000$)
α_j on ($0^{50}100$)
α_j on ($0^{50}010$)
α_j on ($0^{50}001$)
join_50_3
o
α_j

If we associate with the input `i` the clock type variable α_i, the expression `redundancy i` generates the equality constraint $\alpha_i = \alpha_r \texttt{ on } (\texttt{1}^{50}\texttt{0}^3)$. Indeed, once instantiated with a fresh variable α_r, the clock type of the node `redundancy` is $\alpha_r \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) \rightarrow (\alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}) \times \alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}) \times \alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}))$. Hence, the type of its input must be equal to $\alpha_r \texttt{ on } (\texttt{1}^{50}\texttt{0}^3)$. Consequently, the equation `(bit0, bit1, bit2) = redundancy i` adds to the typing environment that `bit0`, `bit1` and `bit2` are of type $\alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100})$.

Similarly, the application of `join_50_3` adds some constraints on the types of its inputs. Once instantiated with a fresh type variable α_j, the clock type of the node `join_50_3` is $(\alpha_j \texttt{ on } (\texttt{1}^{50}\texttt{000}) \times \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{100}) \times \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{010}) \times \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{001})) \rightarrow \alpha_j$. This type imposes the constraint that the type of the first input (here α_i, the type of the data input `i`) has to be equal to $\alpha_j \texttt{ on } (\texttt{1}^{50}\texttt{000})$ and the types of the other inputs (here $\alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100})$) must be, respectively, subtypes of $\alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{100})$, $\alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{010})$ and $\alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{001})$. For these last inputs, we do not impose type equality but rather only subtyping since they are consumed through buffers. Finally, the equation `o = join_50_3 (...)` augments the typing environment with the information that the type of `o` is α_j, the return type of `join_50_3`.

The `cyclic_encoding` node thus has the clock type $\alpha_i \rightarrow \alpha_j$, with the following constraints:

$$C = \left\{ \begin{array}{rcl} \alpha_i & = & \alpha_r \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) \\ \alpha_i & = & \alpha_j \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) \\ \alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{100}) \\ \alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{010}) \\ \alpha_r \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha_j \texttt{ on } (\texttt{0}^{50}\texttt{001}) \end{array} \right\}$$

To finish the typing of this node and to be able to compute the buffer sizes, we have to find a solution to this constraint system, that is we must find instantiations of the variables α_i, α_r and α_j such that the constraints are always satisfied. These instantiations have to be Lucy-n clock types, *i.e.*, of the shape: $ct ::= \alpha \mid ct \texttt{ on } p$ where p is an ultimately periodic binary word.

To solve the constraint system of the example, we start with the equality constraints and choose the following substitution: $\theta = \{\alpha_i \leftarrow \alpha \texttt{ on } (\texttt{1}^{50}\texttt{0}^3)\ ;\quad \alpha_r \leftarrow \alpha\ ;\quad \alpha_j \leftarrow \alpha\ ;\}$. Applying this substitution to C gives:

$$\theta(C) = \left\{ \begin{array}{rcl} \alpha \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) & = & \alpha \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) \\ \alpha \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) & = & \alpha \texttt{ on } (\texttt{1}^{50}\texttt{0}^3) \\ \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) \\ \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{010}) \\ \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{001}) \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{rcl} \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) \\ \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{010}) \\ \alpha \texttt{ on } (\texttt{0}^{50}\texttt{100}) & <: & \alpha \texttt{ on } (\texttt{0}^{50}\texttt{001}) \end{array} \right\}$$

Remark 1. *Notice that there is no complete greedy unification algorithm because there is no most general unifier for clock types. Therefore, to be complete, a resolution algorithm must take into account all the constraints globally. Since the subtyping relation is antisymmetric, a simple way to handle equality constraints is to consider them as two subtyping constraints ($ct_1 = ct_2 \Leftrightarrow (ct_1 <: ct_2) \wedge (ct_2 <: ct_1)$). In this example, greedy structural unification leads to a solution. So for the sake of conciseness, we used it.*

After transforming our constraint system to a system that contains only subtyping constraints, we notice that all the constraints depend on the same type variable. So, we apply a result from [16] to simplify the `on` operators:

$$\theta(C) \Leftrightarrow \left\{ \begin{array}{rcl} (\texttt{0}^{50}\texttt{100}) & <: & (\texttt{0}^{50}\texttt{100}) \\ (\texttt{0}^{50}\texttt{100}) & <: & (\texttt{0}^{50}\texttt{010}) \\ (\texttt{0}^{50}\texttt{100}) & <: & (\texttt{0}^{50}\texttt{001}) \end{array} \right\}$$

Sometimes however, subtyping constraints are not expressed with respect to the same type variable. For example, the program of Figure 2 generates the following set of subtyping constraints where only the second constraint can be simplified:

$$C' = \left\{ \begin{array}{rcl} \alpha_1 \texttt{ on } \texttt{10(1)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(1100)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{rcl} \alpha_1 \texttt{ on } \texttt{10(1)} & <: & \alpha_2 \texttt{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \\ \alpha_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha_3 \texttt{ on } \texttt{(1)} \end{array} \right\}$$

But, in fact, such systems can always be reduced to ones where all the constraints are expressed with respect to a single type variable. To do so, we introduce *word variables* noted c_n and we replace each type variable α_n with $\alpha \texttt{ on } c_n$. Here, the application of the substitution $\theta = \{\alpha_1 \leftarrow \alpha \texttt{ on } c_1;\ \alpha_2 \leftarrow \alpha \texttt{ on } c_2;\ \alpha_3 \leftarrow \alpha \texttt{ on } c_3;\}$ to system C' gives:

$$\theta(C') = \left\{ \begin{array}{rcl} \alpha \texttt{ on } c_1 \texttt{ on } \texttt{10(1)} & <: & \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(10)} & <: & \alpha \texttt{ on } c_3 \texttt{ on } \texttt{(1)} \\ \alpha \texttt{ on } c_2 \texttt{ on } \texttt{(01)} \texttt{ on } \texttt{(01)} & <: & \alpha \texttt{ on } c_3 \texttt{ on } \texttt{(1)} \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{rcl} c_1 \textit{ on } \texttt{10(1)} & <: & c_2 \textit{ on } \texttt{(01)} \\ \texttt{(1100)} & <: & \texttt{(01)} \\ c_2 \textit{ on } \texttt{(01)} \textit{ on } \texttt{(10)} & <: & c_3 \textit{ on } \texttt{(1)} \\ c_2 \textit{ on } \texttt{(01)} \textit{ on } \texttt{(01)} & <: & c_3 \textit{ on } \texttt{(1)} \end{array} \right\}$$

This succession of transformations reduces a system where the unknowns are types into a system where the unknowns are ultimately periodic binary words. The operator *on* and the relation $<:$ on binary words are defined in the following section. The algorithm that infers ultimately periodic binary words c_n to satisfy the $<:$ relation is presented in Section 5.

4. Algebra of Ultimately Periodic Words

In this section, we present the definitions and properties of ultimately periodic binary words that underlie the constraint resolution algorithm presented Section 5. Some details of the proofs are omitted. They can be fond chapter 4 of [19].

4.1 Ultimately Periodic Binary Words

We write w for an infinite binary word ($w ::= \texttt{0}w \mid \texttt{1}w$), u or v for finite binary words ($u, v ::= \texttt{0}u \mid \texttt{1}u \mid \varepsilon$), $|u|$ for the size of u and $|u|_\texttt{1}$ for the number of `1`s it contains. The buffer analysis relies on the instants of presence of data on the flows. Therefore, it mainly manipulates indexes of `1`s in the words:

Definition 1 (index of the jth `1` in w: $\mathcal{I}_w(j)$)**.**
Let w be a word that contains infinitely many `1`s.

$$\begin{array}{rcll} \mathcal{I}_w(1) & \overset{def}{=} & 1 & \textit{if } w = \texttt{1}w' \\ \forall j > 1,\ \mathcal{I}_w(j) & \overset{def}{=} & 1 + \mathcal{I}_{w'}(j-1) & \textit{if } w = \texttt{1}w' \\ \forall j > 0,\ \mathcal{I}_w(j) & \overset{def}{=} & 1 + \mathcal{I}_{w'}(j) & \textit{if } w = \texttt{0}w' \end{array}$$

```
let node f (i1, i2) = o where
  rec aux1 = buffer (i1 when 10(1))
             + aux2
  and aux2 = buffer (i2 when (1100))
             + i2 when (01)
  and o = buffer (aux1 when (10))
          + buffer (aux1 when (01))
```

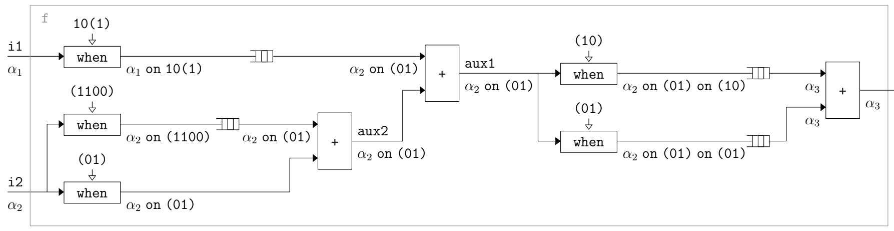


Figure 2. The node `f` and its block diagram representation. The diagram is annotated with the types obtained after the resolution of equality constraints.

For example, the index of the third `1` in $w_1 = \texttt{11010 11010}\ldots$ is 4, *i.e.*, $\mathcal{I}_{w_1}(3) = 4$.

Remark 2 (increasing indexes)**.**
The function $\mathcal{I}_w$ is increasing: $\forall j \geq 1,\ \mathcal{I}_w(j) < \mathcal{I}_w(j+1)$.

Remark 3 (sufficient indexes)**.**
As a direct consequence of Remark 2, the index of the jth `1` is greater than or equal to j: $\forall j \geq 1, \mathcal{I}_w(j) \geq j$.

A word w can also be characterized by its cumulative function which count the number of `1` since the beginning of w.

Definition 2 (cumulative function of w: $\mathcal{O}_w$)**.**[6]

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

In this article, we consider ultimately periodic clocks $u(v)$ which comprise a finite word u as prefix followed by the infinite repetition of a non-empty finite word v:

Definition 3 (ultimately periodic binary word p)**.**

$$p = u(v) \quad \stackrel{def}{\Leftrightarrow} \quad p = uw \text{ with } w = vw$$

For example, $p = \texttt{1101(110)} = \texttt{1101 110 110 110}\ \ldots$ We use the notation $p.u$ for the prefix of p (here `1101`) and $p.v$ for its periodic pattern (here `110`).

On ultimately periodic words, the index of the jth `1` can be computed using the following formulas.

Lemma 1 (index of the jth `1` in p)**.** *Let $p = u(v)$.*

$$\mathcal{I}_p(j) = \begin{cases} \mathcal{I}_u(j) & \textit{if } j \leq |u|_1 \\ |u| + x \times |v| + \mathcal{I}_v(j') & \textit{if } j = |u|_1 + x \times |v|_1 + j' \\ & \textit{with } x \in \mathbb{N},\ 1 \leq j' \leq |v|_1 \end{cases}$$

Moreover, if $j > |u|_1$ and $|v|_1 > 0$,
$\mathcal{I}_p(j) = |u| + \left\lfloor \frac{j-|u|_1-1}{|v|_1} \right\rfloor \times |v| + \mathcal{I}_v(1 + (j - |u|_1 - 1) \bmod |v|_1)$

Proof. The first equality is obtained by the application of the definition 1 and 3.
To prove the second equality, we apply the first equality until $j > |u|_1$ and $|v|_1 > 0$.
Then, if $j = |u|_1 + x \times |v|_1 + j'$ with $x \in \mathbb{N}$ and $1 \leq j' \leq |v|_1$, we have $\mathcal{I}_p(j) = |u| + x \times |v| + \mathcal{I}_v(j')$.
Let $j' = 1 + j''$ with $0 \leq j'' < |v|_1$.
Then $j = |u|_1 + x \times |v|_1 + 1 + j''$, that is $j - |u|_1 - 1 = x \times |v|_1 + j''$.
Therefore, $j'' = (j - |u|_1 - 1) \bmod |v|_1$ and $x = \left\lfloor \frac{j-|u|_1-1}{|v|_1} \right\rfloor$ and we can conclude that
$\mathcal{I}_p(j) = |u| + \left\lfloor \frac{j-|u|_1-1}{|v|_1} \right\rfloor \times |v| + \mathcal{I}_v(1 + (j - |u|_1 - 1) \bmod |v|_1)$. □

[6] The notation $w[i]$ represents the ith element of a word w.

