Scheduling and Buffer Sizing of n-Synchronous Systems

Louis Mandel, Florence Plateau

Scheduling and Buffer Sizing of n-Synchronous Systems

Typing of Ultimately Periodic Clocks in Lucy-n

Louis Mandel Florence Plateau *

Laboratoire de Recherche en Informatique, Université Paris-Sud 11
Laboratoire d'Informatique de l'École Normale Supérieure, INRIA

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 finds schedules for tightly coupled systems. Moreover, it does not overestimate the buffer sizes needed and it provides a way to favor either system throughput or buffer size minimization.

1 Introduction

The *n-synchronous model* [8] is a data-flow programming model. It describes networks of processes that are executed concurrently and 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 [17] has been proposed for programming in the n-synchronous model. It is essentially Lustre [5] 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 [10]. 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 [6]. In the synchronous languages, each flow is associated with 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 a 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 ...`

* Presently at Prove & Run.

The clock calculus forces each expression to satisfy a typing constraint similar to the following ($e_1 + e_2$ is the pointwise application of the addition operator $+$):

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

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; 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 stored in a buffer of bounded size to be consumed on a clock of type ct', we say that ct is a subtype of ct', denoted $ct \ \mathbf{<:} \ ct'$:

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

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

To solve such constraints, we have to be able to unify types ($ct_1 \ \mathbf{=} \ ct_2$) and to verify the subtyping relation ($ct_1 \ \mathbf{<:} \ ct_2$). These two operations depend very much on the clock language. One especially interesting and useful clock language can be built from *ultimately periodic binary words* which comprise a finite prefix followed by an infinite repetition of a finite pattern. An algorithm to solve constraints on the types of ultimately periodic clocks is proposed in [17]. The algorithm exploits clock abstraction [9] where the exact "shape" of clocks is forgotten in favor of simpler specifications of the presence instants of the flows: their asymptotic rate and two offsets bounding the potential delay with respect to this rate.

Type constraints on abstract clocks can be solved 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 the abstraction. Therefore, when 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.

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

A program (d) is a sequence of node and clock definitions.

d ::=	`let node` $f(pat) = e$	node definition
	| `let clock` $c = ce$	clock definition
	| d d	sequence of definitions

A pattern (pat) can be a variable or a tuple.

pat ::=	x | `(`pat`,...,`pat`)`	pattern

The body of a node is defined by an expression (e).

e ::=	i	constant flow
	| x	flow variable
	| `(`e`,...,`e`)`	tuple
	| e op e	imported operator
	| `if` e `then` e `else` e	mux operator
	| f e	node application
	| e `where rec` eqs	local definitions
	| e `fby` e	initialized delay
	| e `when` ce | e `whenot` ce	sampling
	| `merge` ce e e	merging
	| `buffer`(e)	buffering
eqs ::=	$pat = e$ | eqs `and` eqs	mutually recursive equations

Clock expressions (ce) are either clock names or ultimately periodic words.

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

Fig. 1. The Lucy-n kernel.

Section 2 and 3 we present the Lucy-n language and its clock calculus by way of an extended example. Section 4 introduces the properties used in Section 5, which presents an algorithm for resolving constraints. Section 6 discusses results obtained on examples and compares them with previous resolution algorithms. Finally, Section 7 concludes the article.

An extended version of the article [16] with additional details and proofs, the code of the examples, a commented implementation of the algorithm and the Lucy-n compiler are available at `http://www.lri.fr/~mandel/mpc12`.[2]

2 The Lucy-n Language

The kernel of the Lucy-n language is summarized in Figure 1. In this section, we present the language through the programming of a GSM voice encoder component. This component is a cyclic encoder. It takes as input a flow of bits

[2] While the present paper is based on [15], it presents some new results. In particular, we generalize the first version of the algorithm, which allows us to define both a semi-decidable and complete algorithm and a decidable algorithm which is complete on a well defined class of systems. Finally, we also explain how to favor either system throughput or buffer size minimization.

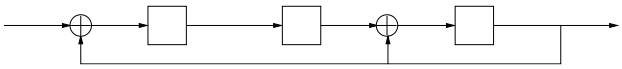


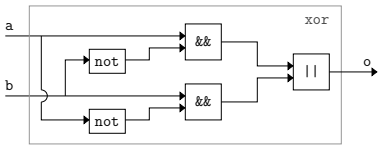
Fig. 2. Circuit for division [19] by $X^3 + X + 1$. The input flow is the sequence of fifty coefficients of the polynomial to divide. After consuming the fiftieth input bit, all the coefficients of the quotient polynomial have been produced at the output and the registers contain the coefficients of the remainder polynomial.

representing voice samples and produces an output flow that contains 3 new redundancy bits after every 50 data bits. The redundancy bits are the coefficients of the remainder of the division of the 50 bits to encode, considered as a polynomial of degree 49, by a polynomial peculiar to the encoder, here $X^3 + X + 1$.

The classical circuit to divide a polynomial is shown in Figure 2; 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 (corresponding block diagrams are shown to the right of code samples):

```
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
```

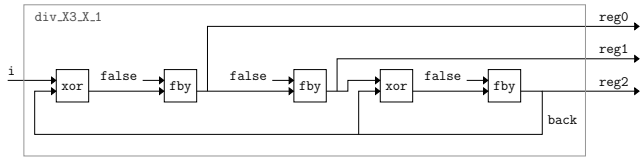


The node `xor` takes as input two flows `a` and `b` and computes the value of the output flow `o`. The value of `o` is defined by the equation `o = (a && (not b)) || (b && (not a))` where the scalar operators `&&`, `||` and `not` are applied pointwise to their input flows. Hence, if we apply the node `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 to the 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 2.




```
let node div_X3_X_1 i = (reg0,reg1,reg2) where
  rec reg0 = false fby (xor(i, back))
  and reg1 = false fby reg0
  and reg2 = false fby (xor(reg1, back))
  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 `reg1 = false fby reg0` means that `reg1` is equal to `false` at the first instant and equal to the preceding value of `reg0` 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 some 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 `1`s and 0^3 the concatenation of three `0`s. To understand the type of `div_X3_X_1`, notice that *'a* (the activation rhythm of the node) 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*. Finally, the clock type of the three outputs is *'a*, it means that the values of the registers are produced at each instant.

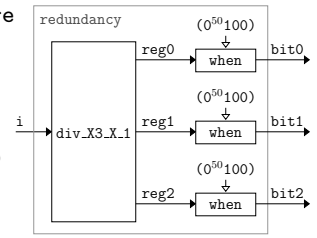
Now, to define a node `redundancy` which computes only the redundancy bits corresponding to a flow of polynomials, we sample the output of the node `div_X3_X_1`. In our implementation of the node `div_X3_X_1`, the remainder of the division is contained in the registers after the 50th input bit and output 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 filters the values of the flow following the pattern defined by the sampler: if the flow is absent, the output of the `when` is absent; if the input flow is present and the next element of the sampler is `1`, the

[3] It is implicit, here and in the following, that such behaviors iterate repeatedly.

[4] The source code of the modified node is available at `http://www.lri.fr/~mandel/mpc12/gsm.ls.html`.

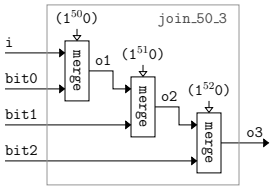
value of the flow is output; if the input flow is present and the next element of the sampler is 0, the output of the `when` is absent. To keep only the 51st element of a sequence of 53 bits, we use the sampling condition ($0^{50}100$):

