

---

# LUCY-N : une extension n-synchrone de LUSTRE

---

Louis Mandel, Florence Plateau & Marc Pouzet

*LRI, Université Paris-Sud 11*

*INRIA Saclay*

{mandel,plateau,pouzet}@lri.fr

## Résumé

Les langages synchrones flot de données permettent de programmer des réseaux de processus communicant sans buffers. Pour cela, chaque flot est associé à un type d'horloges, qui indique les instants de présence de valeurs sur le flot. La communication entre deux processus  $f$  et  $g$  peut être faite sans buffer si le type du flot de sortie de  $f$  est égal au type du flot d'entrée de  $g$ . Un système de type, le calcul d'horloge, infère des types tels que cette condition est vérifiée. Le modèle n-synchrone a pour but de relâcher ce modèle de programmation en autorisant les communications à travers des buffers de taille bornée. En pratique, cela consiste à introduire une règle de sous-typage dans le calcul d'horloge.

Nous avons présenté l'année dernière un article décrivant comment abstraire des horloges pour vérifier la relation de sous-typage. Cette année, nous présentons un langage de programmation n-synchrone : LUCY-N. Dans ce langage, l'inférence des types d'horloges est paramétrable par l'algorithme de résolution des contraintes de sous-typage. Nous montrons ici un algorithme basé sur les travaux de l'an dernier et comment programmer en LUCY-N à travers l'exemple d'une application de traitement multimédia.

## 1. Introduction

Nous nous intéressons aux modèles et aux langages pour la programmation d'applications de traitement de flux ayant des contraintes de temps-réel comme les applications multimédias. Dans le cas d'une application vidéo par exemple, il faut traiter des flots infinis de pixels sur lesquels sont appliquées des opérations successives, ou *filters*. Il y a des contraintes de temps-réel car la fréquence d'affichage des pixels ne peut être diminuée. Il y a également des contraintes de performance car des milliards d'opérations doivent être effectuées par seconde. Enfin, il y a des contraintes de sûreté : on veut garantir que le comportement de l'application est déterministe, sans blocage et que la mémoire nécessaire à l'exécution est bornée.

Afin de respecter ces contraintes, les ingénieurs du domaine sont habitués à modéliser leurs systèmes par des réseaux de processus de Kahn [8]. Dans ce modèle, des nœuds de calcul s'exécutent de manière concurrente et communiquent par des canaux munis de *mémoires tampon* (ou *buffers*) infinies de type *First In First Out*. La lecture est bloquante si le buffer est vide, et comme les buffers sont de taille infinie, l'écriture est non bloquante. On ne peut pas tester l'absence de valeur dans un buffer.

L'intérêt ce modèle de programmation est d'être concurrent tout en préservant le déterminisme. De plus, il peut être vu comme un modèle mathématique où chaque nœud correspond à une fonction sur les suites et le réseau correspond à la composition de ces fonctions, qui est aussi une fonction de suites. On sait ainsi raisonner de manière formelle sur les systèmes décrits dans ce formalisme. Néanmoins, l'inconvénient de ce modèle est qu'il ne donne pas de garanties sur l'exécution en temps-réel, l'absence de blocage et le caractère borné de la mémoire nécessaire à la communication.

Il est possible d'implémenter des réseaux de Kahn sans prendre le risque d'écritures dans des buffers pleins en mettant en place un mécanisme de rétroaction (ou *back pressure*). Ce mécanisme

rend l'écriture bloquante. Ainsi, il permet d'éviter les débordements de capacité des buffers, mais il crée des blocages artificiels dans le cas où les tailles des buffers ont été sous-estimées. Ce problème peut être traité en augmentant les tailles durant l'exécution en cas de blocage dû à un manque de place [12]. Cependant, on aimerait avoir des garanties statiques sur l'exécution. Pour cela, il faut savoir calculer des tailles suffisantes pour les buffers si elles existent, et rejeter les réseaux qui nécessitent des buffers de taille infinie.

Le problème de savoir si un réseau de Kahn quelconque peut être exécuté en mémoire bornée est indécidable [2]. Pour avoir des garanties sur le caractère borné des buffers nécessaires, il faut donc se placer dans un sous-ensemble des réseaux de Kahn sur lesquels des informations supplémentaires sont disponibles. C'est l'approche choisie par le modèle des *Synchronous Dataflow* [9, 11] qui se restreignent à des réseaux de Kahn dans lesquels le nombre de valeurs produites et consommées par chaque nœud à chaque activation est connu statiquement. Cela permet de calculer un ordonnancement statique périodique et une taille suffisante pour les buffers pour cet ordonnancement.

L'approche consistant à se placer dans un sous-ensemble des réseaux de Kahn est aussi l'approche suivie par le modèle de programmation synchrone [1]. Dans ce modèle, on ne considère que les réseaux qui peuvent être exécutés sans buffers. Pour cela, chaque canal est associé à une *horloge* qui définit les instants où une valeur est présente.

L'intérêt des langages synchrones est de permettre de spécifier formellement une application temps-réel tout en étant compilables vers du code exécutable. Ils expriment la concurrence des différents traitements, et sont donc bien adaptés à une compilation permettant d'exécuter parallèlement des filtres. Ils offrent des garanties fortes, comme le déterminisme, l'absence de blocage et un besoin en mémoire borné. La contrepartie des garanties fournies malgré une grande expressivité des rythmes est un manque de souplesse dans la composition des flots. La composition sans buffer n'est acceptée que si le calcul d'horloges sait vérifier l'égalité des horloges des flots que l'on compose.