Similarly, the index of `1`s in an ultimately periodic word can be computed as follows:

Lemma 2 (cumulative function of p)**.** *Let $p = u(v)$.*

$$\mathcal{O}_p(i) = \begin{cases} \mathcal{O}_u(i) & \textit{if } i \leq |u| \\ |u|_1 + x \times |v|_1 + \mathcal{O}_v(i') & \textit{if } i = |u| + x \times |v| + i' \\ & \textit{with } x \in \mathbb{N} \textit{ and } 0 \leq i' < |v| \end{cases}$$

Moreover, of $i \geq |u|$,
$\mathcal{O}_p(i) = |u|_1 + \left\lfloor \frac{i-|u|}{|v|} \right\rfloor \times |v|_1 + \mathcal{O}_v((i - |u|) \bmod |v|)$

Proof. The first equality is obtained by the application of the definition 2 and 3.
To prove the second equality, we apply the first equality in the case where $i \geq |u|$ with $x = \left\lfloor \frac{i-|u|}{|v|} \right\rfloor$ and $i' = (i - |u|) \bmod |v|$.
Thus we have:
$\mathcal{O}_p(i) = |u|_1 + \left\lfloor \frac{i-|u|}{|v|} \right\rfloor \times |v|_1 + \mathcal{O}_v((i - |u|) \bmod |v|)$. □

The rate of p is the ratio between the number of `1`s and the size of its periodic pattern:

Definition 4 (rate of p)**.** $rate(p) = \dfrac{|p.v|_1}{|p.v|}$

In the following, we only consider words of non-null rate, *i.e.*, such that the periodic pattern contains at least one `1` (these words have an infinite number of `1`s and are the clocks of flows that produce values infinitely).

The following two properties ensure that a periodic word is well formed.

Remark 4 (periodicity)**.** *Two successive occurrences of the same `1` of a periodic pattern are separated by a distance equal to the size of the pattern:*

$$\forall j > |p.u|_1,\ \mathcal{I}_p(j + |p.v|_1) = \mathcal{I}_p(j) + |p.v|$$

For example, if $p = \texttt{101(10010)}$, $\mathcal{I}_p(3+2) = \mathcal{I}_p(3) + 5$.

Remark 5 (sufficient size)**.** *The size of the periodic pattern of a word p ($|p.v|$) is greater than or equal to the number of elements included between the indexes of the first and last `1` of the periodic pattern of p:*

$$|p.v| \geq 1 + \mathcal{I}_p(|p.u|_1 + |p.v|_1) - \mathcal{I}_p(|p.u|_1 + 1)$$

For example, if $p = \texttt{101(10010)}$, $|p.v| \geq 1 + 7 - 4$.

An ultimately periodic binary word has an infinite number of different representations. For example, $\texttt{(10)} = \texttt{(1010)} = \texttt{1(01)} = \ldots$. But, there exists a normal form which is the representation where the prefix and the periodic pattern have the shortest size. However, for some binary operations, it is more convenient to put the two words in a form which is longer than the normal form. For example, for some operations we would prefer that the operands have the same size, or the same number of `1`s, or even that the number of `1`s in the first word is equal to the size of the second word.

Remark 6. *We can change the shape of an ultimately periodic binary word with the following manipulations:*

- *Increase prefix size: for example, we can add two elements to the prefix of the word p =* `1101(110)` *to obtain the form* `1101 11(0 11)`. *Increasing the size of the prefix can be used to increase the number of* `1`*s it contains.*
- *Repeat periodic pattern: for example, we can triple the size of the periodic pattern of p =* `1101(110)` *(and thus triple its number of* `1`*s) to obtain* `1101(110 110 110)`.

4.2 Adaptability Relation

We now define the relation $<:$ on binary words, called the *adaptability* relation. The relation $w_1 <: w_2$ holds if and only if a flow of clock w_1 can be stored in a buffer of bounded size and consumed at the rhythm of the clock w_2. It means that data does not accumulate without bound in the buffer, and that reads are not attempted when the buffer is empty. The adaptability relation is the conjunction of the *precedence* and *synchronizability* relations. The synchronizability relation between two words w_1 and w_2 (written $w_1 \bowtie w_2$) ensures that there is an upper bound on the number of values present in the buffer during an execution. It asserts that the asymptotic numbers of reads and writes from and to the buffer are equal. The precedence relation between the words w_1 and w_2 (written $w_1 \preceq w_2$) ensures the absence of reads from the buffer when it is empty. It asserts that the jth write to the buffer always occurs before the jth read.

Two words w_1 and w_2 are synchronizable if the difference between the number of occurrences of `1`s in w_1 and the number of occurrences of `1`s in w_2 is bounded.

Definition 5 (synchronizability $\bowtie$)**.**

$$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2,\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$$

To test this synchronizability relation on ultimately periodic binary words, we only have to check that the periodic patterns of the two words have the same proportion of `1`s. For example, `1(1100)` $\bowtie$ `(101001)` because $\frac{2}{4} = \frac{3}{6}$.

Proposition 1 (synchronizability test)**.**

$$p_1 \bowtie p_2 \quad \Leftrightarrow \quad rate(p_1) = rate(p_2)$$

Proof. Let $p_1 = u_1(v_1)$ and $p_2 = u_2(v_2)$.
Thanks to Remark 6, we can always rewrite p_1 as $u'_1(v'_1)$ and p_2 as $u'_2(v'_2)$ such that $|u'_1| = |u'_2| = \max(|u_1|, |u_2|)$ and $|v'_1| = |v'_2| = \mathrm{lcm}(|v_1|, |v_2|)$. Since, $\frac{|v_1|_1}{|v_1|} = \frac{|v'_1|_1}{|v'_1|}$ and $\frac{|v_2|_1}{|v_2|} = \frac{|v'_2|_1}{|v'_2|}$, we can suppose in the following that $|u_1| = |u_2|$ and $|v_1| = |v_2|$.
Thanks to the lemma 2,
for all $i_x = |u_1| + x \times |v_1|$ ($= |u_2| + x \times |v_2|$) with $x \in \mathbb{N}$,
we have:
$$\begin{aligned}\mathcal{O}_{p_1}(i_x) &= |u_1|_1 + x \times |v_1|_1\\ \mathcal{O}_{p_2}(i_x) &= |u_2|_1 + x \times |v_2|_1\end{aligned}$$

Let prove that $\frac{|v_1|_1}{|v_1|} \neq \frac{|v_2|_1}{|v_2|} \Rightarrow p_1 \not\bowtie p_2$
We have:
$\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x) = |u_2|_1 - |u_1|_1 + x \times (|v_2|_1 - |v_1|_1)$.
If $\frac{|v_1|_1}{|v_1|} < \frac{|v_2|_1}{|v_2|}$
then $|v_1|_1 < |v_2|_1$ and $\lim_{x\to+\infty}(\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x)) = +\infty$.
Therefore $\nexists b_2,\ \forall i \geq 0,\ \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$.
If $\frac{|v_1|_1}{|v_1|} > \frac{|v_2|_1}{|v_2|}$
then $|v_1|_1 > |v_2|_1$ and $\lim_{x\to+\infty}(\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_x)) = -\infty$.
Therefore $\nexists b_1,\ \forall i \geq 0,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i)$.

Let prove that $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|} \Rightarrow p_1 \bowtie p_2$
Since $\mathcal{O}$ est increasing, $\forall i$ such that $i_x \leq i \leq i_{x+1}$ with $x \in \mathbb{N}$,
$\mathcal{O}_{p_1}(i_x) \leq \mathcal{O}_{p_1}(i) \leq \mathcal{O}_{p_1}(i_{x+1})$ and
$\mathcal{O}_{p_2}(i_x) \leq \mathcal{O}_{p_2}(i) \leq \mathcal{O}_{p_2}(i_{x+1})$.
Hence $\forall i,\ i_x \leq i \leq i_{x+1}$,
$$\mathcal{O}_{p_2}(i_x) - \mathcal{O}_{p_1}(i_{x+1}) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq \mathcal{O}_{p_2}(i_{x+1}) - \mathcal{O}_{p_1}(i_x).$$
That is $\forall i,\ i_x \leq i \leq i_{x+1}$,
$$|u_2|_1 + x \times |v_2|_1 - (|u_1|_1 + (x+1) \times |v_1|_1) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1 + (x+1) \times |v_2|_1 - (|u_1|_1 + x \times |v_1|_1).$$
Therefore $\forall i,\ i_x \leq i \leq i_{x+1}$,
$$|u_2|_1 - |u_1|_1 - |v_1|_1 + x \times (|v_2|_1 - |v_1|_1) \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1 - |u_1|_1 + |v_2|_1 + x \times (|v_2|_1 - |v_1|_1).$$
If $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|}$ then $|v_1|_1 = |v_2|_1$ and $\forall i,\ i_x \leq i \leq i_{x+1}$,
$|u_2|_1 - |u_1|_1 - |v_1|_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1 - |u_1|_1 + |v_2|_1$.
Therefore, $\exists b_1, b_2,\ \forall i \geq |u_1|,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$.
In addition $\forall i \leq |u_1|,\ -|u_1|_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq |u_2|_1$.
So, we can conclude that $\exists b_1, b_2,\ \forall i \geq 1,\ b_1 \leq \mathcal{O}_{p_2}(i) - \mathcal{O}_{p_1}(i) \leq b_2$ and thus that $p_1 \bowtie p_2$. □

A word w_1 precedes a word w_2 if the jth `1` of w_1 always occurs before or at the same time as the jth `1` of w_2.

Definition 6 (precedence $\preceq$)**.**

$$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall j \geq 1,\ \mathcal{I}_{w_1}(j) \leq \mathcal{I}_{w_2}(j)$$

To check this relation on ultimately periodic words, we only have to consider this relation until the "common" periodic behavior is reached. For example, `1(1100)` $\preceq$ `(110100)` because the relation $\mathcal{I}_{\texttt{1(1100)}}(j) \preceq \mathcal{I}_{\texttt{(110100)}}(j)$ for all j such that $1 \leq j \leq 7$ and the relative behavior between the two words from the 8th `1` is exactly the same as the one from the 2nd `1`. It can be seen if we rewrite `1(1100)` as `1(110011001100)` and `(110100)` as `1(101001101001)`, the two words restart their periodic pattern simultaneously.

Proposition 2 (precedence test)**.**
Gives p_1 and p_2 such that $p_1 \bowtie p_2$.
Let $h = \max(|p_1.u|_1, |p_2.u|_1) + \mathrm{lcm}(|p_1.v|_1, |p_2.v|_1)$.

$$p_1 \preceq p_2 \quad \Leftrightarrow \quad \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)$$

The intuition for the value of the bound h is the following. By Remark 6, we can adjust the respective components of p_1 and p_2 to have the same number of `1`s. We obtain two words p'_1 and p'_2 (equivalent to p_1 and p_2) such that the number of `1`s in their prefixes is $\max(|p_1.u|_1, |p_2.u|_1)$ and the number of `1`s in their periodic pattern is $\mathrm{lcm}(|p_1.v|_1, |p_2.v|_1)$. Hence, after the traversal of $h = |p'_1.u|_1 + |p'_1.v|_1 = |p'_2.u|_1 + |p'_2.v|_1$ `1`s in p'_1 and p'_2, the periodic patterns of both words restart simultaneously. And since the two words have the same rate, we are in exactly the same situation as we were at the beginning of the first traversal of the periodic patterns. So, if the condition holds until the hth `1`, it always hold.

Proof. Let $p_1 = u_1(v_1)$ and $p_2 = u_2(v_2)$.
$\Rightarrow$) Trivial by the definition of $\preceq$.
$\Leftarrow$) Let prove that
$(\forall j,\ 1 \leq j \leq h, \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)) \Rightarrow (\forall j \geq 1, \mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j))$
with $h = \max(|u_1|_1, |u_2|_1) + \mathrm{lcm}(|v_1|_1, |v_2|_1)$.

The case where $|v_1|_1 = |v_2|_1 = 0$ is trivial. In this case, the word has a finite number of `1`s, and thus the test is executed on all the `1`s of the word.

The last case to treat is the one where $|v_1|_1 \neq 0$ and $|v_2|_1 \neq 0$ (since $p_1 \bowtie p_2$). Let $n = \max(|u_1|_1, |u_2|_1)$. By Remark 6, and since $|v_1|_1 > 0$, we can rewrite p_1 as $u'_1(v'_1)$ with $|u'_1|_1 = n$, $|v'_1| = |v_1|$ and $|v'_1|_1 = |v_1|_1$.
Let $j > n + \mathrm{lcm}(|v_1|_1, |v_2|_1)$. By Lemma 1, we have:

$$\begin{aligned}
\mathcal{I}_{p_1}(j) &= |u'_1| + \left\lfloor \tfrac{j-n-1}{|v_1|_1} \right\rfloor \times |v_1| + \mathcal{I}_{v'_1}(1 + (j-n-1) \bmod |v_1|_1) \\
&= |u'_1| + \left\lfloor \frac{\left\lfloor \frac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \operatorname{lcm}(|v_1|_1,|v_2|_1) + (j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \times |v_1| \\
&\quad + \mathcal{I}_{v'_1}(1 + (j-n-1) \bmod |v_1|_1) \\
&= |u'_1| + \left(\left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} + \left\lfloor \tfrac{(j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \right) \times |v_1| \\
&\quad + \mathcal{I}_{v'_1}(1 + (j-n-1) \bmod |v_1|_1 + 1) \\
&= |u'_1| + \left\lfloor \tfrac{(j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \right\rfloor \times |v_1| \\
&\quad + \mathcal{I}_{v'_1}(1 + (j-n-1) \bmod |v_1|_1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1| \\
&= \mathcal{I}_{p_1}(((j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)) + n + 1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1|
\end{aligned}$$

In the same way, we rewrite p_2 as $u'_2(v'_2)$ with $|u'_2|_1 = n$, $|v'_2| = |v_2|$ and $|v'_2|_1 = |v_2|_1$.
Let $j > n + \operatorname{lcm}(|v_1|_1, |v_2|_1)$.

$$\begin{aligned}
\mathcal{I}_{p_2}(j) &= \mathcal{I}_{p_2}(((j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)) + n + 1) \\
&\quad + \left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_2|_1} \times |v_2|
\end{aligned}$$

Since $((j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)) + n + 1 \leq n + \operatorname{lcm}(|v_1|_1,|v_2|_1)$,
by hypothesis we have:

$$\mathcal{I}_{p_1}(((j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)) + n + 1) \leq \mathcal{I}_{p_2}(((j-n-1) \bmod \operatorname{lcm}(|v_1|_1,|v_2|_1)) + n + 1).$$

Since $p_1 \bowtie p_2$, by the proposition 1, we have

$$\left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_1|_1} \times |v_1| = \left\lfloor \tfrac{j-n-1}{\operatorname{lcm}(|v_1|_1,|v_2|_1)} \right\rfloor \times \tfrac{\operatorname{lcm}(|v_1|_1,|v_2|_1)}{|v_2|_1} \times |v_2|.$$

Therefore, we can conclude that $\forall j > n + \operatorname{lcm}(|v_1|_1, |v_2|_1)$, $\mathcal{I}_{p_1}(j) \leq \mathcal{I}_{p_2}(j)$. □

The *adaptability* relation is the conjunction of the synchronizability and precedence relations.