```
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 ($1^{50}0$) and obtain a flow of 51 bits. Then, we use the condition ($1^{51}0$) to incorporate the second redundancy bit, and finally the condition ($1^{52}0$) 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
```

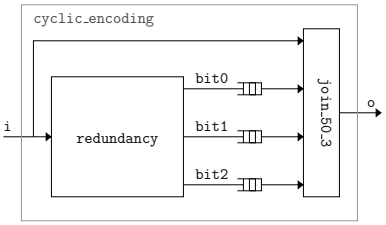


We will see in Section 4 that the clock type of `join_50_3` is equivalent to:
$\forall\alpha.\,(\alpha$ `on` $(1^{50}000) \times \alpha$ `on` $(0^{50}100) \times \alpha$ `on` $(0^{50}010) \times \alpha$ `on` $(0^{50}001)) \rightarrow \alpha$
This type expresses that the flow containing data must be present for the first 50 instants, and then absent for the following 3 instants. The flows containing the first, second and third redundancy bits must arrive at the 51st, 52nd, and 53rd instants respectively.

To complete the cyclic encoder, we must 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 at instant 51 which is too early for the `join_50_3` node which expects them successively at instants 51, 52 and 53. 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
```

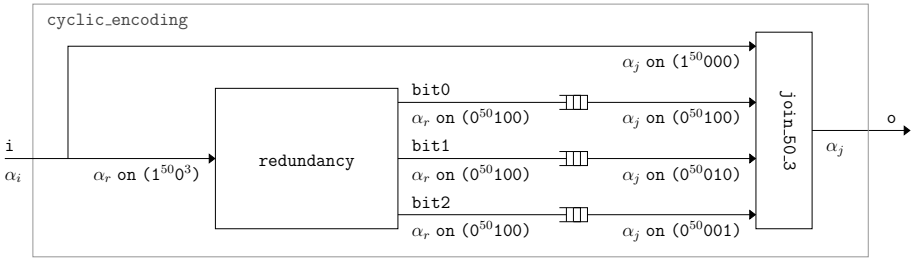


The compiler automatically computes the buffer sizes required. We can see that the buffer at line 41 is not really needed, the inferred size is 0. This buffer is used for the communication of the first redundancy bit (`bit0`) between the `redundancy` node and the `join_50_3` node. This bit is produced 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 the outputs of the first node are produced at the very same instants that they are to be consumed as inputs by the second node. 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 (the rules of the clock calculus are detailed in annex A). To illustrate the typing inference algorithm which collects the constraints, we return to the `cyclic_encoding` node of the previous section.



If we associate with the input `i` the clock type variable α_i, the expression `redundancy i` generates the equality constraint α_i `=` α_r `on` $(1^{50}0^3)$. Indeed,

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

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 $(\alpha_j$ `on` $(1^{50}000) \times \alpha_j$ `on` $(0^{50}100) \times \alpha_j$ `on` $(0^{50}010) \times \alpha_j$ `on` $(0^{50}001)) \to \alpha_j$. 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 α_j `on` $(1^{50}000)$ and the types of the other inputs (here α_r `on` $(0^{50}100)$) must be, respectively, subtypes of α_j `on` $(0^{50}100)$, α_j `on` $(0^{50}010)$ and α_j `on` $(0^{50}001)$. For these last inputs, we do not impose type equality but rather only subtyping (**<:**) since they are consumed through buffers. The subtyping relation ensures that there are neither reads in an empty buffer nor writes in a full buffer. 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 $\alpha_i \to \alpha_j$, with the following constraints:

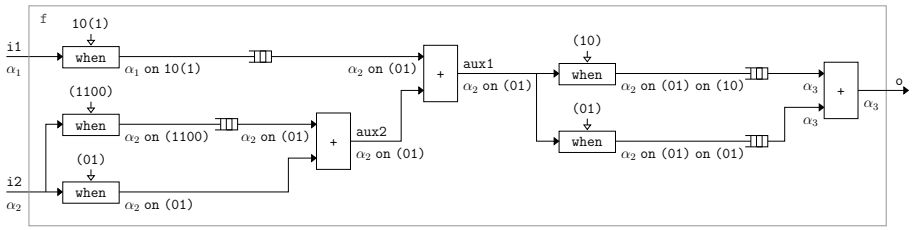
$$C = \left\{ \begin{array}{rcl} \alpha_i & \texttt{=} & \alpha_r \texttt{ on } (1^{50}0^3) \\ \alpha_i & = & \alpha_j \texttt{ on } (1^{50}0^3) \\ \alpha_r \texttt{ on } (0^{50}100) & \texttt{<:} & \alpha_j \texttt{ on } (0^{50}100) \\ \alpha_r \texttt{ on } (0^{50}100) & \texttt{<:} & \alpha_j \texttt{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

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: $ct ::= \alpha \mid (ct$ `on` $p)$ where p is an ultimately periodic binary word (formally defined in Section 4.1).

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

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

Remark 1. Notice that there is no complete greedy unification algorithm because there is no most general unifier for clock types [21]. Therefore, to be complete, a resolution algorithm must take into account all the constraints globally. As in this example greedy structural unification leads to a solution, we used it for the sake of conciseness. In the general case, 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)).




```
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))
```

Fig. 3. The node `f` and its block diagram representation. The diagram is annotated with the types obtained after the resolution of equality constraints.

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 [17] to simplify the `on` operators:

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

We will see in Section 5.1 that these constraints on words can be checked. Sometimes however, subtyping constraints are not expressed with respect to the same type variable. For example, the program of Figure 3 generates the following set of subtyping constraints where only the second constraint can be simplified:

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

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* denoted c_n and we replace each type variable α_n with α `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:

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

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

This succession of operations transforms 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 in Section 5. Proofs are provided in the extended version of the article [16].