La vérification des horloges en LUSTRE [3] et LUCID SYNCHRONE [14] est un problème de typage et peut être décrite par un système de types [4, 6]. Chaque expression est associée à un type d'horloges et le compilateur vérifie que le programme respecte des contraintes sur ces types. Considérons par exemple la règle suivante :

$$\frac{H \vdash e_1 : ck \quad H \vdash e_2 : ck}{H \vdash e_1 + e_2 : ck}$$

Elle indique qu'une addition de deux flots  $e_1$  et  $e_2$  est correcte à condition que  $e_1$  et  $e_2$  aient le même type d'horloges  $ck$ . Dans ce cas, le type d'horloges du résultat  $e_1 + e_2$  est aussi  $ck$ . Ainsi, pour typer un programme, les questions qui se posent concernent toutes l'égalité de types. L'idée du modèle n-synchrone [5] est d'assouplir cette contrainte d'égalité afin de permettre la composition de flots non nécessairement synchrones dès lors que leurs horloges sont proches l'une de l'autre. Si par insertion d'un buffer borné, un flot  $x$  dont l'horloge est de type  $ck$  peut être consommé un peu plus tard sur une horloge de type  $ck'$ , on dira que  $ck$  est un sous-type de  $ck'$ . Cette relation sera notée  $ck <: ck'$ . On étend alors le langage avec une construction **buffer** qui indique les points où il faut appliquer la règle de sous-typage :

$$\frac{H \vdash e : ck \quad ck <: ck'}{H \vdash \mathbf{buffer} e : ck'}$$

Dans cet article, nous présentons le langage LUCY-N, une extension de LUSTRE fondée sur le modèle n-synchrone. La section 2 décrit le noyau du langage et comment programmer avec celui-ci à travers des exemples simples. La section 3 donne les bases d'algèbre sur les mots binaires qui sont utilisées dans le système de type. La section 4 présente le système de type du langage et la section 6 comment résoudre les contraintes de sous-typage. La section 6 utilise les résultats de [10] qui sont rappelés section 5. Enfin, l'implantation du système de type est discutée section 7 et utilisée section 8 sur un exemple d'application vidéo. Le prototype associé à LUCY-N est disponible à l'adresse <http://www.lri.fr/~plateau/jfla10>.

$e$	$::=$	$i$	flot constant
		$  x$	variable de flot
		$  (e, e)$	paire
		$  op (e, e)$	opérateur importé
		$  f e$	application
		$  e \textbf{ where rec } eqs$	définitions locales
		$  e \textbf{ fby } e$	délai initialisé
		$  e \textbf{ when } ce \mid e \textbf{ whenot } ce$	échantillonnage
		$  \textbf{ merge } ce e e$	fusion
		$  \textbf{ buffer } e$	bufferisation
$eqs$	$::=$	$pat = e$	équation
		$  eqs \textbf{ and } eqs$	ensemble d'équations
$pat$	$::=$	$x \mid (pat, pat)$	motifs
$d$	$::=$	$\textbf{ let node } f \textbf{ } pat = e$	définition d'un nœud
		$  \textbf{ let clock } c = ce$	définition d'une horloge
		$  d$	séquence de définitions

FIG. 1 – Noyau du langage.

## 2. Le langage

Le langage LUCY-N est un langage synchrone flot de données, semblable à LUSTRE, étendu avec un opérateur de bufferisation. Le noyau du langage est défini dans la figure 1.

Un programme ( $d$ ) est une séquence de définitions de fonctions sur les flots, appelées nœuds, et d'horloges. Le comportement d'un nœud est défini par une expression ( $e$ ). Les expressions s'évaluent vers les flots de valeurs et peuvent être définies à l'aide d'ensembles d'équations mutuellement récursives ( $eqs$ ).

Dans ce noyau synchrone, les expressions d'horloges ( $ce$ ) peuvent être définies avec tout langage d'horloges s'évaluant vers des séquences booléennes infinies et muni des opérations suivantes :

- un test d'égalité des horloges, pour vérifier que les communications réalisées sans buffer sont synchrones ;
- un test d'*adaptabilité* des horloges, pour vérifier que les communications réalisées à travers un buffer sont n-synchrones ;
- une opération permettant de calculer une taille suffisante pour chaque buffer.

En LUSTRE, les expressions d'horloges peuvent être définies par n'importe quelle expression booléenne du langage. Ces expressions pouvant être arbitrairement compliquées, le test d'égalité des horloges se limite à de l'égalité syntaxique. Dans ce cadre, on peut difficilement imaginer un test d'adaptabilité et un calcul des tailles de buffers. On doit donc utiliser un langage d'horloges plus simple pour pouvoir utiliser des communications par buffers.

Dans cet article, nous nous limiterons à des expressions d'horloges définies par des mots binaires ultimement périodiques. Elles sont décrites par la grammaire suivante :

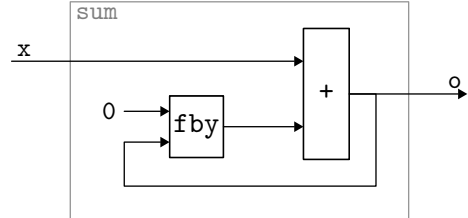
$$\begin{aligned}
 ce &::= c \mid u(v) \\
 u &::= \varepsilon \mid 0.u \mid 1.u \\
 v &::= 0 \mid 1 \mid 0.v \mid 1.v
 \end{aligned}$$

Une expression d'horloges est ici soit un nom d'horloge  $c$ , soit un mot binaire  $p$  composé d'un préfixe  $u$  (éventuellement vide  $\varepsilon$ ) suivi de la répétition infinie d'un motif  $v$ . Par exemple, l'expression (10) désigne le flot booléen  $101010\dots$ , où 1 désigne la valeur *vrai* et 0 désigne la valeur *faux*.