Definition 7 (adaptability test). $p_1 <: p_2 \Leftrightarrow p_1 \bowtie p_2 \wedge p_1 \preceq p_2$

4.3 Buffer Size

To compute the size of a buffer, we must know the number of values that are written and read during an execution. More precisely, to compute the number of values in a buffer, we consider the cumulative functions of its input and output clocks.

Consider a buffer that takes as input a flow with clock w_1, and gives as output the same flow but with clock w_2. The number of elements present at each instant i in the buffer is the difference between the number of values that have been written into it ($\mathcal{O}_{w_1}(i)$) and the number of values that have been read from it ($\mathcal{O}_{w_2}(i)$). The necessary and sufficient buffer size is the maximum number of values present in the buffer during the execution.

Definition 8 (buffer size)**.**

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

To compute this size on ultimately periodic binary words, we need only to consider the initial patterns of the two words before their "common" periodic behavior is reached.

Proposition 3 (buffer size on ultimately periodic words)**.**
Gives p_1 and p_2 such that $p_1 <: p_2$.
Let $H = \max(|p_1.u|, |p_2.u|) + \operatorname{lcm}(|p_1.v|, |p_2.v|)$.

$$size(p_1, p_2) = \max_{1 \leq i \leq H}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$$

The proof of this property is based on the fact that there is only a finite number of values for the difference between $\mathcal{O}_{p_1}(i)$ and $\mathcal{O}_{p_2}(i)$.

Lemma 3. *Let p_1 and p_2 such that $p_1 \bowtie p_2$.*
Let $H = \max(|p_1.u|, |p_2.u|) + \operatorname{lcm}(|p_1.v|, |p_2.v|)$.
$\forall i \geq H,\ \exists i' < H,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$

Proof. Let $p_1 = u_1(v_1)$ and $p_2 = u_2(v_2)$.
$\forall i \geq |u_1|,\ \mathcal{O}_{p_1}(i + x \times |v_1|) = \mathcal{O}_{p_1}(i) + x \times |v_1|_1$
and $\forall i \geq |u_2|,\ \mathcal{O}_{p_2}(i + x \times |v_2|) = \mathcal{O}_{p_2}(i) + x \times |v_2|_1$.
Hence $\forall i \geq \max(|u_1|, |u_2|)$,
$\mathcal{O}_{p_1}(i + x \times \operatorname{lcm}(|v_1|, |v_2|)) = \mathcal{O}_{p_1}(i) + x \times \frac{\operatorname{lcm}(|v_1|,|v_2|)}{|v_1|} \times |v_1|_1$
and
$\mathcal{O}_{p_2}(i + x \times \operatorname{lcm}(|v_1|, |v_2|)) = \mathcal{O}_{p_2}(i) + x \times \frac{\operatorname{lcm}(|v_1|,|v_2|)}{|v_2|} \times |v_2|_1$.
Since $p_1 \bowtie p_2$, we have $\frac{|v_1|_1}{|v_1|} = \frac{|v_2|_1}{|v_2|}$.
Therefore, $\forall i \geq \max(|u_1|, |u_2|)$,

$$\mathcal{O}_{p_1}(i + x \times \operatorname{lcm}(|v_1|, |v_2|)) - \mathcal{O}_{p_2}(i + x \times \operatorname{lcm}(|v_1|, |v_2|)) = \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i).$$

Thus $\forall i \geq \max(|u_1|, |u_2|) + \operatorname{lcm}(|p_1|, |p_2|)$,
$\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$ with
$i' = \max(|u_1|, |u_2|) + (i - \max(|u_1|, |u_2|)) \bmod (\operatorname{lcm}(|p_1|, |p_2|))$.
Hence $i' < \max(|u_1|, |u_2|) + \operatorname{lcm}(|p_1|, |p_2|)$. □

Proof of proposition 3. Let $p_1 = u_1(v_1)$ and $p_2 = u_2(v_2)$.
Let $t = \max_{1 \leq i < \max(|u_1|,|u_2|) + \operatorname{lcm}(|v_1|,|v_2|)}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i))$.
Let prove that $\max_{i \in \mathbb{N}^*}(\mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i)) = t$,
By Lemma 3, we have:
$\forall i \geq 1,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) = \mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i')$
with $i' < \max(|u_1|, |u_2|) + \operatorname{lcm}(|v_1|, |v_2|)$.
By hypothesis, $\mathcal{O}_{p_1}(i') - \mathcal{O}_{p_2}(i') \leq t$.
Thus $\forall i \geq 1,\ \mathcal{O}_{p_1}(i) - \mathcal{O}_{p_2}(i) \leq t$. □

Note that the bound H is not the same as the one of Proposition 2, because here we iterate over indexes (not over `1`s).

4.4 Sampled Clocks

The *on* operator computes the rhythm of a sampled flow. It can express the output clock of the `when` operator that keeps or suppresses values of a flow of clock w_1 depending on a condition w_2:

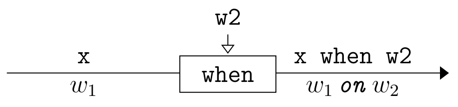


Figure 3. If `x` has clock w_1, `x when w2` has clock w_1 *on* w_2.

Definition 9 (*on* operator).

$$\begin{aligned}
0w_1 \ \mathit{on}\ w_2 &\overset{def}{=} 0(w_1 \ \mathit{on}\ w_2) \\
1w_1 \ \mathit{on}\ 1w_2 &\overset{def}{=} 1(w_1 \ \mathit{on}\ w_2) \\
1w_1 \ \mathit{on}\ 0w_2 &\overset{def}{=} 0(w_1 \ \mathit{on}\ w_2)
\end{aligned}$$

For example, if $w_1 = 11010111\ldots$ and $w_2 = 101100\ldots$, then $w_1 \ \mathit{on}\ w_2 = 10010100\ldots$.
Consider the sampling of a flow `x` with clock w_1 by a condition w_2:

x	2	5		3		7	9	4	...
w2	1	0		1		1	0	0	...
x when w2	2			3		7			...

w_1	1	1	0	1	0	1	1	1	...
w_2	1	0		1		1	0	0	...
w_1 *on* w_2	1	0	0	1	0	1	0	0	...

We observe that if `x` is present (*i.e.*, the corresponding element of w_1 is equal to `1`), then we look at the next element of the sampling condition (w_2). If this element is `1`, then the value of x is kept by the sampling and the flow `x when w2` is present (w_1 *on* w_2 equals `1`). Otherwise, the value of `x` is suppressed by the sampling and the flow `x when w2` is absent (w_1 *on* w_2 equals `0`). Finally, if `x` is absent (*i.e.*, w_1 equals `0`), the flow `x when w2` is absent (*i.e.*, w_1 *on* w_2 equals `0`) and the sampling condition (w_2) is not considered.

The elements of w_1 *on* w_2 can be seen as the elements of w_2, when w_2 is traversed at the rhythm of the `1`s in w_1. Thus, it is not necessary to explicitly compute the word w_1 *on* w_2 to have its cumulative function: it can be computed from the cumulative functions of w_1 and w_2.

Proposition 4 (cumulative function of w_1 *on* w_2)**.**

$$\forall i \geq 0,\ \mathcal{O}_{w_1 \textit{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$$

Proof. By induction on i.
By definition of $\mathcal{O}$, $\mathcal{O}_{w_1 \textit{ on } w_2}(0) = 0 = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(0))$.
Suppose that $\mathcal{O}_{w_1 \textit{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$, then prove that $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$.

By definition of $\mathcal{O}$,

$$\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \begin{cases} \mathcal{O}_{w_1 \textit{ on } w_2}(i) & \text{if } (w_1 \textit{ on } w_2)[i+1] = \mathtt{0} \\ \mathcal{O}_{w_1 \textit{ on } w_2}(i) + 1 & \text{if } (w_1 \textit{ on } w_2)[i+1] = \mathtt{1} \end{cases}$$

Case $(w_1 \textit{ on } w_2)[i+1] = \mathtt{0}$**:** $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_1 \textit{ on } w_2}(i)$.
By induction hypothesis, $\mathcal{O}_{w_1 \textit{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$.
By Definition 9, we have:

$$(w_1 \textit{ on } w_2)[i+1] = \mathtt{0} \Leftrightarrow w_1[i+1] = \mathtt{0} \vee (w_1[i+1] = \mathtt{1} \wedge w_2[\mathcal{O}_{w_1}(i+1)] = \mathtt{0})$$

If $w_1[i+1] = \mathtt{0}$, we have $\mathcal{O}_{w_1}(i+1) = \mathcal{O}_{w_1}(i)$.
Hence, $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$ and the property $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ is proved.
If $w_1[i+1] = \mathtt{1} \wedge w_2[\mathcal{O}_{w_1}(i+1)] = \mathtt{0}$, we have
$\mathcal{O}_{w_1}(i+1) = \mathcal{O}_{w_1}(i) + 1$ and
$\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1) - 1)$.
Hence, $\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$ and the property $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ is proved.

Case $(w_1 \textit{ on } w_2)[i+1] = \mathtt{1}$**:** $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_1 \textit{ on } w_2}(i) + 1$.
By induction hypothesis, $\mathcal{O}_{w_1 \textit{ on } w_2}(i) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i))$.
By Definition 9, we have:

$$(w_1 \textit{ on } w_2)[i+1] = \mathtt{1} \Leftrightarrow (w_1[i+1] = \mathtt{1} \wedge w_2[\mathcal{O}_{w_1}(i+1)] = \mathtt{1})$$

Therefore
$$\begin{aligned}\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1)) &= \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1) - 1) + 1 \\ &= \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) + 1\end{aligned}$$
and the property $\mathcal{O}_{w_1 \textit{ on } w_2}(i+1) = \mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i+1))$ is proved.

□

To compute the *on* operator on two ultimately periodic binary words p_1 and p_2, we first compute the size of the expected result, *i.e.*, $|(p_1 \textit{ on } p_2).u|$ the size of the prefix and $|(p_1 \textit{ on } p_2).v|$ the size of the periodic part. Then, we compute the value of the elements of the prefix and the periodic part by the application of Definition 9.

Proposition 5 (computation of p_1 *on* p_2)**.**
Let $p_1 = u_1(v_1)$ *and* $p_2 = u_2(v_2)$.
Then $p_1 \textit{ on } p_2 = u_3(v_3)$ *with:*

$$|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$$

$$|v_3| = \frac{\operatorname{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$$

and $\forall i,\ 1 \leq i \leq |u_3|,\ \ u_3[i] = (p_1 \textit{ on } p_2)[i]$

$$\forall i,\ 1 \leq i \leq |v_3|,\ \ v_3[i] = (p_1 \textit{ on } p_2)[|u_3| + i]$$

Intuitively, the prefix of p_1 *on* p_2 terminates when the processing of the prefix of p_1 and p_2 in the computation of p_1 *on* p_2 terminates. Since the elements of p_2 are processed only when there is a `1` in p_1, the processing of the elements of the prefix of p_2 terminates at the index $\mathcal{I}_{p_1}(|u_2|)$. The size of the periodic pattern is obtained by the computation of the common period of p_1 and p_2 when p_2 is processed at the rhythm of the `1`s of p_1.