4.1 Ultimately Periodic Binary Words

We write w for an infinite binary word ($w ::= \mathtt{0}w \mid \mathtt{1}w$), u or v for finite binary words ($u, v ::= \mathtt{0}u \mid \mathtt{1}u \mid \varepsilon$), $|u|$ for the size of u and $|u|_{\mathtt{1}}$ 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 binary word that contains infinitely many `1`s.

$$\begin{array}{rll} \mathcal{I}_w(0) & \stackrel{def}{=} 0 & \\ \mathcal{I}_w(1) & \stackrel{def}{=} 1 & \text{if } w = \mathtt{1}w' \\ \forall j > 1,\ \mathcal{I}_w(j) & \stackrel{def}{=} 1 + \mathcal{I}_{w'}(j-1) & \text{if } w = \mathtt{1}w' \\ \forall j > 0,\ \mathcal{I}_w(j) & \stackrel{def}{=} 1 + \mathcal{I}_{w'}(j) & \text{if } w = \mathtt{0}w' \end{array}$$

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

Remark 2 (increasing indexes). $\mathcal{I}_w$ 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 counts the number of `1`s since the beginning of w.

Definition 2 (cumulative function of w: $\mathcal{O}_w$).[5]

$$\mathcal{O}_w(0) \stackrel{def}{=} 0 \qquad \forall i \geq 1,\ \mathcal{O}_w(i) \stackrel{def}{=} \sum_{0 \leq i' \leq i} w[i']$$

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

Definition 3 (ultimately periodic word). $p = u\mathtt{(}v\mathtt{)} \stackrel{def}{\Leftrightarrow} p = uw$ *with* $w = vw$

For example, $p = \mathtt{1101(110)} = \mathtt{1101\ 110\ 110\ 110}\ \ldots$ We use the notation $p.u$ for the prefix of a word p (e.g. $(\mathtt{1101(110)}).u = \mathtt{1101}$) and $p.v$ for its periodic pattern (e.g. $(\mathtt{1101(110)}).v = \mathtt{110}$).

An ultimately periodic binary word has an infinite number of different representations. For example, $\mathtt{(10)} = \mathtt{(1010)} = \mathtt{1(01)} = \ldots$. But, there exists a normal form which is the representation where the prefix and the periodic

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

pattern have the shortest size. For some binary operations, however, 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 1s, or even that the number of 1s in the first word is equal to the size of the second word.

Remark 4. We can change the shape of an ultimately periodic binary word with the following manipulations:

- Increase prefix size: $u(vv') = uv(v'v)$. 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 1s it contains.
- Repeat periodic pattern: $u(v) = u(v^k)$ with $k > 0$. For example, we can triple the size of the periodic pattern of $p = 1101(110)$ (and thus triple its number of 1s) to obtain $1101(110\ 110\ 110)$.

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

Remark 5 (periodicity). Two successive occurrences of the same 1 of a periodic pattern are separated by a distance equal to the size of the pattern:

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

As a direct consequence of this property, the distance between any 1 in a repetition of a periodic pattern and the corresponding 1 in the first occurrence of the pattern is a multiple of the size of the pattern.

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

For example, if $p = 101(10010)$, $\mathcal{I}_p(3+2) = \mathcal{I}_p(3) + 5$ and $\mathcal{I}_p(4 + 2 \times 2) = \mathcal{I}_p(3) + 2 \times 5$.

Remark 6 (sufficient size). The size of the periodic pattern of a word p (i.e., $|p.v|$) is greater than or equal to the number of elements between the indexes of the first and last 1 of the periodic pattern of p (for words with at least one 1 in the periodic pattern):

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

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

The rate of a word w is the proportion of 1s in the word w:

Definition 4 (rate of p). $rate(w) = \lim_{i \to +\infty} \frac{\mathcal{O}_w(i)}{i}$

For an ultimately periodic binary word p, the rate is the ratio between the number of 1s and the size of its periodic pattern:

Proposition 1 (rate of p). $rate(p) = \frac{|p.v|_1}{|p.v|}$

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 1s and are the clocks of flows that produce values infinitely often).

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 finite bound in the buffer, and that reads are not attempted when the buffer is empty. The adaptability relation is the conjunction of *precedence* and *synchronizability* relations. The synchronizability relation between two words w_1 and w_2 (written $w_1 \bowtie w_2$) asserts that there is a finite upper bound on the number of values present in the buffer during an execution. It states 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 $w_1 \preceq w_2$) asserts the absence of reads from an empty buffer. It states 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 $\bowtie$).

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

To test this synchronizability relation on ultimately periodic binary words, we have only 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 2 (synchronizability test). $p_1 \bowtie p_2 \ \Leftrightarrow\ rate(p_1) = rate(p_2)$

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 a "common" periodic behavior is reached.[6] For example, `1(1100)` $\preceq$ `(110100)` because $\mathcal{I}_{\texttt{1(1100)}}(j) \leq \mathcal{I}_{\texttt{(110100)}}(j)$ 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 periodic patterns within these two words recommence simultaneously.

Proposition 3 (precedence test). *Consider p_1 and p_2 such that $p_1 \bowtie p_2$. Let $h = \max(|p_1.u|_1, |p_2.u|_1) + \mathrm{lcm}(|p_1.v|_1, |p_2.v|_1)$. Then:*

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

The intuition for the value of the bound h is the following. By Remark 4, 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

[6] A common periodic behavior of two ultimately periodic words p_1 and p_2 is defined by an index h and a size k such that $\forall j > h,\ \mathcal{I}_{p_2}(j) - \mathcal{I}_{p_1}(j) = \mathcal{I}_{p_2}(j-k) - \mathcal{I}_{p_1}(j-k)$.

number of 1s in their prefixes is $\max(|p_1.u|_1, |p_2.u|_1)$ and the number of 1s in their periodic pattern is $\mathrm{lcm}(|p_1.v|_1, |p_2.v|_1)$. Hence, after the traversal of h 1s in p'_1 and p'_2 (with $h = |p'_1.u|_1 + |p'_1.v|_1 = |p'_2.u|_1 + |p'_2.v|_1$), 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 holds.

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.

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 any execution.

Definition 8 (buffer size). $size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$

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

Proposition 4 (buffer size).
Consider p_1 *and* p_2 *such that* $p_1 <: p_2$.
Let $H = \max(|p_1.u|, |p_2.u|) + \mathrm{lcm}(|p_1.v|, |p_2.v|)$. *Then:*

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

Note that the bound H is not the same as the one of Proposition 3, because here we iterate over indexes (not over 1s).