**Premier programme.** Les programmes LUCY-N s'écrivent de façon similaire aux programmes LUSTRE. Illustrons cela au travers de l'exemple d'un intégrateur, qui calcule la somme des valeurs prises par son flot d'entrée :

```
let node sum x = o where
  rec o = x + (0 fby o)
```

La sortie de ce nœud est la valeur définie par  $o$ . Le mot clé **where** introduit l'équation récursive définissant cette variable. L'opérateur **fby** permet d'utiliser des valeurs précédentes d'un flot. L'expression  $0 \text{ fby } o$  vaut 0 au premier instant, puis la valeur précédente de  $o$  aux instants suivants. La sortie  $o$  est donc égale à la somme de sa valeur précédente et de l'entrée courante.

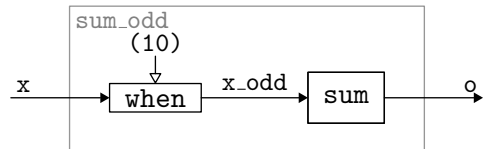


Si l'on applique le nœud **sum** sur l'entrée  $5 \ 7 \ 3 \ 6 \ 2 \ 8 \ \dots$ , on obtient les suites de valeurs suivantes pour les expressions du nœud :

$x$	5	7	3	6	2	8	...
$0 \text{ fby } o$	0	5	12	15	21	23	...
$o$	5	12	15	21	23	31	...

**Échantillonnage et fusion de flots.** L'opérateur **when** permet de supprimer certaines valeurs d'un flot. Dans l'expression  $x \text{ when } c$ ,  $x$  est le flot qu'on échantillonne et  $c$  est la condition d'échantillonnage : les valeurs de  $x$  sont conservées lorsque  $c$  vaut 1, et supprimées sinon. Par exemple, si l'on ne veut conserver que les valeurs de rang impair d'un flot  $x$ , on peut l'échantillonner avec l'horloge  $(10) = 101010 \dots$ . Ainsi, on peut réutiliser le nœud **sum** précédent pour écrire un nœud qui somme les éléments de rang impair d'un flot :

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



Si l'on applique ce nœud au flot d'entrée de l'exemple précédent, on obtient les suites de valeurs suivantes :

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

On observe que les flots  $x\_odd$  et  $o$  ne sont définis qu'un instant sur deux. Pour distinguer les instants où un flot est défini des instants où il n'est pas défini, on lui associe une horloge.<sup>1</sup> Celle-ci prend la valeur 1 aux instants où le flot est présent, et la valeur 0 aux instants où le flot est absent. Ainsi, le flot  $x$  et la condition d'échantillonnage qui sont définis à chaque instant sont associés à l'horloge  $(1) = 1111 \dots$ . Le flot  $x\_odd$  n'est quant à lui défini qu'un instant sur deux à partir du premier instant et son horloge est donc  $(10)$ . Cette horloge est le résultat de l'opération  $(1) \text{ on } (10)$ . L'opérateur **on** calcule l'horloge de  $x \text{ when } (10)$  en fonction de l'horloge de  $x$  et de l'horloge  $(10)$ .

**Définition 1** (opérateur **on**).  $0.w_1 \text{ on } w_2 \stackrel{\text{def}}{=} 0.(w_1 \text{ on } w_2)$   
 $1.w_1 \text{ on } 1.w_2 \stackrel{\text{def}}{=} 1.(w_1 \text{ on } w_2)$   
 $1.w_1 \text{ on } 0.w_2 \stackrel{\text{def}}{=} 0.(w_1 \text{ on } w_2)$

<sup>1</sup>L'appellation *horloge* est utilisée pour deux notions différentes : les conditions d'échantillonnage et les suites représentant les instants de présence des flots.

On peut échantillonner à nouveau un flot qui a déjà été échantillonné. Par exemple :

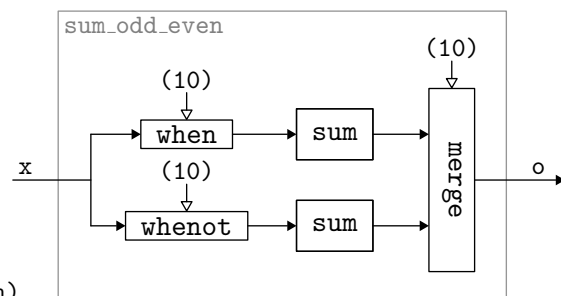
flot	valeurs	horloge
x	5 7 3 6 2 8 ...	(1)
(10)	1 0 1 0 1 0 ...	(1)
x when (10)	5 3 2 ...	(1) on (10) = (10)
(110)	1 1 0 ...	(10)
(x when (10)) when (110)	5 3 ...	((1) on (10)) on (110) = (101000)

L'horloge du flot échantillonné par (10) puis par (110) est ((1) on (10)) on (110). En appliquant la définition du *on*, on peut observer que cette horloge vaut (101000) : le flot est présent au premier instant, puis absent, puis présent, puis absent durant trois instants. Les conditions d'échantillonnage étant périodiques, ce comportement se reproduit infiniment.

L'opérateur *merge* permet de fusionner deux flots échantillonnés. Dans l'expression *merge c x1 x2*, les flots que l'on veut fusionner sont *x1* et *x2*, et *c* est la condition de fusion : si elle vaut 1, c'est la valeur de *x1* qui doit être transmise en sortie, et *x2* doit être absent, si elle vaut 0 c'est *x2* qui est transmise, et *x1* qui doit être absent.