Proof. Let $p_1 = u_1(v_1)$, $p_2 = u_2(v_2)$
and $p_3 = p_1 \textit{ on } p_2 = u_3(v_3)$.
We gives the proof of the general case where p_1 and p_2 and be of null rate. Therefore, we will prove that

$$|u_3| = \begin{cases} |u_1| & \text{if } |u_2| = 0 \\ \max(|u_1|, \mathcal{I}_{p_1}(|u_2|)) & \text{otherwise} \end{cases}$$
$$|v_3| = \begin{cases} 1 & \text{if } |v_1|_1 = 0 \\ \frac{\operatorname{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1| & \text{otherwise} \end{cases}$$

But first, note that for all ultimately periodic word $p = u(v)$,

$$p[i] = \begin{cases} u[i] & \text{if } i \leq |u| \\ v[1 + (i - |u| - 1) \bmod |v|] & \text{if } i > |u| \end{cases}$$

We search the size of the prefix of p_1 *on* p_2**.** To this aim, we compute p_1 *on* p_2 to determine when the periodic pattern starts.
By definition of *on*,

$$p_3[i] = \begin{cases} \mathtt{0} & \text{if } p_1[i] = \mathtt{0} \\ p_2[\mathcal{O}_{p_1}(i)] & \text{if } p_1[i] = \mathtt{1} \end{cases}$$

We study the different cases depending on the value of i with respect to $|u_1|$ and $|u_2|$:

Case $i \leq |u_1|$**:**
$$p_3[i] = \begin{cases} \mathtt{0} & \text{if } u_1[i] = \mathtt{0} \\ p_2[\mathcal{O}_{u_1}(i)] & \text{if } u_1[i] = \mathtt{1} \end{cases}$$
Two subcases can occur if $u_1[i] = \mathtt{1}$:
Case $\mathcal{O}_{u_1}(i) \leq |u_2|$**:**
$p_3[i] = u_2[\mathcal{O}_{u_1}(i)]$
Case $\mathcal{O}_{u_1}(i) > |u_2|$**:**
$p_3[i] = v_2[1 + (\mathcal{O}_{u_1}(i) - |u_2| - 1) \bmod |v_2|]$

Case $i > |u_1|$**:**
$$p_3[i] = \begin{cases} \mathtt{0} & \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{0} \\ p_2[\mathcal{O}_{p_1}(i)] & \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{1} \end{cases}$$
Two subcases can occur if $v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{1}$:
Case $\mathcal{O}_{p_1}(i) \leq |u_2|$**:**
$p_3[i] = u_2[\mathcal{O}_{p_1}(i)]$
Case $\mathcal{O}_{p_1}(i) > |u_2|$**:**
$p_3[i] = v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \bmod |v_2|]$

Note that for all $i > |u_1|$ such that $\mathcal{O}_{p_1}(i) > |u_2|$, we have:
$$p_3[i] = \begin{cases} \mathtt{0} & \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{0} \\ v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \bmod |v_2|] & \\ \quad \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{1} & \end{cases}$$
Therefore, we can obtain the index from which the periodic behavior starts:

Case $|u_2| = 0$**:**
In this case, for all i, we have $\mathcal{O}_{p_1}(i) > |u_2|$. The previous formula is thus verified for all $i > |u_1|$. It means that, when $i \leq |u_1|$, we are in the initialization phase.

Case $|u_2| > 0$:
Since $\mathcal{O}_{p_1}(i) > |u_2| \Leftrightarrow i \geq \mathcal{I}_{p_1}(|u_2|+1)$ and moreover since $\forall i \geq 1,\ \mathcal{I}_{p_1}(|u_2|) < i < \mathcal{I}_{p_1}(|u_2|+1),\ p_1[i] = \mathtt{0}$, we obtain: $\forall i > \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$,

$$p_3[i] = \begin{cases} \mathtt{0} & \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{0} \\ v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \bmod |v_2|] \\ \qquad \text{if } v_1[1 + (i - |u_1| - 1) \bmod |v_1|] = \mathtt{1} \end{cases}$$

It means that, when $i \leq \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$, we are in the initialization phase.

We search now the size of the periodic pattern of p_1 *on* p_2, that is the smallest $x \in \mathbb{N} - \{0\}$ such that $\forall i > |u_3|,\ p_3[i+x] = p_3[i]$. When $\forall i > |u_3|$, by the previous formulas we have:

$$\begin{aligned} & p_3[i+x] = p_3[i] \Leftrightarrow \\ & \quad v_1[1 + (i + x - |u_1| - 1) \bmod |v_1|] \\ & \qquad = v_1[1 + (i - |u_1| - 1) \bmod |v_1|] \qquad (1) \\ & \quad \wedge \\ & \quad v_2[1 + (\mathcal{O}_{p_1}(i+x) - |u_2| - 1) \bmod |v_2|] \\ & \qquad = v_2[1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \bmod |v_2|] \qquad (2) \end{aligned}$$

Since we do not interpret the elements of v_1,

$$\begin{aligned} (1) \quad & \Leftrightarrow \quad 1 + (i + x - |u_1| - 1) \bmod |v_1| \\ & \qquad = 1 + (i - |u_1| - 1) \bmod |v_1| \\ & \Leftrightarrow \quad x = x' \times |v_1| \text{ avec } x' \in \mathbb{N} - \{0\} \end{aligned}$$

Since we do not interpret the elements of v_2,

$$\begin{aligned} (2) \quad & \Leftrightarrow \quad 1 + (\mathcal{O}_{p_1}(i+x) - |u_2| - 1) \bmod |v_2| \\ & \qquad = 1 + (\mathcal{O}_{p_1}(i) - |u_2| - 1) \bmod |v_2| \\ & \Leftrightarrow \quad \forall i > |u_3|,\ \mathcal{O}_{p_1}(i+x) \bmod |v_2| = \mathcal{O}_{p_1}(i) \bmod |v_2| \end{aligned}$$

Hence,

$$\begin{aligned} (1) \wedge (2) \quad & \Leftrightarrow \quad \mathcal{O}_{p_1}(i + x' \times |v_1|) \bmod |v_2| \\ & \qquad = \mathcal{O}_{p_1}(i) \bmod |v_2| \\ & \Leftrightarrow \quad (\mathcal{O}_{p_1}(i) + x' \times |v_1|_1) \bmod |v_2| \\ & \qquad = \mathcal{O}_{p_1}(i) \bmod |v_2| \\ & \Leftrightarrow \quad (x' \times |v_1|_1) \bmod |v_2| = 0 \end{aligned}$$

The smallest x' which satisfies the equation is $\frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1}$.
Therefore $|v_3| = x = \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$.

Finally, we compute the elements of u_3 and v_3. Since we know $|u_3|$ and $|v_3|$, it is trivial that

$$\forall i,\ 1 \leq i \leq |u_3|,\ u_3[i] = (p_1 \ \mathit{on}\ p_2)[i]$$

$$\forall i,\ 1 \leq i \leq |v_3|,\ v_3[i] = (p_1 \ \mathit{on}\ p_2)[|u_3| + i]$$

□

We can notice that the 1s of p_1 *on* p_2 are the 1s coming from p_2. Therefore, we can compute $|(p_1 \ \mathit{on}\ p_2).v|_1$:

Lemma 4 (number of 1s in the periodic pattern of a *on*).
Let $p_1 = u_1(v_1)$, $p_2 = u_2(v_2)$.

$$|(p_1 \ \mathit{on}\ p_2).v|_1 = \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$$

Proof. Let $p_1 = u_1(v_1)$, $p_2 = u_2(v_2)$
and $p_3 = p_1 \ \mathit{on}\ p_2 = u_3(v_3)$.
By definition, $|v_3|_1 = \mathcal{O}_{p_3}(|u_3| + |v_3|) - \mathcal{O}_{p_3}(|u_3|)$.
By Proposition 4, $|v_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) - \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|))$.
By Proposition 5, $|v_3| = \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$.
Therefore, $\mathcal{O}_{p_1}(|u_3| + |v_3|) = \mathcal{O}_{p_1}(|u_3| + \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|)$.
We can deduce that

$$\begin{aligned} \mathcal{O}_{p_1}(|u_3| + |v_3|) &= \mathcal{O}_{p_1}(|u_3|) + \tfrac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|_1 \\ &= \mathcal{O}_{p_1}(|u_3|) + \mathrm{lcm}(|v_1|_1, |v_2|) \end{aligned}$$

Therefore,

$$\begin{aligned} \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) &= \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|) + \mathrm{lcm}(|v_1|_1, |v_2|)) \\ &= \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|) + \tfrac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|) \end{aligned}$$

We can deduce that

$$\mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|)) + \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$$

Since $|v_3|_1 = \mathcal{O}_{p_3}(|u_3| + |v_3|) - \mathcal{O}_{p_3}(|u_3|)$, we can conclude that

$$\begin{aligned} |v_3|_1 &= \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3| + |v_3|)) - \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|)) \\ &= \tfrac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1 \end{aligned}$$

□

The result of the *on* operation can be simpler to compute for certain shape of arguments. The simplest case is the one where the number of 1s in the prefix and in the periodic pattern of the first word are equal respectively to the size of the prefix and the size of the periodic pattern of the second word as in the following example:

p_1	1	1	0	1	(	1	1	1	0	0	1	1	0	)
p_2	1	0		1	(	1	0	0			1	0		)
p_1 *on* p_2	1	0	0	1	(	1	0	0	0	0	1	0	0	)

Proposition 6.
Gives p_1 and p_2 such that $|p_1.u|_1 = |p_2.u|$ and $|p_1.v|_1 = |p_2.v|$. Then:

$$|(p_1 \ \mathit{on}\ p_2).u| = |p_1.u| \qquad |(p_1 \ \mathit{on}\ p_2).u|_1 = |p_2.u|_1$$
$$|(p_1 \ \mathit{on}\ p_2).v| = |p_1.v| \qquad |(p_1 \ \mathit{on}\ p_2).v|_1 = |p_2.v|_1$$

Proof. Let $p_1 = u_1(v_1)$, $p_2 = u_2(v_2)$,
and $p_1 \ \mathit{on}\ p_2 = u_3(v_3)$.
First, we prove that $|u_3| = |u_1|$ and $|v_3| = |v_1|$.
By Proposition 5,

$$|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|)) \qquad |v_3| = \tfrac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$$

By hypothesis, $|u_1|_1 = |u_2|$ and $|v_1|_1 = |v_2|$.
Therefore, $\mathcal{I}_{p_1}(|u_2|) = \mathcal{I}_{p_1}(|u_1|_1)$ and $\mathcal{I}_{p_1}(|u_1|_1) \leq |u_1|$. We can thus simplify the previous formula as follows:

$$|u_3| = |u_1| \qquad |v_3| = |v_1|$$

Then, we prove that $|v_3|_1 = |v_2|_1$.
By Lemma 4, $|v_3|_1 = \frac{\mathrm{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$ which can be simplified as $|v_3|_1 = |v_2|_1$.
Finally, we prove that $|u_3|_1 = |u_2|_1$.
By definition, $|u_3|_1 = \mathcal{O}_{p_3}(|u_3|)$.
By Property 4, $|u_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|))$.
By hypothesis, $|u_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_1|))$.
By definition, $|u_3|_1 = \mathcal{O}_{p_2}(|u_1|_1)$.
By hypothesis, $|u_3|_1 = \mathcal{O}_{p_2}(|u_2|)$.
By definition, $|u_3|_1 = |u_2|_1$. □

As explained in Remark 6, it is possible to increase the size and the number of 1s in the prefixes and periodic patterns of words. Therefore, we can always adjust the operands of the *on* such that they satisfy the assumptions of proposition 6.

Remark 7. *Explicitly increasing the size of the operands of the on operator is not strictly necessary. We need only compute the size they should have after the adjustment. So, if we want to compute p_1 on p_2, we suppose that we rewrite p_1 as p'_1 and p_2 as p'_2 such that $|p'_1.u|_1 = |p'_2.u|$ and $|p'_1.v|_1 = |p'_2.v|$. The size of p_1 on p_2 is the size of p'_1 and it has the same number of* 1s *as does p'_2.*

Finally, to compute the index of the $j\text{th}$ 1 of $w_1 \mathit{ on } w_2$ ($\mathcal{I}_{w_1 \mathit{ on } w_2}(j)$), it is not necessary to compute the word $w_1 \mathit{ on } w_2$ and then to apply the $\mathcal{I}$ function. It can be computed directly from $\mathcal{I}_{w_1}$ and $\mathcal{I}_{w_2}$.

Proposition 7 (index of the $j\text{th}$ 1 of $w_1 \mathit{ on } w_2$)**.**

$$\forall j \geq 1,\ \mathcal{I}_{w_1 \mathit{ on } w_2}(j) = \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$$

Indeed, in the computation of $w_1 \mathit{ on } w_2$, the elements of w_2 are given when there is a 1 in w_1. Therefore the index of the $i\text{th}$ element of w_2 is at index $\mathcal{I}_{w_1}(i)$. Since the 1s of $w_1 \mathit{ on } w_2$ are the 1s of w_2, the $j\text{th}$ 1 of $w_1 \mathit{ on } w_2$ is the $\mathcal{I}_{w_2}(j)\text{th}$ element of w_2 and thus at the index $\mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j))$.