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 `w2`:

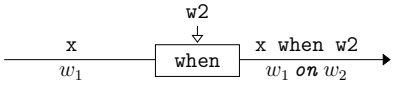


Fig. 4. If `x` has clock w_1, `x when w2` has clock w_1 *on* w_2.

Definition 9 (*on* operator).

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

For example, if $w_1 = 11010111...$ and $w_2 = 101100...$, then $w_1 \;\mathit{on}\; w_2 = 10010100...$
Consider the sampling of a flow `x` with clock w_1 by a condition `w2`:

`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 ...
$w_1 \;\mathit{on}\; w_2$	1 0 0 1 0 1 0 0 ...

At each instant, if `x` is present, that is, the corresponding element of w_1 is equal to `1`, the next element of the sampling condition w_2 is considered. If this element is `1`, then the value of `x` is sampled and the flow `x when w2` is present ($w_1 \;\mathit{on}\; w_2$ equals `1`). If the element is `0`, the value of `x` is not sampled and the flow `x when w2` is absent ($w_1 \;\mathit{on}\; w_2$ equals `0`). If `x` is absent (w_1 equals `0`), the sampling condition w_2 is not considered, and the flow `x when w2` will be absent ($w_1 \;\mathit{on}\; w_2$ equals `0`).

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., $|(p_1 \;\mathit{on}\; p_2).u|$, the size of the prefix, and $|(p_1 \;\mathit{on}\; p_2).v|$, the size of the periodic part. Then, we compute the value of the elements of the prefix and the periodic part by applying Definition 9.

Proposition 5 (computation of $p_1 \;\mathit{on}\; p_2$). *Let $p_1 = u_1(v_1)$ and $p_2 = u_2(v_2)$. Then $p_1 \;\mathit{on}\; p_2 = u_3(v_3)$ with:* $|u_3| = \max(|u_1|, \mathcal{I}_{p_1}(|u_2|))$

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

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

Intuitively, the prefix of $p_1 \;\mathit{on}\; p_2$ is obtained after completely processing the prefixes of p_1 and p_2. One element of p_1 is processed to produce one element of $p_1 \;\mathit{on}\; p_2$. Thus, the processing of the elements of $p_1.u$ for the computation of the *on* terminates at the index $|p_1.u|$. An element of p_2 is processed only when there is a `1` in p_1. Thus the processing of the elements of $p_2.u$ terminates at the index $\mathcal{I}_{p_1}(|p_2.u|)$. Therefore, the size of the prefix of $p_1 \;\mathit{on}\; p_2$ is the maximum of $|p_1.u|$ and $\mathcal{I}_{p_1}(|p_2.u|)$. 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.

The result of the *on* operation can be computed more simply for certain shapes of arguments. The simplest case is the one where the number of `1`s in the prefix and in the periodic pattern of the first word are, respectively, equal 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		)
$p_1 \;\mathit{on}\; p_2$	1	0	0	1	(	1	0	0	0	0	1	0	0	)

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

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

As explained in Remark 4, 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 $\texttt{on}$ such that they satisfy the assumptions of Proposition 6. Proposition 6 can be generalized to the case where the number of 1s of p_1 is increased by any multiple of the size of $p_2.v$.

Proposition 7. *Consider p_1 and p_2 such that $|p_1.u|_1 = |p_2.u| + k \times |p_2.v|$ and $|p_1.v|_1 = k' \times |p_2.v|$ with $k \in \mathbb{N}$ and $k' \in \mathbb{N} - \{0\}$. Then:*

$$|(p_1 \mathbin{\texttt{on}} p_2).u| = |p_1.u| \qquad |(p_1 \mathbin{\texttt{on}} p_2).u|_1 = |p_2.u|_1 + k \times |p_2.v|_1$$
$$|(p_1 \mathbin{\texttt{on}} p_2).v| = |p_1.v| \qquad |(p_1 \mathbin{\texttt{on}} p_2).v|_1 = k' \times |p_2.v|_1$$

Finally, to compute the index of the jth 1 of $w_1 \mathbin{\texttt{on}} w_2$ ($\mathcal{I}_{w_1 \mathbin{\texttt{on}} w_2}(j)$), there is no need to compute the word $w_1 \mathbin{\texttt{on}} w_2$ and then to apply the $\mathcal{I}$ function since it can be computed directly from $\mathcal{I}_{w_1}$ and $\mathcal{I}_{w_2}$.

Proposition 8 (index of the jth 1 of $w_1 \mathbin{\texttt{on}} w_2$).

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

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

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 5.[7] 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 adaptability constraints have the form $c_x \mathbin{\texttt{on}} p_x <: c_y \mathbin{\texttt{on}} p_y$. In Section 5.2, we explain the transformation of adaptability constraints (S3) into a system (S6) of linear inequalities 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

[7] A detailed and commented implementation of the algorithm in OCaml is provided at `http://www.lri.fr/~mandel/mpc12`.

S1. Subtyping constraints: $\alpha_x \texttt{ on } p_1 \texttt{ on } ... \texttt{ <: } \alpha_y \texttt{ on } p_2 \texttt{ on } ...$

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

S2. Adaptability constraints: $p_1 \texttt{ on } ... <: p_2 \texttt{ on } ...$

$\Leftrightarrow$ { computation of $\texttt{on}$; $c_x \texttt{ on } p'_1 \texttt{ on } ... <: c_y \texttt{ on } p'_2 \texttt{ on } ...$
simplification of $p_1 <: p_2$ constraints }

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

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

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

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

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

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

S6. Indexes of $\texttt{1}$s 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:	$\mathcal{I}_{c_n}(j + l \times \lvert c_n.v\rvert_1) - \mathcal{I}_{c_n}(j) = l \times \lvert c_n.v\rvert$
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:	$\mathcal{I}_{c_n}(j') - \mathcal{I}_{c_n}(j) \geq j' - j$

Fig. 5. Summary of the subtyping constraints resolution algorithm. The form of the constraints is given for each system.

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, if the words (c_1, c_2) satisfy a constraint $c_1 \texttt{ on } p_1 <: c_2 \texttt{ on } p_2$, then $(\texttt{1}^d c_1, \texttt{1}^d c_2)$ and $(\texttt{0}^d c_1, \texttt{0}^d c_2)$ also satisfy it whatever d is. 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

We begin by considering adaptability constraint systems. After the computation of $\texttt{on}$ operators (Proposition 5), these systems comprise constraints of the form $p_x <: p_y$ and $c_x \texttt{ on } p_x <: c_y \texttt{ on } p_y$, where p_x and p_y are known words of non-null rate and c_x and c_y are unknown words.