Le nœud *sum\_odd\_even* sépare son flot d'entrée en deux flots plus lents complémentaires, leur applique l'opérateur d'intégration, puis les fusionne pour rendre une sortie présente aussi souvent que l'entrée.

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

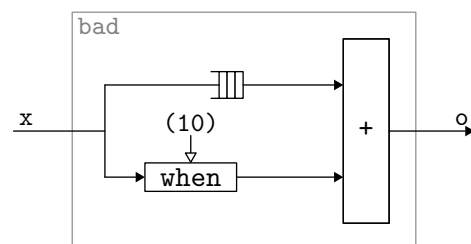


flot	valeurs	horloge
x	5 7 3 6 2 8 ...	(1)
(10)	1 0 1 0 1 0 ...	(1)
x_odd	5 3 2 ...	(1) on (10) = (10)
x_even	7 6 8 ...	(1) on not (10) = (01)
sum x_odd	5 8 10 ...	(10)
sum x_even	7 13 21 ...	(01)
o	5 7 8 13 10 21 ...	(1)

**Communication synchrone et n-synchrone.** De même qu'en LUSTRE, la composition de flots doit être synchrone, c'est-à-dire que les valeurs sur les flots doivent être présentes exactement aux instants où elles sont nécessaires aux calculs. Cette condition de synchronisme permet de garantir une exécution sans buffers, et en particulier de rejeter le programme suivant :

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

flot	valeurs	horloge
x	5 7 3 6 2 8 ...	(1)
x when (10)	5 3 2 ...	(1) on (10) = (10)



L'opérateur + doit être appliqué point à point : par conséquent, comme deux fois plus de valeurs arrivent sur le flot x que sur le flot x when (10), l'opération ne peut pas être réalisée de manière

synchrone. Le stockage des valeurs du flot  $x$  dans un buffer dans l'attente qu'elles soient traitées nécessiterait une mémoire infinie. L'absence de tels problèmes est garantie par les restrictions imposées par les langages synchrones [3]. Cependant, dans certains cas, on aimerait pouvoir composer des flots qui ne sont pas strictement synchrones. Par exemple, le programme suivant peut être exécuté en mémoire bornée, mais il est rejeté dans le cadre strictement synchrone car  $x1$  et  $x2$  n'ont pas la même horloge :

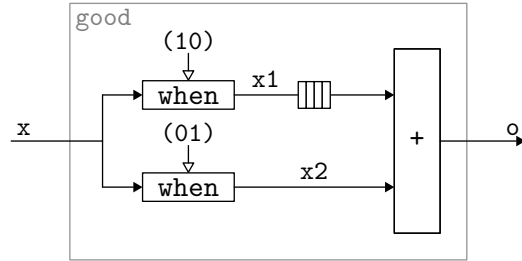
```
let node not_so_bad x = o where
  rec x1 = x when (10)
  and x2 = x when (01)
  and o = x1 + x2
```

flot	valeurs	horloge
$x1$	5    3    2    ...	(10)
$x2$	7    6    8    ...	(01)

Pour cela, on ajoute un opérateur **buffer** qui permet de spécifier les endroits où l'on autorise une composition nécessitant de stocker des flots dans des buffers, car leurs valeurs arrivent avant les instants où elles sont nécessaires. La communication est alors qualifiée de  $n$ -synchrone,  $n$  représentant la taille du buffer à mettre en place, par opposition à 0-synchrone qui ne nécessite pas de buffer.

Le nœud `not_so_bad` peut ainsi être accepté s'il est réécrit de la manière suivante :

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



flot	valeurs	horloge
$x1$	5    3    2    ...	(10)
$\text{buffer}(x1)$	5    3    2    ...	(01)
$x2$	7    6    8    ...	(01)
$\text{buffer}(x1) + x2$	12   9   10   ...	(01)

Un programme utilisant l'opérateur **buffer** est accepté seulement si l'horloge du flot stocké dans le buffer est *adaptable* à l'horloge où l'on veut consommer ce flot. Nous définissons dans la section suivante cette relation d'adaptabilité.

### 3. Relation d'adaptabilité

Donnons tout de suite l'intuition de la relation d'adaptabilité : *l'horloge  $w_1$  est adaptable à l'horloge  $w_2$  si et seulement si tout flot d'horloge  $w_1$  peut être consommé sur l'horloge  $w_2$  par insertion d'un buffer de taille bornée*. Cette relation garantit non seulement que la mémoire à mettre en place pour bufferiser le flot en attendant son utilisation est de taille bornée, mais aussi que l'on ne risque aucune lecture dans une mémoire vide.

Pour définir formellement cette relation, nous devons dans un premier temps introduire la *fonction de cumul* d'un mot binaire :  $\mathcal{O}_w(i)$ . Cette fonction permet de décrire un mot binaire en associant à un indice  $i$  le nombre de 1 dans un mot  $w$  entre le début du mot et l'indice  $i$ . On peut voir figure 2 les fonctions de cumul des mots  $w_1 = (11010)$  et  $w_2 = 0(00111)$ .

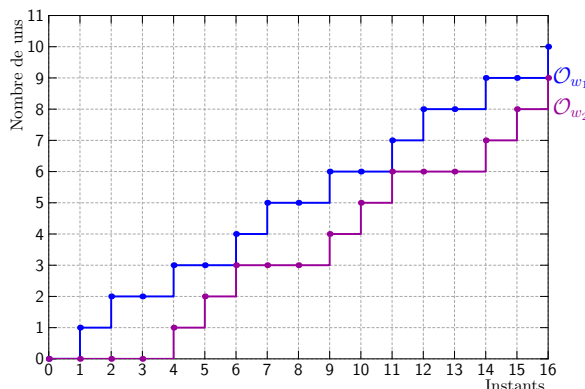


FIG. 2 – Fonctions de cumul des mots  $w_1 = (11010)$  et  $w_2 = 0(00111)$ .