Proof. $\mathcal{I}_{w_1 \mathit{ on } w_2}(j)$ is the smallest i such that $\mathcal{O}_{w_1 \mathit{ on } w_2}(i) = j$. Therefore, by Proposition 4,

$$\mathcal{I}_{w_1 \mathit{ on } w_2}(j) = \min_{i \in \mathbb{N}^*} \{i,\ (\mathcal{O}_{w_2}(\mathcal{O}_{w_1}(i)) = j)\}$$

Since $\mathcal{O}$ is increasing, we can deduce that

$$\mathcal{I}_{w_1 \mathit{ on } w_2}(j) = \min_{i \in \mathbb{N}^*} \left\{ i,\ (\mathcal{O}_{w_1}(i) = \min_{i' \in \mathbb{N}^*} \{i',\ (\mathcal{O}_{w_2}(i') = j)\}) \right\}$$

We can conclude that

$$\begin{aligned} \mathcal{I}_{w_1 \mathit{ on } w_2}(j) &= \min_{i \in \mathbb{N}^*} \{i,\ (\mathcal{O}_{w_1}(i) = \mathcal{I}_{w_2}(j))\} \\ &= \mathcal{I}_{w_1}(\mathcal{I}_{w_2}(j)) \end{aligned}$$

□

We now have all the algebraic tools needed to define an algorithm for the resolution of adaptability constraints on ultimately periodic binary words.

5. Adaptability Constraints Resolution Algorithm

We saw in Section 3 that subtyping constraints can be reduced to adaptability constraints where the unknowns are no longer types but rather ultimately periodic binary words. This is the first step of the subtyping constraints resolution algorithm which is summarized in Figure 4. This section details the remaining steps.

In Section 5.1, we explain how to simplify an adaptability constraint system (S2) to obtain a system (S3) where all the constraints have the form $c_x \mathit{ on } p_x <: c_y \mathit{ on } p_y$. In Section 5.2, we explain the transformation of adaptability constraints (S3) into a system of linear inequalities (S6) where the unknowns are the size and the indexes of 1s of the sought words. This last system can be solved using standard techniques from Integer Linear Programming, and the resulting solutions can be used to reconstruct the unknown words. In Section 5.3, we discuss the choice of the objective function for the resolution of the linear inequalities. Finally, we discuss the correctness, completeness and complexity of the algorithm.

Before the detailed explanation of the algorithm, we note that, as was the case for the equality constraints, there is no greedy algorithm for solving adaptability constraints. Indeed, for a given rate, there is no word which is smaller (or greater) than all the other words in the relation $\preceq$[7] and thus nor in the relation $<:$. For example: $\ldots <: \mathtt{1}^{999}\mathtt{(10)} <: \ldots <: \mathtt{1(10)} <: \mathtt{(10)} <: \mathtt{0(10)} <: \ldots <: \mathtt{0}^{999}\mathtt{(10)} <: \ldots$ In fact, if given c_1 and c_2 which satisfy a constraint $c_1 \mathit{ on } p_1 <: c_2 \mathit{ on } p_2$, then the following words also satisfy it:

- $\mathtt{1}^{d_1}c_1$ and $\mathtt{0}^{d_2}c_2$, whatever the d_1 and d_2,
- $\mathtt{1}^{d_1}c_1$ and $\mathtt{1}^{d_2}c_2$, whatever the d_1 and d_2 provided that $d_1 \geq d_2$,
- $\mathtt{0}^{d_1}c_1$ and $\mathtt{0}^{d_2}c_2$, whatever the d_1 and d_2 provided that $d_1 \leq d_2$.

[7] If the rate is not fixed, this property is not true for the words (1) and (0). Indeed, for all p, $\mathtt{(1)} \preceq p \preceq \mathtt{(0)}$

S1. Subtyping constraints: $\alpha_x \texttt{ on } p \texttt{ on } \ldots$ **<:** $\alpha_y \texttt{ on } p' \texttt{ on } \ldots$

$\Leftrightarrow$ { introduction of word variables c_n ;
simplification of type variables }

S2. Adaptability constraints: $c_x \mathit{ on } p \mathit{ on } \ldots <: c_y \mathit{ on } p \mathit{ on } \ldots$
$p \mathit{ on } \ldots <: p' \mathit{ on } \ldots$

$\Leftrightarrow$ { computation of *on* ;
simplification of $p <: p'$ constraints }

S3. Simplified adaptability constraints: $c_x \mathit{ on } p_x <: c_y \mathit{ on } p_y$

$\Leftrightarrow$ { for each c_n, equalization of the size of its samplers }

S4. Adjusted adaptability constraints: $c_x \mathit{ on } p_x <: c_y \mathit{ on } p_y$

$\Leftarrow$ { choice of the number of 1s of the c_ns ;
splitting of the adaptability constraints }

S5. Synchronizability and precedence constraints:
$c_x \mathit{ on } p_x \bowtie c_y \mathit{ on } p_y$
$c_x \mathit{ on } p_x \preceq c_y \mathit{ on } p_y$

$\Leftrightarrow$ { simplification of synchronizability and precedence constraints ;
introduction of well formedness constraints }

S6. Indexes of 1s and size constraints:

synchronizability:	$\lvert p_y.v\rvert_1 \times \lvert c_x.v\rvert = \lvert p_x.v\rvert_1 \times \lvert c_y.v\rvert$
precedence:	$\mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$
periodicity:	$\mathcal{I}_{c_n}(j + \lvert c_n.v\rvert_1) - \mathcal{I}_{c_n}(j) = \lvert c_n.v\rvert$
sufficient size:	$1 + \mathcal{I}_{c_n}(\lvert c_n.u\rvert_1 + \lvert c_n.v\rvert_1) - \mathcal{I}_{c_n}(\lvert c_n.u\rvert_1 + 1) \leq \lvert c_n.v\rvert$
sufficient indexes:	$\mathcal{I}_{c_n}(j) \geq j$
increasing indexes:	$\mathcal{I}_{c_n}(j') - \mathcal{I}_{c_n}(j) \geq j' - j$

Figure 4. Summary of the subtyping constraints resolution algorithm. The form of the constraints is given for each system.

Hence, contrary to a classical subtyping system, we cannot simply take the greatest word for a variable on the left of an adaptability constraint, and the smallest one for a variable on the right. In our case, the inference of values satisfying constraints must necessarily be performed globally to choose words big enough, and/or small enough to satisfy all constraints.

5.1 Constraint System Simplification

The systems to be solved comprise constraints of the form $p_x <: p_y$ and $c_x \mathit{ on } p_x <: c_y \mathit{ on } p_y$, where p_x and p_y are known words of non-null rate and c_x and c_y are unknown words. These constraints are obtained after the computation of *on* operators (Proposition 5).

Since a constraint of the form $p_x <: p_y$ contains no variables, its truth value cannot be altered. We need only to check that each such constraint is satisfied, which is done applying Definition 7. If any are false, then the whole system is unsatisfiable. The true constraints can be removed from the system.

Returning to the `cyclic_encoding` example, the adaptability constraint system was:

$$\left\{ \begin{array}{lcl} \mathtt{(0^{50}100)} & <: & \mathtt{(0^{50}100)} \\ \mathtt{(0^{50}100)} & <: & \mathtt{(0^{50}010)} \\ \mathtt{(0^{50}100)} & <: & \mathtt{(0^{50}001)} \end{array} \right\}$$

Through the application of the adaptability test, we can check that each constraint is always satisfied. Here, after simplification, the system is empty.

In the general case, after simplification all the remaining constraints contain variables. For example, the subtyping constraint system C' of Section 3 can be rewritten to the adaptability constraint system A':

$$\theta(C') \Leftrightarrow \left\{ \begin{array}{rcl} c_1 \text{ on } \mathtt{10(1)} & <: & c_2 \text{ on } \mathtt{(01)} \\ \mathtt{(1100)} & <: & \mathtt{(01)} \\ c_2 \text{ on } \mathtt{(01)} \text{ on } \mathtt{(10)} & <: & c_3 \text{ on } \mathtt{(1)} \\ c_2 \text{ on } \mathtt{(01)} \text{ on } \mathtt{(01)} & <: & c_3 \text{ on } \mathtt{(1)} \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{lcl} c_1 \text{ on } \mathtt{10(1)} & <: & c_2 \text{ on } \mathtt{(01)} \\ c_2 \text{ on } \mathtt{(0100)} & <: & c_3 \text{ on } \mathtt{(1)} \\ c_2 \text{ on } \mathtt{(0001)} & <: & c_3 \text{ on } \mathtt{(1)} \end{array} \right\} = A'$$

All the remaining constraints are of the form $c_x \text{ on } p_x <: c_y \text{ on } p_y$.

5.2 Constraint System Solving

The goal now is to solve adaptability constraint systems like A':

$$\left\{ \begin{array}{lcl} c_1 \text{ on } p_1 & <: & c_2 \text{ on } p_2 \\ c_2 \text{ on } p'_2 & <: & c_3 \text{ on } p_3 \\ c_2 \text{ on } p''_2 & <: & c_3 \text{ on } p'_3 \end{array} \right\}$$

The values p_1, p_2, p'_2, p''_2, p_3, p'_3 are some known ultimately periodic binary words of non-null rate. The variables c_1, c_2, c_3 are the unknowns of the system. Solving the system means associating ultimately periodic words of non-null rate to the unknowns, such that the constraints are satisfied.

Remark 8. *If solutions containing null rates are allowed, all systems have a solution. For example, the instantiation $\forall n.\ c_n = \mathtt{(0)}$ is a trivial solution of all systems. A solution containing a null rate gives a system that will be executed at only a finite number of instants. We are not interested in such solutions.*

An adaptability constraint $c_x \text{ on } p_x <: c_y \text{ on } p_y$ can be decomposed into a synchronizability constraint and a precedence constraint:

$$\begin{array}{lll} c_x \text{ on } p_x <: c_y \text{ on } p_y & \Leftrightarrow & \{ \text{ by Definition 7 } \} \\ & & (c_x \text{ on } p_x \bowtie c_y \text{ on } p_y) \wedge \\ & & (c_x \text{ on } p_x \preceq c_y \text{ on } p_y) \end{array}$$

The synchronizability constraint can be itself rewritten:

$$\begin{array}{ll} \multicolumn{2}{l}{c_x \text{ on } p_x \bowtie c_y \text{ on } p_y} \\ \Leftrightarrow & \{ \text{ by Proposition 1 } \} \\ & rate(c_x \text{ on } p_x) = rate(c_y \text{ on } p_y) \\ \Leftrightarrow & \{ \text{ by Definition 4 } \} \\ & \frac{|(c_x \text{ on } p_x).v|_1}{|(c_x \text{ on } p_x).v|} = \frac{|(c_y \text{ on } p_y).v|_1}{|(c_y \text{ on } p_y).v|} \end{array}$$

As can the precedence constraint:

$$\begin{array}{ll} \multicolumn{2}{l}{c_x \text{ on } p_x \preceq c_y \text{ on } p_y} \\ \Leftrightarrow & \{ \text{ by Proposition 2 } \} \\ & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x \text{ on } p_x}(j) \leq \mathcal{I}_{c_y \text{ on } p_y}(j) \\ \Leftrightarrow & \{ \text{ by Proposition 7 } \} \\ & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \end{array}$$

$$\text{with } h = \max(|(c_x \text{ on } p_x).u|_1, |(c_y \text{ on } p_y).u|_1) + \text{lcm}(|(c_x \text{ on } p_x).v|_1, |(c_y \text{ on } p_y).v|_1)$$

We are thus interested in the size of, and the number of `1`s in, the prefixes and periodic patterns of $c_x \text{ on } p_x$ and of $c_y \text{ on } p_y$ forms.

We have seen in Section 4 (Proposition 6) that the size and number of `1`s in the prefix and in the periodic pattern of $c_n \text{ on } p_n$ can easily be expressed as a function of the size and number of `1`s in the prefixes and periodic patterns of c_n and p_n in the following case:

– if $|c_n.u|_1 = |p_n.u|$, then $|(c_n \text{ on } p_n).u|_1 = |p_n.u|_1$ and $|(c_n \text{ on } p_n).u| = |c_n.u|$.

– if $|c_n.v|_1 = |p_n.v|$, then $|(c_n \text{ on } p_n).v|_1 = |p_n.v|_1$ and $|(c_n \text{ on } p_n).v| = |c_n.v|$.

To put the system into this "simple" form, we adjust the known words of the system such that the following property is satisfied: for any unknown c_n, all the words p_n, p'_n, ... such that $c_n \text{ on } p_n$, $c_n \text{ on } p'_n$, ... that appear in the system have the same prefix size ($|p_n.u| = |p'_n.u| = \ldots$) and the same periodic pattern size ($|p_n.v| = |p'_n.v| = \ldots$). This operation is always possible (thanks to Remark 6) and does not change the semantics of the system.

We can then make the following choice for the number of `1`s in the unknown c_n.