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 by 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:

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

Through the application of the adaptability test, we can check that each constraint is always satisfied. Here, after simplification, the system is empty and thus the node `cyclic_encoding` is well typed.

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':

$$\theta(C') \Leftrightarrow \left\{ \begin{array}{r} c_1 \ \mathtt{on}\ \mathtt{10(1)} <: c_2 \ \mathtt{on}\ \mathtt{(01)} \\ \mathtt{(1100)} <: \mathtt{(01)} \\ c_2 \ \mathtt{on}\ \mathtt{(01)}\ \mathtt{on}\ \mathtt{(10)} <: c_3 \ \mathtt{on}\ \mathtt{(1)} \\ c_2 \ \mathtt{on}\ \mathtt{(01)}\ \mathtt{on}\ \mathtt{(01)} <: c_3 \ \mathtt{on}\ \mathtt{(1)} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{l} c_1 \ \mathtt{on}\ \mathtt{10(1)} <: c_2 \ \mathtt{on}\ \mathtt{(01)} \\ c_2 \ \mathtt{on}\ \mathtt{(0100)} <: c_3 \ \mathtt{on}\ \mathtt{(1)} \\ c_2 \ \mathtt{on}\ \mathtt{(0001)} <: c_3 \ \mathtt{on}\ \mathtt{(1)} \end{array} \right\} = A'$$

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

5.2 Constraint System Solving

The goal now is to solve adaptability constraint systems of the form:

$$\{c_{x_i} \ \mathtt{on}\ p_{x_i} <: c_{y_i} \ \mathtt{on}\ p_{y_i}\}_{i=1..\text{number of constraints}}$$

The values p_{x_i}, p_{y_i} are some known ultimately periodic binary words of non-null rate. The variables c_{x_i}, c_{y_i} are the unknowns of the system. Note that some unknown variables can appear several times in a system like in A' (we only know that $c_{x_i} \neq c_{y_i}$ since simplification has been performed):[8]

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

Solving the system means associating ultimately periodic words of non-null rate to the unknowns (c_1, c_2, c_3), such that the constraints are satisfied.

Remark 7. 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 $c_x \ \mathtt{on}\ p_x <: c_y \ \mathtt{on}\ p_y$ can be decomposed into a synchronizability constraint and a precedence constraint:

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

[8] We choose the same index for a variable c_n and the words p_n, p'_n, ... that sample c_n.

The synchronizability constraint can itself be rewritten:

$$c_x \,\texttt{on}\, p_x \bowtie c_y \,\texttt{on}\, p_y \Leftrightarrow \{ \text{ by Proposition 2 and Proposition 1 } \}$$
$$\frac{|(c_x \,\texttt{on}\, p_x).v|_1}{|(c_x \,\texttt{on}\, p_x).v|} = \frac{|(c_y \,\texttt{on}\, p_y).v|_1}{|(c_y \,\texttt{on}\, p_y).v|}$$

As can the precedence constraint:

$$c_x \,\texttt{on}\, p_x \preceq c_y \,\texttt{on}\, p_y \Leftrightarrow \{ \text{ by Proposition 3 and Proposition 8 } \}$$
$$\forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$$
$$\text{with } h = \max(|(c_x \,\texttt{on}\, p_x).u|_1, |(c_y \,\texttt{on}\, p_y).u|_1) + \text{lcm}(|(c_x \,\texttt{on}\, p_x).v|_1, |(c_y \,\texttt{on}\, p_y).v|_1)$$

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

We have seen in Section 4 (Proposition 7) that the size and number of 1s in the prefix and in the periodic pattern of $c_n \,\texttt{on}\, p_n$ can easily be expressed as a function of the size and number of 1s in the prefixes and periodic patterns of c_n and p_n in the following case: $|c_n.u|_1 = |p_n.u| + k \times |p_n.v|$ and $|c_n.v|_1 = k' \times |p_n.v|$.

To put the system into this "simple" form, we adjust the known words of the system to satisfy the property: for any unknown c_n, all its samplers[9] p_n, p'_n, ... 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 4) and does not change the semantics of the system.

We can then choose the number of 1s in the unknown c_n.

Choice 1 (number of 1s in the c_n) *Let $k \in \mathbb{N}$ and $k' \in \mathbb{N} - \{0\}$.*

$$\begin{array}{lcllll} |c_n.u|_1 & = & |p_n.u| + k \times |p_n.v| & (= & |p'_n.u| + k \times |p'_n.v| & = \ \ldots) \\ |c_n.v|_1 & = & k' \times |p_n.v| & (= & k' \times |p'_n.v| & = \ \ldots) \end{array}$$

where p_n, p'_n, ... are the samplers of c_n.

Remark 8. The algorithm is parameterized by constants k and k', which restrict the number of 1s in any solution and may thus lead to failures in the resolution of constraints. We will discuss this choice in Section 5.4.

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

$$c_x \,\texttt{on}\, p_x \bowtie c_y \,\texttt{on}\, p_y \Leftrightarrow \{ \text{ by Proposition 7 } \}$$
$$\frac{k' \times |p_x.v|_1}{|c_x.v|} = \frac{k' \times |p_y.v|_1}{|c_y.v|}$$
$$\Leftrightarrow |p_y.v|_1 \times |c_x.v| \ = |p_x.v|_1 \times |c_y.v| \tag{1}$$

And a precedence constraint becomes:

$$c_x \,\texttt{on}\, p_x \preceq c_y \,\texttt{on}\, p_y \Leftrightarrow \{ \text{ by Proposition 7 } \}$$
$$\forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j))$$
$$\text{with } h = \max(|p_x.u|_1 + k \times |p_x.v|_1, |p_y.u|_1 + k \times |p_y.v|_1) + \text{lcm}(k' \times |p_x.v|_1, k' \times |p_y.v|_1) \tag{2}$$

[9] A word p_n is a *sampler* of c_n if $c_n \,\texttt{on}\, p_n$ is in the constraint system.