**Définition 2** (éléments et fonction de cumul de  $w$ ). Soit  $w = b.w'$  avec  $b \in \{0, 1\}$ .

On note  $w[i]$  l'accès au  $i$ ème élément d'un mot :

$$\begin{aligned} w[1] &\stackrel{\text{def}}{=} b \\ \forall i > 1, w[i] &\stackrel{\text{def}}{=} w'[i-1] \end{aligned}$$

On note  $\mathcal{O}_w$  la fonction de cumul d'un mot  $w$  :

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

Nous définissons maintenant la relation d'*adaptabilité* entre deux mots  $w_1$  et  $w_2$ , notée  $w_1 <: w_2$ . Cette relation assure qu'un flot produit sur le rythme  $w_1$  peut être consommé sur le rythme  $w_2$  par insertion d'un buffer borné. Pour cela, il faut d'une part qu'il n'y ait pas de lecture dans un buffer vide, c'est-à-dire qu'à chaque instant il y ait toujours eu plus d'écritures que de lectures dans le buffer. Cette propriété est exprimée par la relation de *précédence* entre les mots  $w_1$  et  $w_2$ , notée  $w_1 \preceq w_2$ . D'autre part, il faut que le nombre de valeurs présentes dans le buffer au cours de l'exécution soit borné. Cette propriété est exprimée par la relation de *synchronisabilité* entre les mots  $w_1$  et  $w_2$ , notée  $w_1 \bowtie w_2$ . La relation d'*adaptabilité* est donc la conjonction des relations de précédence et de synchronisabilité.

**Définition 3** (synchronisabilité  $\bowtie$ , précédence  $\preceq$  et adaptabilité  $<:$ ).

$$\begin{aligned} w_1 \bowtie w_2 &\stackrel{\text{def}}{\iff} \exists b_1, b_2 \in \mathbb{Z}, \forall i \geq 0, b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2 \\ w_1 \preceq w_2 &\stackrel{\text{def}}{\iff} \forall i > 0, \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i) \\ w_1 <: w_2 &\stackrel{\text{def}}{\iff} w_1 \preceq w_2 \wedge w_1 \bowtie w_2 \end{aligned}$$

Dans la figure 2,  $w_1 \bowtie w_2$  car la distance verticale entre les deux courbes est bornée et  $w_1 \preceq w_2$  car la courbe  $\mathcal{O}_{w_1}$  est toujours située au dessus de  $\mathcal{O}_{w_2}$ .

**Taille du buffer.** Considérons un buffer qui prend en entrée un flot sur le rythme  $w_1$ , et fournit en sortie un flot sur le rythme  $w_2$ . Le nombre d'éléments présents à chaque instant  $i$  dans le buffer est la différence entre le nombre de valeurs qui ont été écrites dans le buffer ( $\mathcal{O}_{w_1}(i)$ ) et le nombre de valeurs qui y ont été lues ( $\mathcal{O}_{w_2}(i)$ ) :

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

Une valeur négative pour cette différence signifie que plus de valeurs ont été lues qu'écrites, c'est-à-dire que des lectures ont été effectuées dans le buffer alors qu'il était vide.

La taille de buffer nécessaire et suffisante est le nombre maximal de valeurs présentes dans le buffer durant l'exécution :

$$size(w_1, w_2) = \max_{i \in \mathbb{N}^*} (\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$$

Ainsi, nous pouvons exprimer la propriété souhaitée pour la relation d'adaptabilité : si  $w_1$  est adaptable à  $w_2$ , la consommation d'un flot d'horloge  $w_1$  peut être faite sur l'horloge  $w_2$  par insertion d'un buffer de taille bornée et sans lectures dans un buffer vide.

**Théorème 1** (communication par buffers).

$$w_1 <: w_2 \Rightarrow (size(w_1, w_2) < +\infty) \wedge (\forall i, size_i(w_1, w_2) \geq 0)$$

**Preuve :** La relation d'adaptabilité est la conjonction de la relation de précédence et de la relation de synchronisabilité. Par définition de la relation de synchronisabilité et par la formule donnant la taille du buffer, on a la garantie que  $size(w_1, w_2) < +\infty$ . Par définition de la relation de précédence et par la formule donnant le nombre d'éléments dans le buffer à chaque instant, on a la garantie que  $\forall i, size_i(w_1, w_2) \geq 0$ .  $\square$

Comme les mots ultimement périodiques ont un comportement répétitif à partir d'un certain rang, la vérification de la relation d'adaptabilité et le calcul de la taille des buffers peuvent être effectués statiquement.

## 4. Calcul d'horloge

Tout comme les types de données sont utilisés pour abstraire les données, on introduit des types d'horloges pour abstraire les horloges des flots d'un programme. Ainsi, on peut définir un système de type des horloges qui garantit la cohérence des horloges d'un programme, c'est-à-dire que le programme peut être exécuté avec des buffers de taille finie. Ce système de type s'appelle le calcul d'horloge.