Choice 1 (number of `1`s in the c_n)**.**

$$\begin{array}{l} |c_n.u|_1 = |p_n.u| = |p'_n.u| = \ldots \\ |c_n.v|_1 = |p_n.v| = |p'_n.v| = \ldots \end{array}$$

where p_n, p'_n, ... are the samplers of c_n (i.e., the words such that $c_n \text{ on } p_n$, $c_n \text{ on } p'_n$, ... appear in the constraint system)

Remark 9. *We will discuss this choice in Sections 5.4 and 6.1.*

This choice allows us to express the size and the number of `1`s in the prefixes and periodic patterns of the $c_n \text{ on } p_n$ in terms of those of c_n and p_n. Hence, a synchronizability constraint becomes:

$$\begin{array}{lll} \multicolumn{3}{l}{c_x \text{ on } p_x \bowtie c_y \text{ on } p_y} \\ \Leftrightarrow & \{ \text{ by Proposition 6 } \} & \\ & \frac{|p_x.v|_1}{|c_x.v|} = \frac{|p_y.v|_1}{|c_y.v|} & \\ \Leftrightarrow & |p_y.v|_1 \times |c_x.v| = |p_x.v|_1 \times |c_y.v| & (1) \end{array}$$

And a precedence constraint becomes:

$$\begin{array}{lll} \multicolumn{3}{l}{c_x \text{ on } p_x \preceq c_y \text{ on } p_y} \\ \Leftrightarrow & \{ \text{ by Proposition 6 } \} & \\ & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) & \\ & \text{with } h = \max(|p_x.u|_1, |p_y.u|_1) & (2) \\ & \qquad + \text{lcm}(|p_x.v|_1, |p_y.v|_1) & \end{array}$$

For example, we can adjust the system A' such that all the samplers of a particular variable have the same size:

$$A' = \left\{ \begin{array}{lcl} c_1 \text{ on } \mathtt{10(1)} & <: & c_2 \text{ on } \mathtt{(01)} \\ c_2 \text{ on } \mathtt{(0100)} & <: & c_3 \text{ on } \mathtt{(1)} \\ c_2 \text{ on } \mathtt{(0001)} & <: & c_3 \text{ on } \mathtt{(1)} \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{lcl} c_1 \text{ on } \mathtt{10(1)} & <: & c_2 \text{ on } \mathtt{(0101)} \\ c_2 \text{ on } \mathtt{(0100)} & <: & c_3 \text{ on } \mathtt{(1)} \\ c_2 \text{ on } \mathtt{(0001)} & <: & c_3 \text{ on } \mathtt{(1)} \end{array} \right\}$$

Then, we choose the number of `1`s for the c_ns to be equal to the size of the respective samplers:

$$\begin{array}{lll} |c_1.u|_1 = 2 & |c_2.u|_1 = 0 & |c_3.u|_1 = 0 \\ |c_1.v|_1 = 1 & |c_2.v|_1 = 4 & |c_3.v|_1 = 1 \end{array}$$

By Formula (1), the synchronizability constraints become a system of linear equations on the size of the periodic patterns of the c_ns:

$$\left\{ \begin{array}{rcl} |\mathtt{(0101)}.v|_1 \times |c_1.v| & = & |\mathtt{(10(1))}.v|_1 \times |c_2.v| \\ |\mathtt{(1)}.v|_1 \times |c_2.v| & = & |\mathtt{(0100)}.v|_1 \times |c_3.v| \\ |\mathtt{(1)}.v|_1 \times |c_2.v| & = & |\mathtt{(0001)}.v|_1 \times |c_3.v| \end{array} \right\}$$

$$\Leftrightarrow \left\{ \begin{array}{rcl} 2 \times |c_1.v| & = & |c_2.v| \\ |c_2.v| & = & |c_3.v| \\ |c_2.v| & = & |c_3.v| \end{array} \right\} \qquad (Sync)$$

By Formula (2), the precedence constraints become a system of linear inequalities on the indexes of `1`s in the c_ns:

$$\left\{\begin{array}{lcl} \forall j,\ 1 \leq j \leq 3,\ \mathcal{I}_{c_1}(\mathcal{I}_{10(1)}(j)) & \leq & \mathcal{I}_{c_2}(\mathcal{I}_{(0101)}(j)) \\ \forall j,\ 1 \leq j \leq 1,\ \mathcal{I}_{c_2}(\mathcal{I}_{(0100)}(j)) & \leq & \mathcal{I}_{c_3}(\mathcal{I}_{(1)}(j)) \\ \forall j,\ 1 \leq j \leq 1,\ \mathcal{I}_{c_2}(\mathcal{I}_{(0001)}(j)) & \leq & \mathcal{I}_{c_3}(\mathcal{I}_{(1)}(j)) \end{array}\right\}$$

$$\Leftrightarrow \left\{\begin{array}{lcl} \mathcal{I}_{c_1}(1) & \leq & \mathcal{I}_{c_2}(2) \\ \mathcal{I}_{c_1}(3) & \leq & \mathcal{I}_{c_2}(4) \\ \mathcal{I}_{c_1}(4) & \leq & \mathcal{I}_{c_2}(6) \\ \mathcal{I}_{c_2}(2) & \leq & \mathcal{I}_{c_3}(1) \\ \mathcal{I}_{c_2}(4) & \leq & \mathcal{I}_{c_3}(1) \end{array}\right\} \qquad (\mathit{Prec})$$

Now, to give a value to each unknown c_n, we must find its size and the positions of its `1`s. Hence, the sizes $|c_n.v|$ and the indexes $\mathcal{I}_{c_n}(j)$ will no longer be considered as function applications, but rather as the new unknowns of the problem. These new unknowns must also satisfy the constraints of Remarks 2 to 5 which ensure that the solution will be a well formed ultimately periodic binary word. So, we have to augment the synchronizability constraints *Sync* and the precedence constraints *Prec* with the following four sets of constraints:[8]

Periodicity:

$$\mathit{Per} = \{\mathcal{I}_{c_n}(j + |c_n.v|_1) - \mathcal{I}_{c_n}(j) = |c_n.v|\}_{\substack{\mathcal{I}_{c_n}(j+|c_n.v|_1) \in \mathit{Prec} \\ \wedge\ j > |c_n.u|_1}}$$

Sufficient size:

$$\mathit{Size} = \{1 + \mathcal{I}_{c_n}(|c_n.u|_1 + |c_n.v|_1) - \mathcal{I}_{c_n}(|c_n.u|_1 + 1) \leq |c_n.v|\}$$

Sufficient indexes:

$$\mathit{Init} = \{\mathcal{I}_{c_n}(j) \geq j\}_{\mathcal{I}_{c_n}(j) \in \mathit{Prec} \cup \mathit{Per} \cup \mathit{Size}}$$

Increasing indexes:

$$\mathit{Incr} = \{\mathcal{I}_{c_n}(j') - \mathcal{I}_{c_n}(j) \geq j' - j\}_{(\mathcal{I}_{c_n}(j), \mathcal{I}_{c_n}(j')) \in \mathit{Prec} \cup \mathit{Per} \cup \mathit{Size}}$$

Finally, we can use a generic solver for Integer Linear Programming (ILP) problems to solve the system:

$$S = \mathit{Sync} \cup \mathit{Prec} \cup \mathit{Per} \cup \mathit{Size} \cup \mathit{Init} \cup \mathit{Incr}$$

Applying a solver to the system associated with A' produces the results:

$$|c_1.v| = 2 \qquad |c_2.v| = 4 \qquad |c_3.v| = 4$$
$$\mathcal{I}_{c_1}(1) = 1 \qquad \mathcal{I}_{c_1}(3) = 3 \qquad \mathcal{I}_{c_1}(4) = 5$$
$$\mathcal{I}_{c_2}(1) = 1 \qquad \mathcal{I}_{c_2}(2) = 2 \qquad \mathcal{I}_{c_2}(4) = 4 \qquad \mathcal{I}_{c_2}(6) = 6$$
$$\mathcal{I}_{c_3}(1) = 4$$

Thanks to this information and the number of `1`s in the prefixes and periodic patterns of the c_ns chosen previously, we can build the following solution to the A' system:

$$c_1 = \texttt{11(10)} \quad c_2 = \texttt{(1111)} = \texttt{(1)} \quad c_3 = \texttt{000(1000)} = \texttt{(0}^3\texttt{1)}$$

We now know all the clock types of the system of Figure 2. The result gives us the clock type of the node `f` which is $\forall\alpha,\ \alpha$ `on` $c_1 \times \alpha$ `on` $c_2 \rightarrow \alpha$ `on` c_3, that is:

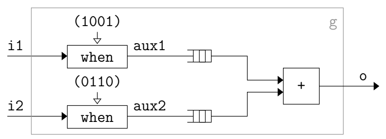
$$\texttt{f} :: \forall\alpha,\ \alpha \texttt{ on 11(10)} \times \alpha \texttt{ on (1)} \rightarrow \alpha \texttt{ on (0}^3\texttt{1)}$$

And since we have the types of the buffers, we can compute their sizes. For example, we know that the writing clock of the first buffer is of type α `on` c_1 `on 10(1)` $= \alpha$ `on 11(10) on 10(1)` and that the reading clock is of type α `on` c_2 `on (01)` $= \alpha$ `on (1) on (01)`. By Proposition 3, the size of this buffer is:

$$\mathit{size}(\texttt{11(10)}\ \mathit{on}\ \texttt{10(1)}, \texttt{(1)}\ \mathit{on}\ \texttt{(01)}) = 1$$

5.3 Guiding the Resolution Algorithm

The resolution algorithm requires the solution of linear inequalities on the indexes of `1`s and on the size of the unknown words. Tools for solving such inequalities are parameterized by an *objective function* determining the criterion to optimize. In the previous example, we choose to optimize the sum of the indexes of `1`s to produce a kind of As-Soon-As-Possible schedule. But we can also use the objective function to favor either system throughput or buffer sizes minimization. We illustrate this trade-off on an example:



```
let node g (i1, i2) = o where
  rec aux1 = i1 when (1001)
  and aux2 = i2 when (0110)
  and o = buffer aux1 + buffer aux2
```

If we give the type $\forall\alpha.\ (\alpha \times \alpha) \rightarrow \alpha$ `on (01)` to the node `g`, the buffers will be of size 1. But, this node can be executed without buffers if we give to it the type:

$$g :: \forall\alpha.(\alpha \texttt{ on (011110)} \times$$
$$\alpha \texttt{ on (110011)}) \rightarrow \alpha \texttt{ on (010010)}$$

The first solution can be obtained by an objective function that minimizes the size of the solution. Indeed, since the number of `1`s in the solution is fixed, minimizing the size increases the rate.

The second solution is obtained by an objective function that minimizes the precedence constraints. That is, for all precedence constraints $\mathcal{I}_{c_x}(j_1) \leq \mathcal{I}_{c_y}(j_2)$, we minimize the value $\mathcal{I}_{c_y}(j_2) - \mathcal{I}_{c_x}(j_1)$. It means that we minimize the number of instants between the writing and the reading of a value in a buffer which has the consequence of reducing the buffer sizes.

5.4 Correctness, Completeness and Complexity

The resolution algorithm shown in Figure 4 relies on the step-by-step transformation of the adaptability constraint system (S2) into linear inequalities (S6), for which there exist algorithms that find a solution if it exists [20]. Each step, except the one between S4 and S5, is a rewriting of a constraint system into an equivalent one (thanks to equivalence properties stated in Section 4). The step from S4 to S5 is the choice of the number of `1`s in the c_ns. It is correct to seek a solution in a subset of all possible words. Nevertheless, it may lead to incompleteness, since it is possible that a system has no solution in the subset of words considered.

Therefore, the solution computed by the resolution algorithm is correct and there is only one step which is a possible source of incompleteness.

We note that for optimization reasons, the indexes of `1`s that are not constrained by the adaptability constraints do not appear in the system of linear inequalities. When the words corresponding to the solution are built, these indexes are chosen such that they respect well formedness constraints.

The complexity of the resolution algorithm is dominated by the resolution of the constraint system on the indexes of `1`s and the sizes. This is an Integer Linear Programming (ILP) problem which is known to be NP-complete [20]. Even if there is only one adaptability constraint per buffer, the size of the complete ILP problem can be big: it depends on the size of the samplers in the adaptability constraint system.

6. Variants of the Resolution Algorithm

In this section, we present two variants of the resolution algorithm. The first one is complete but is only semi-decidable. The second one is complete and can be solved with polynomial complexity but it handles only a subset of systems.

[8] We will use the notation $\mathcal{I}_{c_n}(j) \in S$ to designate the presence of the unknown $\mathcal{I}_{c_n}(j)$ in S.

6.1 Variant 1: Semi-Decidable Complete Algorithm

The resolution algorithm presented Section 5 is correct and its only source of incompleteness is the choice of the number of 1s in the c_ns. We think this is a good choice; we have not yet come across any examples where if fails. However, it is also possible to build a semi-decidable algorithm which is complete.