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

$$A' = \left\{ \begin{array}{ll} c_1 \texttt{ on } \texttt{10(1)} & <: c_2 \texttt{ on } \texttt{(01)} \\ c_2 \texttt{ on } \texttt{(0100)} & <: c_3 \texttt{ on } \texttt{(1)} \\ c_2 \texttt{ on } \texttt{(0001)} & <: c_3 \texttt{ on } \texttt{(1)} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{ll} c_1 \texttt{ on } \texttt{10(1)} & <: c_2 \texttt{ on } \texttt{(0101)} \\ c_2 \texttt{ on } \texttt{(0100)} & <: c_3 \texttt{ on } \texttt{(1)} \\ c_2 \texttt{ on } \texttt{(0001)} & <: c_3 \texttt{ on } \texttt{(1)} \end{array} \right\}$$

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

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

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

$$\left\{ \begin{array}{r} |\texttt{(0101)}.v|_1 \times |c_1.v| = |\texttt{(10(1))}.v|_1 \times |c_2.v| \\ |\texttt{(1)}.v|_1 \times |c_2.v| = |\texttt{(0100)}.v|_1 \times |c_3.v| \\ |\texttt{(1)}.v|_1 \times |c_2.v| = |\texttt{(0001)}.v|_1 \times |c_3.v| \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{r} 2 \times |c_1.v| = |c_2.v| \\ |c_2.v| = |c_3.v| \\ |c_2.v| = |c_3.v| \end{array} \right\} \qquad (Sync)$$

By Formula (2), if we choose the constants k and k' to be equal to 0 and 1, the precedence constraints become a system of linear inequalities on the indexes of 1s in the c_ns:

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

which is equivalent to the following system after the computation of the $\mathcal{I}_{p_n}(j)$:

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

Now, to give a value to each unknown c_n, we must find its size (satisfying *Sync*) and the positions of its 1s (satisfying *Prec*). 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, 3, 5 and 6 which ensure that the solution will be a well formed ultimately periodic binary word. So, we have to augment *Sync* and *Prec* with the following four sets of constraints:[10]

Periodicity: $Per = \{\mathcal{I}_{c_n}(j + l \times |c_n.v|_1) - \mathcal{I}_{c_n}(j) = l \times |c_n.v|\}_{\substack{\mathcal{I}_{c_n}(j+l\times|c_n.v|_1) \in Prec \\ \wedge\ |p.u|_1 < j \leq |p.u|_1 + |p.v|_1}}$

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

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

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

Finally, we can use a generic solver for Integer Linear Programming (ILP) problems to solve the system: $S = Sync \cup Prec \cup Per \cup Size \cup Init \cup Incr$.

[10] The notation $\mathcal{I}_{c_n}(j) \in S$ designates the presence of the unknown $\mathcal{I}_{c_n}(j)$ in S.

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

$|c_1.v| = 2 \quad \mathcal{I}_{c_1}(1) = 1 \quad \mathcal{I}_{c_1}(3) = 3 \quad \mathcal{I}_{c_1}(4) = 5$

$|c_2.v| = 4 \quad \mathcal{I}_{c_2}(1) = 1 \quad \mathcal{I}_{c_2}(2) = 2 \quad \mathcal{I}_{c_2}(4) = 4 \quad \mathcal{I}_{c_2}(6) = 6$

$|c_3.v| = 4 \quad \mathcal{I}_{c_3}(1) = 4$

Thanks to this information and the number of 1s in the prefixes and periodic patterns of the c_ns chosen previously, we can build the following solution to the A' system: $c_1 = \texttt{11(10)} \quad c_2 = \texttt{(1111)} = \texttt{(1)} \quad c_3 = \texttt{000(1000)} = \texttt{(0}^3\texttt{1)}$.

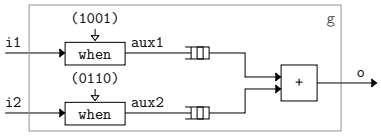
We now know all the clock types of the system of Figure 3. The result gives us the clock type of the node `f` which is $\forall\alpha,\ \alpha\ \texttt{on}\ c_1 \times \alpha\ \texttt{on}\ c_2 \rightarrow \alpha\ \texttt{on}\ c_3$, that is: $\texttt{f} :: \forall\alpha,\ \alpha\ \texttt{on 11(10)} \times \alpha\ \texttt{on (1)} \rightarrow \alpha\ \texttt{on (0}^3\texttt{1)}$.

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 $\alpha\ \texttt{on}\ c_1\ \texttt{on 10(1)} = \alpha\ \texttt{on 11(10) on 10(1)}$ and that the reading clock is of type $\alpha\ \texttt{on}\ c_2\ \texttt{on (01)} = \alpha\ \texttt{on (1) on (01)}$. By Proposition 4, 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 1s 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 1s 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 assign the type $\forall\alpha.\,(\alpha \times \alpha) \rightarrow \alpha\ \texttt{on (01)}$ to the node `g`, the buffers will be of size 1. But, this node can be executed without buffers if we give 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 1s in the solution is fixed, minimizing the size increases the rate.

The second solution is obtained by an objective function that, for all precedence constraints $\mathcal{I}_{c_x}(j_1) \leq \mathcal{I}_{c_y}(j_2)$, minimizes 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 5 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 the equivalence properties stated in Section 4). The step from S4 to S5 is the choice of the number of 1s 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.

We have parameterized our resolution algorithm by two constants k and k' which modify the number of 1s in the sought solution. A semi-decidable algorithm to solve adaptability constraints iterates the previous algorithm with $k = 0, 1, 2, \ldots$ and $k' = k + 1$ 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 $\forall c'_n \in S'$,

$$\begin{array}{lcl} |c'_n.u|_1 & = & |p_n.u| + k \times |p_n.v| \quad (= \; |p'_n.u| + k \times |p'_n.v| \quad = \quad \ldots) \\ |c'_n.v|_1 & = & (k+1) \times |p_n.v| \qquad (= \; (k+1) \times |p'_n.v| \qquad = \quad \ldots) \end{array}$$

where k can be computed from the original solution S. The idea of the proof is to use Remark 4 to rewrite S into S' (the detailed proof is in the extended version of the paper).

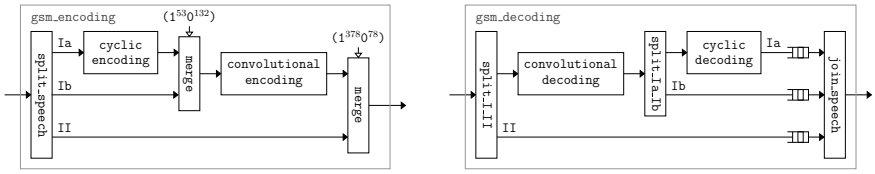
Moreover, note that the step that equalizes of the size of the samplers (from S3 to S4, Figure 5) can be adapted such that the choice $k = 0$ and $k = 1$ always leads to a solution, if it exists, for (1) systems that do not have prefixes, (2) systems where the prefixes of the samplers of a variable are made of 0s and have the same size, and (3) systems with only one constraint. The algorithm is given in the extended version of the paper.