Le calcul d'horloge traditionnel n'autorise que des communications strictement synchrones (sans buffer). Il ne contient donc qu'une notion d'égalité d'horloges. Afin de pouvoir typer l'opérateur `buffer`, le calcul d'horloge est enrichi d'une règle de sous-typage appliquée aux points d'utilisation de cet opérateur. Cette règle de sous-typage garantit que l'horloge du flot en entrée du buffer est adaptable à l'horloge du flot en sortie.

Les types d'horloges (ou simplement types) sont séparés en trois catégories : les schémas de types ( $\sigma$ ) qui décrivent les horloges des sorties des nœuds en fonction des horloges de leurs entrées, les types des expressions ( $ct$ ) et les types des flots ( $ck$ ). Les types des flots peuvent décrire les horloges de flots échantillonnés ( $ck$  on  $ce$ ,  $ck$  on not  $ce$ ) et les horloges de flots quelconques ( $\alpha$ ).

$$\begin{aligned} \sigma & ::= \forall \beta_1, \dots, \beta_m. \forall \alpha_1, \dots, \alpha_n. ct \rightarrow ct \\ ct & ::= \beta \mid ct \times ct \mid ck \\ ck & ::= \alpha \mid ck \text{ on } ce \mid ck \text{ on not } ce \end{aligned}$$

Les règles du calcul d'horloge sont de la forme  $H \vdash e : ct \mid C$ . Elles sont définies figure 3.2 Le jugement  $H \vdash e : ct \mid C$  signifie que « l'expression  $e$  a le type d'horloges  $ct$  dans l'environnement de typage  $H$ , sous l'ensemble de contraintes de sous-typage  $C$  ». Par exemple, la règle (OP) vérifie que les deux arguments d'un opérateur importé sont synchrones, c'est-à-dire de même type d'horloges. Dans la règle (IM), on peut observer qu'une constante peut être utilisée sur n'importe quel rythme  $ck$ .

L'originalité de ce calcul d'horloge réside dans la règle (BUF) de typage des buffers. Le type d'une expression bufferisée de type  $ck_1$  peut être n'importe quel type  $ck_2$  tel que  $ck_1 <: ck_2$ . Pour cela, on donne le type  $ck_2$  à la sortie du buffer, et on ajoute la contrainte  $ck_1 <: ck_2$  à l'ensemble de

<sup>2</sup> $H + [pat : ct]$  ajoute chaque variable du motif  $pat$  avec son type dans la partie adéquate de l'environnement.

$$\begin{array}{c}
\text{(IM)} \quad H \vdash i : ck \mid \emptyset \qquad \text{(CE)} \quad H \vdash ce : ck \mid \emptyset \qquad \text{(VAR)} \quad H \vdash x : H(x) \mid \emptyset \\
\text{(PAIR)} \quad \frac{H \vdash e_1 : ct_1 \mid C_1 \quad H \vdash e_2 : ct_2 \mid C_2}{H \vdash (e_1, e_2) : ct_1 \times ct_2 \mid C_1, C_2} \qquad \text{(OP)} \quad \frac{H \vdash e_1 : ck \mid C_1 \quad H \vdash e_2 : ck \mid C_2}{H \vdash op(e_1, e_2) : ck \mid C_1, C_2} \\
\text{(APP)} \quad \frac{ct_1 \rightarrow ct_2 \in inst(H(f)) \quad H \vdash e : ct_1 \mid C}{H \vdash fe : ct_2 \mid C} \qquad \text{(FBY)} \quad \frac{H \vdash e_1 : ck \mid C_1 \quad H \vdash e_2 : ck \mid C_2}{H \vdash e_1 \text{ fby } e_2 : ck \mid C_1, C_2} \\
\text{(WHEN)} \quad \frac{H \vdash e : ck \mid C \quad H \vdash ce : ck \mid \emptyset}{H \vdash e \text{ when } ce : ck \text{ on } ce \mid C} \qquad \text{(WHENOT)} \quad \frac{H \vdash e : ck \mid C \quad H \vdash ce : ck \mid \emptyset}{H \vdash e \text{ whenot } ce : ck \text{ on not } ce \mid C} \\
\text{(MERGE)} \quad \frac{H \vdash ce : ck \mid \emptyset \quad H \vdash e_1 : ck \text{ on } ce \mid C_1 \quad H \vdash e_2 : ck \text{ on not } ce \mid C_2}{H \vdash \text{merge } ce \ e_1 \ e_2 : ck \mid C_1, C_2} \\
\text{(WHERE)} \quad \frac{H \vdash eqs : H' \mid C_1 \quad H + H' \vdash e : ct \mid C_2}{H \vdash e \text{ where rec } eqs : ct \mid C_1, C_2} \qquad \text{(BUF)} \quad \frac{H \vdash e : ck_1 \mid C}{H \vdash \text{buffer } e : ck_2 \mid C, \{ck_1 <: ck_2\}} \\
\text{(EQ)} \quad \frac{H + [pat : ct] \vdash e : ct \mid C}{H \vdash pat = e : [pat : ct] \mid C} \qquad \text{(EQS)} \quad \frac{H + H_2 \vdash eqs_1 : H_1 \mid C_1 \quad H + H_1 \vdash eqs_2 : H_2 \mid C_2}{H \vdash eqs_1 \text{ and } eqs_2 : H_1 + H_2 \mid C_1, C_2} \\
\text{(NODE)} \quad \frac{H + [x : ct_1] \vdash e : ct_2 \mid C}{H \vdash \text{let node } f \ x = e : [f : gen(ct_1 \rightarrow ct_2, C)]} \qquad \text{(CLOCK)} \quad \frac{}{H \vdash \text{let clock } c = ce : [c : ce]} \\
\text{(DEF)} \quad \frac{H \vdash d_1 : H_1 \quad H + H_1 \vdash d_2 : H_2}{H \vdash d_1; d_2 : H_1 + H_2} \\
inst(\forall \beta_1, \dots, \beta_m. \forall \alpha_1, \dots, \alpha_n. ct) = \{ ct' \mid ct' = ct[\beta_1 \leftarrow ct_1, \dots, \beta_m \leftarrow ct_m, \alpha_1 \leftarrow ck_1, \dots, \alpha_n \leftarrow ck_n] \} \\
H ::= ([x_1 : ct_1, \dots, x_p : ct_p], [f_1 : \sigma_1, \dots, f_m : \sigma_m], [c_1 : ce_1, \dots, c_n : ce_n]) \\
C ::= \{ck_1 <: ck'_1; \dots; ck_t <: ck'_m\}
\end{array}$$