The choice of the number of 1s in the c_ns is directed by Proposition 6 which simplifies the computation of the size of, and number of 1s in the result of an *on* operation. This proposition can be generalized as follows:

Proposition 8.
Given p_1 *and* p_2 *such that* $|p_1.u|_1 = |p_2.u| + k \times |p_2.v|$ *and* $|p_1.v|_1 = (k+1) \times |p_2.v|$ *with* $k \in \mathbb{N}$*, then:*

$$|(p_1 \text{ on } p_2).u| = |p_1.u| \qquad |(p_1 \text{ on } p_2).u|_1 = |p_2.u|_1 + k \times |p_2.v|_1$$
$$|(p_1 \text{ on } p_2).v| = |p_1.v| \qquad |(p_1 \text{ on } p_2).v|_1 = (k+1) \times |p_2.v|_1$$

Proof. Let $p_1 = u_1(v_1)$, $p_2 = u_2(v_2)$,
and $p_1 \text{ on } p_2 = u_3(v_3)$.
First, we prove that $|u_3| = |u_1|$ **and** $|v_3| = |v_1|$**.**
By Proposition 5,

$$|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|)) \qquad |v_3| = \frac{\operatorname{lcm}(|v_1|_1, |v_2|)}{|v_1|_1} \times |v_1|$$

By hypothesis, $|u_2| \leq |u_1|_1$.
Since $\mathcal{I}_{p_1}$ is increasing, $\mathcal{I}_{p_1}(|u_2|) \leq \mathcal{I}_{p_1}(|u_1|_1)$.
Since $\mathcal{I}_{p_1}(|u_1|_1) \leq |u_1|$, by transitivity, $\mathcal{I}_{p_1}(|u_2|) \leq |u_1|$.
Therefore $|u_3| = |u_1|$.

$$\begin{aligned}\text{By hypothesis,}\quad |v_3| &= \tfrac{\operatorname{lcm}((k+1)\times|v_2|, |v_2|)}{|v_1|_1} \times |v_1| \\ &= \tfrac{(k+1)\times|v_2|}{|v_1|_1} \times |v_1| \\ &= \tfrac{|v_1|_1}{|v_1|_1} \times |v_1|\end{aligned}$$

Therefore $|v_3| = |v_1|$.
Then, we prove that $|v_3|_1 = (k+1) \times |v_2|_1$**.**
By Lemma 4, $|v_3|_1 = \frac{\operatorname{lcm}(|v_1|_1, |v_2|)}{|v_2|} \times |v_2|_1$.

$$\begin{aligned}\text{By hypothesis,}\quad |v_3|_1 &= \tfrac{\operatorname{lcm}((k+1)\times|v_2|, |v_2|)}{|v_2|} \times |v_2|_1 \\ &= \tfrac{(k+1)\times|v_2|}{|v_2|} \times |v_2|_1\end{aligned}$$

Therefore, $|v_3|_1 = (k+1) \times |v_2|_1$.
Finally, we prove that $|u_3|_1 = |u_2|_1 + k \times |v_2|_1$**.**
By definition, $|u_3|_1 = \mathcal{O}_{p_3}(|u_3|)$.
By Property 4, $|u_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_3|))$.
By hypothesis, $|u_3|_1 = \mathcal{O}_{p_2}(\mathcal{O}_{p_1}(|u_1|))$.
By definition, $|u_3|_1 = \mathcal{O}_{p_2}(|u_1|_1)$.
By hypothesis, $|u_3|_1 = \mathcal{O}_{p_2}(|u_2| + k \times |v_2|)$.
By definition, $|u_3|_1 = |u_2|_1 + k \times |v_2|_1$. □

Therefore, in the resolution algorithm, instead of choosing the number of 1s in the prefix and in the periodic pattern of the c_n to be equal to the size of their samplers, we can parameterize our choice by a constant k:

Choice 2 (number of 1s in the c_n)**.** *Let* $k \in \mathbb{N}$.

$$|c_n.u|_1 = |p_n.u| + k \times |p_n.v| = |p'_n.u| + k \times |p'_n.v| = \dots$$
$$|c_n.v|_1 = (k+1) \times |p_n.v| = (k+1) \times |p'_n.v| = \dots$$

This modification effects the simplification of the synchronizability and precedence constraints.

Equation (1) becomes:

$$\begin{aligned}& c_x \text{ on } p_x \bowtie c_y \text{ on } p_y \\ \Leftrightarrow\ & \{ \text{ by Proposition 8 } \} \\ & \frac{(k+1) \times |p_x.v|_1}{|c_x.v|} = \frac{(k+1) \times |p_y.v|_1}{|c_y.v|} \\ \Leftrightarrow\ & |p_y.v|_1 \times |c_x.v| = |p_x.v|_1 \times |c_y.v| \qquad (3)\end{aligned}$$

In fact, the value k does not influence the synchronizability of a system (the equation (3) is the same as the equation (1)).

Equation (2) becomes:

$$\begin{aligned}& c_x \text{ on } p_x \preceq c_y \text{ on } p_y \\ \Leftrightarrow\ & \{ \text{ by Proposition 8 } \} \\ & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \\ & \text{with } h = \max(|p_x.u|_1 + k \times |p_x.v|_1, \\ & \qquad |p_y.u|_1 + k \times |p_y.v|_1) + \qquad (4) \\ & \qquad \operatorname{lcm}((k+1) \times |p_x.v|_1, \\ & \qquad (k+1) \times |p_y.v|_1)\end{aligned}$$

This new version constrains more 1s that the equation (2).

Finally, since the number of 1s in the sought words and the indexes of 1s involved in the precedence constraints have changed, the well formedness constraints (periodicity, sufficient size, sufficient indexes, and increasing indexes sets of constraints) must be modified accordingly.

Hence, we have parameterized our resolution algorithm by a constant k which modified the number of 1s in the sought solution.

A semi-decidable algorithm to solve adaptability constraints iterates the previous algorithm with $k = 0, 1, 2, \dots$ until it finds a solution. We can prove that this algorithm is complete because if a system of adaptability constraints has a solution S, then there exists a solution S' such that $\forall c'_n \in S'$,

$$|c'_n.u|_1 = |p_n.u| + k \times |p_n.v| = |p'_n.u| + k \times |p'_n.v| = \dots$$
$$|c'_n.v|_1 = (k+1) \times |p_n.v| = (k+1) \times |p'_n.v| = \dots$$

where k can be computed from the original solution S. The idea of the proof is to use Remark 6 to rewrite S into S'.

Proof. Let A be an adjusted adaptability constraint system (all the samplers of a variable have the same size) of the following shape:

$$\left\{ \begin{array}{ccc} & \dots & \\ c_x \text{ on } p_x & <: & c_y \text{ on } p_y \\ & \dots & \end{array} \right\}$$

Let S be a solution of non-null rate of A.

We first prove that there exists a solution S' **of** A **such that** $\forall c'_n \in S'$

$$|c'_n.u|_1 = |p_n.u| + k^u_n \times |p_n.v| = |p'_n.u| + k^u_n \times |p'_n.v| = \dots$$
$$|c'_n.v|_1 = k^v_n \times |p_n.v| = k^v_n \times |p'_n.v| = \dots$$

where $p_n, p'_n, \dots$ **are the samplers of** c_n **in** A **and where** k^u_n **and** k^v_n **are integer constants.**
First we can notice that $\forall c_n \in S, rate(c_n) > 0$. Therefore, we can increase the number of 1s in the prefix of the solution by an arbitrary number and we can multiply the number of 1s in the periodic pattern an arbitrary number of times.

Prefix case:
For all $c_n \in S$, we will increase the number of 1s in the prefix until it has the shape $|p_n.u| + k^u_n \times |p_n.v|$.
if $|c_n.u|_1 \leq |p_n.u|$**:**
we add $|p_n.u| - |c_n.u|_1$ 1s in the prefix.
Therefore, $|c'_n.u|_1 = |p_n.u| + k^u_n \times |p_n.v|$ with $k^u_n = 0$.
if $|c_n.u|_1 > |p_n.u|$**:**
we add $(|p_n.u| - |c_n.u|_1) \bmod |p_n.v|$ 1s in the prefix.
Therefore, $|c'_n.u|_1 = |p_n.u| + k^u_n \times |p_n.v|$
with $k^u_n = \left\lceil \frac{|p_n.u| - |c_n.u|_1}{|p_n.v|} \right\rceil$.
Periodic pattern case:
For all $c_n \in S$, we can construct a solution which has $k' \times |c_n.v|_1$ 1s. If we choose $k' = |p_n.v|$, we have a solution $|c'_n.v|_1 = k^v_n \times |p_n.v|$ with $k^v_n = |c_n.v|_1$.

Now, we prove that form the previous solutions S and S', we can build a solution S'' such that $\forall c''_n \in S''$

$$|c''_n.u|_1 = |p_n.u| + k \times |p_n.v| = |p'_n.u| + k \times |p'_n.v| = \dots$$

$$|c'_n.v|_1 = (k+1) \times |p_n.v| = (k+1) \times |p'_n.v| = \dots$$

where $p_n, p'_n, \dots$ are the samplers of c_n in A and where k is as integer constant.
Let $k^u = \max\{k^u_n \mid c'_n \in S'$ and $|c'_n.u|_1 = |p_n.u| + k^u_n \times |p_n.v|\}$.
Let $k^v = \operatorname{lcm}\{(k^v_n + 1) \mid c'_n \in S'$ and $|c'_n.v|_1 = k^v_n \times |p_n.v|\}$.
Let $k = \operatorname{lcm}(k^u, k^v)$.

Prefix case:
For all $c_n \in S$, since $|c_n.u|_1 \le |p_n.u| + k \times |p_n.v|$, we can increase the number of 1s of $c_n.u$ such that $|c''_n.u|_1 = |p_n.u| + k \times |p_n.v|$.

Periodic pattern case:
For all $c_n \in S$, since $(k+1)$ is a multiple of $|c_n.v|_1$, we can multiply the number of 1s of $c_n.v$ such that $|c'_n.v|_1 = (k+1) \times |p_n.v|$.

□

Finally, note that sometimes we can find solutions that allow faster execution of a system if we choose of a number of 1s different that the one proposed by Choice 1. For example, consider the following adaptability constraints:

$$\{\ c_1\ \mathit{on}\ \texttt{(1)} \quad <: \quad c_2\ \mathit{on}\ \texttt{(110)}\ \}$$

If we are seeking a solution with one 1 for c_1, we compute the solution $\{c_1 = \texttt{(10)}; c_2 = \texttt{(1011)}\}$ and execute the system one instant over two. Whereas, if we are seeking a solution with two 1s for c_1, we computes the solution $\{c_1 = \texttt{(110)}; c_2 = \texttt{(1}^6\texttt{)}\}$ and executes the system two instants over three. The parameterization of our algorithm by a constant k allows to search the best rate whereas the algorithm of Section 5 can only find the best rate for $k = 0$.

6.2 Variant 2: Polynomial Resolution Algorithm

The idea of this variant of the resolution algorithm is to modify the original algorithm such that the system of linear inequalities on the sizes and the indexes of 1s can be expressed as a *Difference Bound Matrix* [11] (*i.e.*, with constraints of the form $x - y \le n$) and thus which can be solved by the Bellman-Ford algorithm [10]. In contrast to the previous NP-complete algorithm, this approach has only polynomial complexity.

The modification of the original algorithm consists in reinforcing the adjustment conditions of the adaptability constraint system. We add the property that both sides of an adaptability constraint must have the same number of 1s in their prefixes and periodic parts. Adjusted adaptability constraint systems will be such that:

1. All the samplers of a variable have the same size.
2. For each constraint $c_x\ \mathit{on}\ p_x <: c_y\ \mathit{on}\ p_y$, $|p_x.u|_1 = |p_y.u|_1$ and $|p_x.v|_1 = |p_y.v|_1$.

This new property of the adjusted constraints allows us to simplify the synchronizability and precedence constraints.

Equation (1) becomes:

$$\begin{array}{lll} c_x\ \mathit{on}\ p_x \bowtie c_y\ \mathit{on}\ p_y & & \\ \Leftrightarrow & |p_y.v|_1 \times |c_x.v| & = |p_x.v|_1 \times |c_y.v| \\ \Leftrightarrow & |c_x.v| & = |c_y.v| \end{array} \quad (5)$$

And equation (2) becomes:

$$\begin{array}{ll} c_x\ \mathit{on}\ p_x \preceq c_y\ \mathit{on}\ p_y & \\ \Leftrightarrow & \forall j,\ 1 \le j \le h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \le \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \\ & \text{with } h = \max(|p_x.u|_1, |p_y.u|_1) \\ & \qquad + \operatorname{lcm}(|p_x.v|_1, |p_y.v|_1) \\ \Leftrightarrow & \forall j,\ 1 \le j \le h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \le \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \\ & \text{with } h = |p_x.u|_1 + |p_x.v|_1 \end{array} \quad (6)$$

First, we note that to satisfy the synchronizability constraints, we have only to choose the same size for all the c_ns. Second, thanks to the new bound h, there will be no periodicity constraints. Indeed, the set of periodicity constraints,

$$Per = \{\mathcal{I}_{c_n}(j + |c_n.v|_1) - \mathcal{I}_{c_n}(j) = |c_n.v|\}_{\substack{\mathcal{I}_{c_n}(j+|c_n.v|_1) \in Prec \\ \wedge\ j > |c_n.u|_1}}$$

is empty because it constrains the indexes of 1s of the c_ns which are greater than $|c_n.u|_1 + |c_n.v|_1$, and which are in the set of precedence constraints. There are no such elements because, since the last 1 constrained by the precedence constraints is, for each word c_n, its $(\mathcal{I}_{p_n}(|p_n.u|_1 + |p_n.v|_1))$th 1, and since $(\mathcal{I}_{p_n}(|p_n.u|_1 + |p_n.v|_1)) \le |p_n.u| + |p_n.v|$, then, thanks to the choice of the number of 1s in c_n, $(\mathcal{I}_{p_n}(|p_n.u|_1 + |p_n.v|_1))$ is less than or equal to $|c_n.u|_1 + |c_n.v|_1$.