Remark that we can sometimes find solutions that allow faster execution of a system if we choose a number of 1s different than the one proposed by $k = 0$ and $k' = 1$. For example, consider the following adaptability constraints:

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

If we are seeking a solution with one 1 for c_1, we compute the solution $\{c_1 = \texttt{(10)};\ c_2 = \texttt{(1011)}\}$ where $rate(c_1) = \frac{1}{2}$ and $rate(c_2) = \frac{3}{4}$. Whereas, if we are seeking a solution with two 1s for c_1, we compute the solution $\{c_1 = \texttt{(110)};\ c_2 = \texttt{(1}^6\texttt{)}\}$ where $rate(c_1) = \frac{2}{3}$ and $rate(c_2) = 1$. The guarantee provided by the resolution algorithm is that for a given number of 1s, the result is optimal with respect to the objective function given to the ILP solver. It follows the fact that each transformation of the adaptability constraint system, except Choice 1, maintains equivalence. Therefore, there is no loss of information.

The complexity of the resolution algorithm is dominated by the resolution of the constraint system on the indexes of 1s and the sizes. This is an 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 (e.g., millions of variables): it depends on the size of the samplers in the adaptability constraint system.



(a) GSM speech encoder. (b) GSM speech decoder.

In the GSM protocol [13], 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 is 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).

Fig. 6. Excerpt of the GSM speech encoder/decoder.

6 Comparison with Previous Resolution Algorithms

Three algorithms for the resolution of adaptability constraints have been proposed. The first one [8] is based on the successive application of local simplification rules. This algorithm does not always succeed because some systems can only be simplified globally, that is, by resolving all of their constraints simultaneously (one such example is given in the long version of this article).

A second algorithm [17], the *abstract resolution algorithm*, is based on the abstraction of clocks by sets of clocks defined by an asymptotic rate and two offsets 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.

Section 5 of this article presents the third resolution algorithm, the *concrete resolution algorithm*. Technically, the first steps of the algorithm (from S1 to S3, Figure 5) are similar to the ones of the abstract resolution algorithm. The subsequent steps that solve the adaptability constraints (from S3 to S6) differ. 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 6 and the source code is available at `http://www.lri.fr/~mandel/mpc12`. This example illustrates the advantages of concrete resolution.

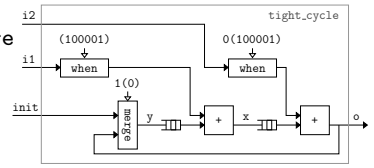
The encoder is depicted in Figure 6(a). The different nodes contain buffers, but they are connected without buffers. The global unification mentioned in

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, when clocks are abstracted, we do not have sufficiently precise information about them to guarantee 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 `II`, which would transform the unification constraints into subtyping constraints which could 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, for the same throughput, such 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.

Let us now consider the GSM speech decoder depicted in Figure 6(b). Notice that the flows `Ia`, `Ib` are `II` are buffered. The concrete resolution algorithm infers buffer sizes of, respectively, 1, 132 and 156, while the abstract resolution algorithm gives 51, 264 and 234. The buffer sizes estimated by the abstract resolution algorithm are almost twice as large as those found by the concrete algorithm.

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

Notice, however, that the abstract resolution algorithm is still interesting for cases like the video application *Picture in Picture* [17]. 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 [12] that solve systems of linear inequalities. Finally, the algorithm with abstraction can handle systems where some words are not exactly periodic [9], that is, those with some jitters.

For a given program, one algorithm may be more appropriate than the other. When the periodic words are well balanced, i.e., when the 1s are regularly spread (as is the case for the nodes of the *Picture in Picture* application), the algorithm with abstraction gives good results quickly. 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 1s 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. Finally, unlike the abstract algorithm, the concrete algorithm is not limited to optimizing system throughput. For example, it can find a schedule for *Picture in Picture* that reduces throughput in order to avoid buffering.

7 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.

Scheduling and finding buffer sizes for 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 [11] and StreamIt [22] which are implementations of the Synchronous Data-Flow model [14]. In Ptolemy, the computation nodes are programmed in a host language and the production and consumption rates of 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 can accept more programs than previous algorithms and it does not overestimate buffer sizes.

Even if the algorithm presented in this paper is computationally more complex than the abstract algorithm presented in [17], our new algorithm can type some programs that would be impossible to type with the other one. In particular, the concrete resolution algorithm has been used [18] 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 [7, 2, 4]. Because of their shape, all these programs make the abstract algorithm fail.

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.

Acknowledgments. First we would like to thank Timothy Bourke for his numerous and useful comments on the article. It has been a great pleasure to interact with him. We are grateful to Gwenaël Delaval for providing the GSM example which is ideal for motivating and illustrating our approach. Marc Pouzet has been very supportive and always has good advice. Last but not least, we would like to thank the reviewers and the MPC program committee for the quality and the benevolence of their remarks.

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, January 2007.
3. L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits ans Systems*, 20(9):1059–1076, September 2001.
4. J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky. Scheduling synchronous elastic designs. In *Application of Concurrency to System Design*, 2009.
5. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Principles of Programming Languages*, 1987.
6. P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, May 1996.
7. M. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *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 *Principles of Programming Languages*, 2006.
9. A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of clocks in synchronous data-flow systems. In *ASIAN Symposium on Programming Languages and Systems*, 2008.
10. J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *International Conference on Embedded Software*, 2003.
11. J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003.
12. GLPK. GNU linear programming kit. `http://www.gnu.org/software/glpk/`.
13. X. Lagrange, P. Godlewski, and S. Tabbane. *Réseaux GSM : des principes à la norme*. Hermès Science, Paris, 2000.
14. Ed. Lee and D. Messerschmitt. Synchronous data flow. *IEEE Transactions on Computers*, 75(9), September 1987.
15. L. Mandel and F. Plateau. Typage des horloges périodiques en Lucy-n. In *Journées Francophones des Langages Applicatifs*, La Bresse, France, January 2011.
16. L. Mandel and F. Plateau. Scheduling and buffer sizing of n-synchronous systems — extended version. Available at `http://www.lri.fr/~mandel/mpc12`, 2012.
17. L. Mandel, F. Plateau, and M. Pouzet. Lucy-n: a n-synchronous extension of Lustre. In *Mathematics of Program Construction*, 2010.