FIG. 3 – Le calcul d’horloge.

contraintes  $C$ . Les autres règles sont similaires à celles de [6]. La seule différence est qu’il faut collecter les contraintes de sous-typage.

Lors du typage des nœuds, les types sont généralisés de la façon suivante :

$$gen(ct, C) = \forall \beta_1, \dots, \beta_m. \forall \alpha_1, \dots, \alpha_n. ct' \quad \text{où } ct' = \theta(ct) \text{ tel que } \theta(C) \text{ est satisfait} \\
\text{et } \{\beta_1, \dots, \beta_m, \alpha_1, \dots, \alpha_n\} = FV(ct')$$

Un type  $ct$  contraint par  $C$  peut être généralisé en instanciant les variables contraintes dans  $C$  par des types de flots satisfaisant ces contraintes (c’est le rôle de la substitution  $\theta$ ); puis en quantifiant universellement les variables du type ainsi obtenu ( $FV(ct')$  désigne l’ensemble des variables libres dans  $ct'$ ).

Remarquons que nous avons choisi de résoudre les contraintes au moment de la généralisation, plutôt que de les transporter dans le type. Cela nous permet de décider de la taille des buffers de façon modulaire et ainsi de pouvoir générer le code de chaque nœud séparément.

**Inférence des types d’horloges.** Tout comme dans le compilateur LUCID SYNCHRONE [6], nous avons choisi de réaliser une inférence des types d’horloges. Inférer les types signifie calculer les types associés aux expressions et aux nœuds, sans demander au programmeur d’annoter son programme.

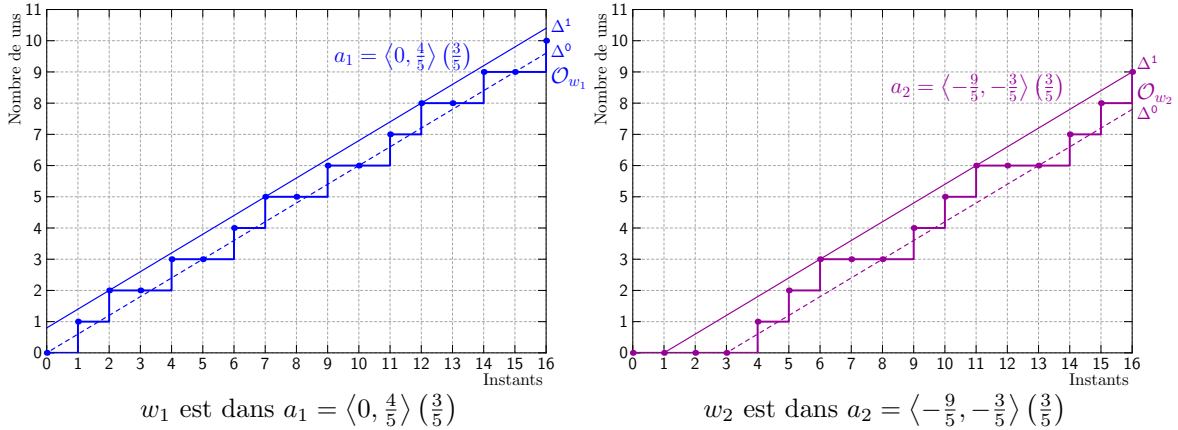


FIG. 4 – Représentation des abstractions.

L'inférence de types nécessite de savoir satisfaire deux sortes de contraintes sur les types d'horloges : d'une part, des contraintes d'unification, d'autre part, des contraintes de sous-typage.

Pour résoudre les contraintes d'unification, il faut trouver des instanciations des variables de types qui rendent les horloges égales. On peut comme en LUSTRE ou LUCID SYNCHRONE se limiter à de l'unification structurelle et tester l'égalité de deux horloges uniquement en comparant leur nom. Mais comme nous avons choisi un langage d'horloges plus simple, nous pouvons interpréter les horloges pour les comparer et nous pouvons utiliser des algorithmes d'unification plus puissants que l'unification structurelle. Par exemple, l'algorithme d'unification proposé dans [5] peut être utilisé.

Pour résoudre les contraintes de sous-typage, il faut trouver des instanciations des variables de types qui ne rendent plus les horloges égales mais adaptables. Comme pour l'unification, nous pouvons travailler sur les horloges périodiques pour trouver une solution au système de contraintes de sous-typage. Techniquement, cela revient à convertir un système de contraintes d'adaptabilité en système d'inéquations linéaires. Le problème est que cette méthode ne passe pas à l'échelle. En effet, le nombre d'inéquations linéaires à résoudre est proportionnel aux nombres de 1 des mots qui sont dans le système de contraintes d'adaptabilité. Pour contourner ce problème, nous n'allons pas travailler sur les mots périodiques, mais sur une abstraction de ces mots.