Now, if we consider the precedence constraints, the sufficient indexes constraints and the increasing indexes constraints, we observe that they all have the form $x - y \le n$. Thus, the indexes of 1s of the c_ns can be computed by the Bellman-Ford algorithm.

We still have to solve the synchronizability constraints and the sufficient size constraints to find the size of the c_ns. It is always possible to find a solution for these constraints since we need only have choose the same size for all words and this size is the minimal value which satisfies all the sufficient size constraints.

We have presented a resolution algorithm where the part which was computationally expensive is replaced by a more efficient algorithm. This modification is possible due to some additional hypotheses on the adjusted systems (S4 in Figure 4).

A problem incurred by the new hypotheses is that to satisfy them (step from S3 to S4), we may have to increase the size of the words involved in the constraint system and thus to increase the number of linear constraints to solve. A worse problem is that the new hypotheses cannot always be satisfied. For example, we cannot adjust the following constraint system:

$$\left\{ \begin{array}{lll} c_1\ \mathit{on}\ \texttt{(1)} & <: & c_2\ \mathit{on}\ \texttt{0(1)} \\ c_2\ \mathit{on}\ \texttt{1(1)} & <: & c_1\ \mathit{on}\ \texttt{(1)} \end{array} \right\}$$

Here, if we satisfy the constraint on the number of 1s on both sides, the constraint on the size of the samplers of a same variable will be violated.

An advantage of the new hypotheses is that, if we can satisfy them, the algorithm is complete. The idea of the proof is the following. Thanks to Variant 1 (Section 6.1) of the algorithm, we know that if a system has a solution, it also has a solution corresponding to the set of constraints produced by our algorithm where the number of 1s has been increased. Then, we prove that increasing the number of 1s cannot turn an unsatisfiable system a satisfiable one. Therefore, if a system has a solution, it also has a solution with the number of 1s given by Choice 1.

Proof. To detail to poof stated above, we have to prove that if a system does not have a solution, increasing in number of 1s will not lead to a solution. Therefore, we have to consider the two cases for which a system may not have a solution: (1) the system has a synchronizability problem, (2) the system has a precedence problem.

Synchronizability problem: We proved in Section 6.1 that modifying the number of 1s in the sought words do not affect the synchronizability of a system. There, if a system has a synchronizability problem, increasing its number of 1s cannot solve it.

Precedence problem: The algorithm presented in this section treats separately the precedence and synchronizability constraints and do not have periodicity constraints. Therefore, increasing the number of 1s in the sought words only adds new precedence constraints (none of the existing constraints are suppressed). Hence, if a system has a precedence problem, increasing its number of 1s cannot solve it.

We have proved that increasing the number of 1s in a sought solution do not allows to find one if the system do not already have a solution. □

A direct consequence of this proof is that the algorithm given Section 5 is also complete if the same hypotheses on the adjusted system hold.

Hypotheses of this variant of the algorithm can always be satisfied in particular on three kind of systems: (1) systems that do not have prefixes, (2) systems where the prefixes of the samplers of a variable are made of 0s and have the same size, and (3) systems with only one constraint.

7. Comparison with other Resolution Algorithms

Three algorithms for the resolution of adaptability constraints have been proposed. The first one was proposed in [8]. It is based on the successive application of local simplification rules. The problem with this algorithm is that some systems cannot be simplified with these rules because finding a solution needs to reason globally. For example, the following system cannot be solved:

$$\left\{ \begin{array}{llll} \alpha_1 <: \alpha_2; & \alpha_2 <: \alpha_1; \\ \alpha_1 <: \alpha_3; & \alpha_3 <: \alpha_1; \\ \alpha_2 <: \alpha_3; & \alpha_3 <: \alpha_2; \end{array} \right\}$$

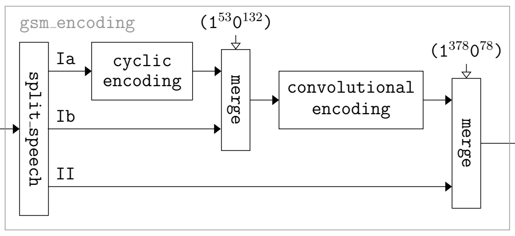
The second resolution algorithm was proposed in [16]. It is based on the abstraction of clocks by sets of clocks defined by an asymptotic rate and two shifts bounding the potential delay with respect to this rate [9]. Thanks to this abstraction, the adaptability relation can be tested by some simple operations on rational numbers. We will call this algorithm *abstract resolution algorithm*.

The third resolution algorithm is the one presented in Section 5 of this article. We will call it *concrete resolution algorithm*. In the rest of the section, we will focus on the comparison of the concrete resolution algorithm and the abstract resolution algorithm via some specific examples.

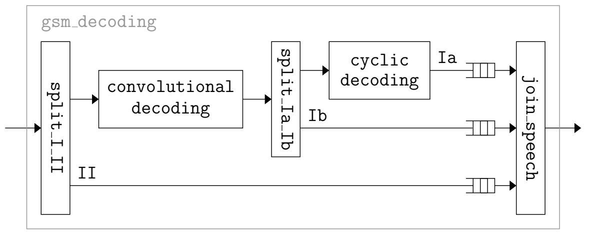
The concrete resolution algorithm allows us to type an excerpt of a GSM speech encoder/decoder. The principle of the encoder/decoder is described in Figure 5 and the source code is available at the address `http://www.lri.fr/~mandel/popl12`. This example illustrates the advantages of concrete resolution.

The encoder is depicted in Figure 5(a). The different nodes contain buffers, but they are connected without buffers. The global unification mentioned in Remark 1 is essential to type this node. It can find a rhythm for consuming the input flow such that all the constraints imposed by the processing of the three branches are satisfied.

The abstract resolution algorithm cannot type this node because it cannot treat a unification constraint as a pair of inverse subtyping constraints as proposed in Remark 1. Indeed, in this case clocks are abstracted and we thus do have not enough precise information on them to guarantee their equality. So, to type the `gsm_encoding` node with the abstract resolution algorithm, we would have to add buffers to communicate the values of the flows `Ia`, `Ib` and



(a) GSM speech encoder.



(b) GSM speech decoder.

In the GSM protocol [14], speech is divided into samples of 20ms, each of which is encoded as a frame of 260 bits. These frames are then encoded, such that it is possible to detect and sometimes to correct errors introduced by radio transmission.

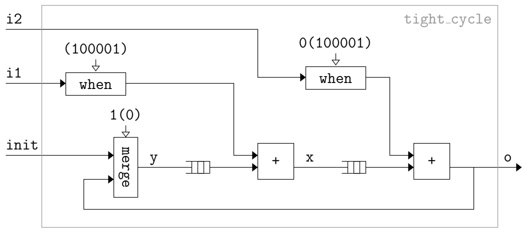
The 260 bits are divided into three classes: Ia, Ib, II. Class Ia contains the first 50 and most important bits, class Ib contains the following 132, moderately important bits and class II contains the last and least important 78 bits. The 50 bits of class Ia go through the cyclic encoder that presented in Section 2. Then the 132 bits of class Ib are appended to the 50 bits of class Ia followed by the 3 cyclic redundancy bits. The 185 bits thus obtained are encoded by a convolution encoder which builds packets of 378 bits, to which the 78 bits of class II are appended.

The decoder has the reverse behavior. It starts by separating the bits of class I from the bits of class II that have not been encoded. The bits of class I are then processed by a convolution decoder that detects and corrects some errors. Finally, a cyclic decoder is applied to the bits of class Ia to check that there are no errors (if errors remain, the bits of the preceding frame are transmitted).

Figure 5. Excerpt of the GSM speech encoder/decoder.

`II`. It transforms the unification constraints into subtyping constraints which can now be solved. With this new version of the `gsm_encoding` node, the buffer sizes estimated by abstract resolution are 50, 132 and 78, whereas the concrete resolution showed that these buffers are not necessary.

The GSM encoder example shows that the concrete resolution algorithm can handle programs that need subtle node scheduling. This advantage is also evident in programs that contain cycles with few initialization values such as the following one:



```
let node tight_cycle (init, i1, i2) = o where
  rec x = i1 when (100001) + buffer y
  and y = merge 1(0) init o
  and o = i2 when 0(100001) + buffer x
```

Here, since there is only one initial value in the cycle, the activations of the two + operators are tightly coupled: they must alternate. The abstract resolution algorithm cannot find such a schedule because, in this case, due to the lost information, it cannot guarantee safe communication through the buffers. The concrete resolution algorithm, on the other hand, finds a correct schedule.

The GSM speech decoder is depicted in Figure 5(b). Notice that the flows `Ia`, `Ib` are `II` are kept in buffers. The necessary sizes inferred for these buffers by the concrete resolution algorithm are respectively 1, 132 and 156. With the abstract resolution algorithm, the sizes are 51, 264 and 234. The estimation found by the abstract resolution algorithm gives buffers that are almost two times bigger than those found using the concrete algorithm.

This example shows that when buffers are necessary, the concrete resolution algorithm gives better estimates of buffer sizes.

Notice, however, that the abstract resolution algorithm is still interesting for some cases like, for instance, the video application *Picture in Picture* [16]. Running the concrete resolution algorithm on this example takes several days of computer time! Indeed, because the size of the clock words involved in the system are on the order of two million bits, our algorithm generates a system of linear inequalities containing numbers of variables and constraints of the same order of magnitude. Constraint systems of this size cannot be handled efficiently with tools like Glpk [13] that solve systems of linear inequalities. Finally, the algorithm with abstraction can handle systems where some words are not exactly periodic [9], *i.e* with some jitters.

One algorithm may be more appropriate than the other for a particular program. When the periodic words are well-balanced, *i.e.*, when the `1`s are regularly spread (as is the case for the nodes of the *Picture in Picture* application), the algorithm with abstraction gives good results in a short time. However, it fails when there are some constraints that are difficult to satisfy: *e.g.* those requiring global unification or those containing cycles. When words are not well balanced, *i.e.*, when the `1`s come in bursts (as in the GSM example) and they are not too long (only hundreds of elements), then the concrete algorithm is better: there is less risk of rejecting a system that has a solution, and the buffer size estimates are better.

8. Conclusion

In this article, we have presented an algorithm that computes schedules and buffer sizes for networks of ultimately periodic processes described as Lucy-n programs.

The question of scheduling and buffer sizing networks of processes is an old problem. Our particularity is to work in the context of a programming language. In that respect, the most related approaches are those of Ptolemy [12] and StreamIt [21] which are implementations of the Synchronous Data-Flow model [15]. In Ptolemy, the computation nodes are programmed in a host language and the production and consumption rate of these nodes are declared by the user. If the values declared by the user are not correct, a program will fail at run-time. The approach of Lucy-n is different, the whole program is written in a single language and the production and consumption rates are inferred automatically from the source code. StreamIt follows the same approach as Lucy-n, but provides only a small number of combinators which restricts the set of networks that can be described.

The main contribution of this paper is to define a resolution algorithm of subtyping constraints that uses all the information contained in the types. Therefore, it does not overestimate the buffer sizes.

Even if the algorithm presented in this paper is computationally more complex than the one presented in [16], our new algorithm can type some programs that would be impossible to type with the algorithm that uses abstraction. In particular, the concrete resolution algorithm has been used [17] to type programs that model latency insensitive design [3]. The types that are obtained for the different nodes of such programs define static schedules for the modeled circuit [2, 4, 7]. Our continuing work in this domain has two aspects. First, we take advantage of the results obtained for such designs (for example, to find well-balanced schedules). Second, we try to provide a method which is more compositional than those of existing approaches.

Finally, a great advantage of the concrete resolution algorithm presented in this article is that it does not restrict the trade-off between buffering and throughput. A direction for future work is to provide new language constructs to declare resource constraints and to use them to guide the resolution algorithm.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

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

[3] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Trans. on CAD of Integrated Circuits ans Systems*, 20(9):1059–1076, Sept. 2001.

[4] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky. Scheduling synchronous elastic designs. In *Int. Conf. on Application of Concurrency to System Design*, June 2009.

[5] P. Caspi and M. Pouzet. Synchronous Kahn Networks. In *International Conference on Functional Programming*, May 1996.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th symposium on Principles of programming languages*, 1987.

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

[8] 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 *International Conference on Principles of Programming Languages*, 2006.

[9] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems*, 2008.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[11] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, 1990.

[12] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003.

[13] Glpk. Gnu linear programming kit. `http://www.gnu.org/software/glpk/`.

[14] X. Lagrange, P. Godlewski, and S. Tabbane. *Réseaux GSM : des principes à la norme*. Hermès Science, Paris, 2000.

[15] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *IEEE Transactions on Computers*, 75(9), Sept. 1987.

[16] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n: a n-synchronous extension of Lustre. In *Tenth International Conference on Mathematics of Program Construction*, 2010.

[17] L. Mandel, F. Plateau, and M. Pouzet. Static scheduling of latency insensitive designs with Lucy-n. Submitted to FMCAD. `http://www.lri.fr/~mandel/fmcad11`, 2011.

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

[19] F. Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud 11, Jan. 2010.

[20] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., 1986.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.