18. L. Mandel, F. Plateau, and M. Pouzet. Static scheduling of latency insensitive designs with Lucy-n. In *Formal Methods in Computer Aided Design*, 2011.
19. W. Wesley Peterson. *Error-Correcting Codes*. The M.I.T. Press, 1961.
20. A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.
21. T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming*, pages 341–352, 2009.
22. W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, 2002.

A Clock constraints collection

Clock types are separated into three categories: type schemes (σ) which represent the types of nodes, types of expressions (t) and types of streams (ct).

$$\begin{array}{l} \sigma \ ::= \forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, ct \to ct \\ t \ ::= \beta \mid t \times t \mid ct \\ ct ::= \alpha \mid ct \ \texttt{on}\ ce \mid ct \ \texttt{on not}\ ce \end{array}$$

The typing environment H is a triple which contains the types of the flow variables, the types of the nodes and the type of the clocks:

$$\begin{array}{rl} H ::= ([\, x_1 : ct_1, \ldots, x_p : ct_p \,], & \\ \quad [\, f_1 : \sigma_1, \ldots, f_m : \sigma_m \,], & \\ \quad [\, c_1 : ce_1, \ldots, c_n : ce_n \,]) & \end{array}$$

We use the notation $H + [z : t]$ to add the association $z : t$ to the appropriate part of the triple. We define the notation $[pat : t]$ as follows:

$$[pat : t] = \begin{cases} [x : t] & \text{if } pat = x \\ [pat_1 : t_1] + \cdots + [pat_n : t_n] & \text{if } pat = (pat_1, \ldots, pat_n) \text{ and } t = t_1 \times \cdots \times t_n \end{cases}$$

Types can be instantiated and generalized using the following rules:

$$\begin{array}{l} inst(\forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, t) = \\ \quad \{\, t' \mid t' = t[\beta_1 \leftarrow t_1, \ldots, \beta_m \leftarrow t_m, \alpha_1 \leftarrow ct_1, \ldots, \alpha_n \leftarrow ct_n] \,\} \\ gen(t, C) = \forall\beta_1,\ldots,\beta_m.\forall\alpha_1,\ldots,\alpha_n.\, t' \\ \quad \text{where } t' = \theta(t) \text{ such that } \theta(C) \text{ is satisfied} \\ \quad \text{and } \{\beta_1, \ldots, \beta_m, \alpha_1, \ldots, \alpha_n\} = FV(t') \end{array}$$

The typing rules which collect the clocking constraints are given in Figure 7. The typing rules have the shape $H \vdash e : t \,|\, C$ which means that in the typing environment H, the expression e has type t and must satisfy the set of constraints C.

Finally, notice that $ct \times ct \equiv ct$. Therefore, constraints such as $t_1 \times t_2 = ct$ can be split into $t_1 = ct$ and $t_2 = ct$.

$$H \vdash i : \alpha \,|\, \emptyset \qquad \frac{H \vdash ce}{H \vdash ce : \alpha \,|\, \emptyset} \qquad H \vdash x : H(x) \,|\, \emptyset$$

$$\frac{H \vdash e_1 : t_1 \,|\, C_1 \qquad \ldots \qquad H \vdash e_n : t_n \,|\, C_n}{H \vdash (e_1, \ldots, e_n) : t_1 \times \cdots \times t_n \,|\, C_1 \cup \cdots \cup C_n}$$

$$\frac{H \vdash e_1 : t_1 \,|\, C_1 \qquad H \vdash e_2 : t_2 \,|\, C_2}{H \vdash e_1 \; op \; e_2 : ct \,|\, \{t_1 \;=\; t_2 \;=\; ct\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \,|\, C \qquad H \vdash e_1 : t_1 \,|\, C_1 \qquad H \vdash e_2 : t_2 \,|\, C_2}{H \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : ct \,|\, \{t \;=\; t_1 \;=\; t_2 \;=\; ct\} \cup C \cup C_1 \cup C_2}$$

$$\frac{t_1 \to t_2 \in inst(H(f)) \qquad H \vdash e : t_3 \,|\, C}{H \vdash f\,e : t_2 \,|\, \{t_1 \;=\; t_3\} \cup C}$$

$$\frac{H \vdash eqs : H' \,|\, C_1 \qquad H + H' \vdash e : ct \,|\, C_2}{H \vdash e\ \texttt{where rec}\ eqs : ct \,|\, C_1 \cup C_2}$$

$$\frac{H \vdash e_1 : t_1 \,|\, C_1 \qquad H \vdash e_2 : t_2 \,|\, C_2}{H \vdash e_1\ \texttt{fby}\ e_2 : ct \,|\, \{t_1 \;=\; t_2 \;=\; ct\} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \,|\, C \qquad H \vdash ce : ct \,|\, \emptyset}{H \vdash e\ \texttt{when}\ ce : ct\ \texttt{on}\ ce \,|\, \{t \;=\; ct\} \cup C} \qquad \frac{H \vdash e : t \,|\, C \qquad H \vdash ce : ct \,|\, \emptyset}{H \vdash e\ \texttt{whenot}\ ce : ct\ \texttt{on not}\ ce \,|\, \{t \;=\; ct\} \cup C}$$

$$\frac{H \vdash ce : ct \,|\, \emptyset \qquad H \vdash e_1 : t_1 \,|\, C_1 \qquad H \vdash e_2 : t_2 \,|\, C_2}{H \vdash \texttt{merge}\ ce\ e_1\ e_2 : ct \,|\, \{ct\ \texttt{on}\ ce \;=\; t_1, ct\ \texttt{on not}\ ce \;=\; t_2, \} \cup C_1 \cup C_2}$$

$$\frac{H \vdash e : t \,|\, C}{H \vdash \texttt{buffer}(e) : \alpha \,|\, C \cup \{t \;<\!:\; \alpha\}}$$

$$\frac{H + [pat : \beta] \vdash e : t \,|\, C}{H \vdash pat = e : [pat : t] \,|\, \{\beta \;=\; t\} \cup C} \qquad \frac{H + H_2 \vdash eqs_1 : H_1 \,|\, C_1 \qquad H + H_1 \vdash eqs_2 : H_2 \,|\, C_2}{H \vdash eqs_1\ \texttt{and}\ eqs_2 : H_1 + H_2 \,|\, C_1 \cup C_2}$$

$$\frac{H + [x : \beta] \vdash e : t \,|\, C}{H \vdash \texttt{let node}\ f(x) = e : [f : gen(\beta \to t, C)]} \qquad \frac{H \vdash ce}{H \vdash \texttt{let clock}\ c = ce : [c : ce]}$$

$$\frac{H \vdash d_1 : H_1 \qquad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2}$$

Fig. 7. Clock type constraints collection.