## 5. Abstraction de mots binaires

L'idée de l'abstraction des mots a été présentée l'an dernier dans [10]. Elle consiste à « encadrer » les fonctions de cumul par deux droites de même pente. Nous présentons dans cette section une version améliorée de [10]. L'abstraction d'un mot binaire infini  $w$  conserve uniquement le taux asymptotique  $r$  de 1 dans  $w$  et deux valeurs  $b^0$  et  $b^1$  qui donnent les décalages minimum et maximum du nombre de 1 vus dans  $w$  par rapport au taux  $r$ . On appelle *enveloppe* cette valeur abstraite et on la note  $\langle b^0, b^1 \rangle (r)$ .

**Définition 4** (concrétisation).

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

avec  $b^0, b^1, r \in \mathbb{Q}$  et  $0 \leq r \leq 1$ .

Reprenons les mots  $w_1 = (11010)$  et  $w_2 = 0(00111)$  que nous avons vu précédemment. Ces mots appartiennent respectivement aux concrétisations des valeurs abstraites  $a_1 = \langle 0, \frac{4}{5} \rangle \left( \frac{3}{5} \right)$  et

$a_2 = \langle -\frac{9}{5}, -\frac{3}{5} \rangle (\frac{3}{5})$ . Cet exemple est représenté figure 4. Dans les chronogrammes, une valeur abstraite  $\langle b^0, b^1 \rangle (r)$  est représentée par deux droites  $\Delta^1 : r \times i + b^1$  et  $\Delta^0 : r \times i + b^0$  qui bornent les fronts montants et les absences de front des mots qu'elle contient. La définition indique que tout front montant doit arriver sous la droite  $\Delta^1$  (représentée par une ligne continue) et toute absence de front doit se produire au dessus de la droite  $\Delta^0$  (représentée par une ligne en pointillés).

Pour que l'ensemble de mots défini par une enveloppe soit non vide, il faut que la droite  $\Delta^1$  soit au-dessus de  $\Delta^0$  et que l'écart entre ces droites soit suffisant.

**Proposition 1** (test de non vacuité).  $\forall a = \langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \rangle (\frac{n}{\ell}), \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow \text{concr}(a) \neq \emptyset$

L'abstraction d'un mot binaire ultimement périodique peut être automatiquement calculée. Le taux asymptotique  $r$  correspond au ratio entre le nombre de 1 du motif périodique et sa longueur. Il suffit ensuite de parcourir le mot en retenant les décalages minimum (lors de l'occurrence d'un 0) et maximum (lors de l'occurrence d'un 1) du nombre de 1 vus à l'instant  $i$  par rapport au nombre de 1 idéal  $r \times i$ . Ces décalages minimum et maximum sont atteints après le parcours du préfixe et un parcours du motif périodique.

**Définition 5** (fonction d'abstraction des mots binaires périodiques).

Soit  $p = u(v)$  un mot binaire périodique.  $\text{abs}(p) \stackrel{\text{def}}{=} \langle b^0, b^1 \rangle (r)$  avec :

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

où  $|u|$  représente la longueur d'un mot  $u$  et  $|u|_1$  son nombre de 1.

**Opérations et relations abstraites.** La force de l'abstraction est de ramener les opérations et les tests de relations sur les mots à quelques opérations arithmétiques simples sur les nombres rationnels. Par exemple, le calcul du  $\text{on}$  sur les valeurs abstraites se fait en trois multiplications et deux additions : <sup>3</sup>

**Définition 6** (opérateur  $\text{on}^\sim$ ). Soient  $b^0_1 \leq 0$  et  $b^0_2 \leq 0$ ,

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

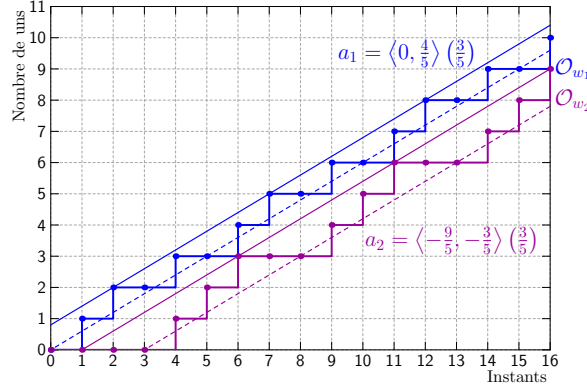
Définissons les relations sur les enveloppes correspondant aux relations sur les mots définies en section 3. Une relation est vérifiée sur des enveloppes si elle est vérifiée sur tout couple de mots appartenant aux concrétisations respectives de celles-ci.

**Définition 7** (synchronisabilité  $\bowtie^\sim$ , précédence  $\preceq^\sim$  et adaptabilité  $<:\sim$  abstraites).

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

Ces relations peuvent être testées par des comparaisons arithmétiques des taux et décalages.

<sup>3</sup>Plus d'explications sur la formule de  $\text{on}^\sim$  sont données dans [13]. Notamment, on peut toujours se ramener au cas  $b^0_1 \leq 0$  et  $b^0_2 \leq 0$  traité dans la définition.

FIG. 5 – L’enveloppe  $a_1$  est synchronisable avec l’enveloppe  $a_2$  et elle la précède.

**Proposition 2** (tests de synchronisabilité, précédence et adaptabilité).

Soient  $a_1 = \langle b^0_1, b^1_1 \rangle (r_1)$  et  $a_2 = \langle b^0_2, b^1_2 \rangle (r_2)$ . On a :

$$\begin{aligned} r_1 = r_2 & \Leftrightarrow a_1 \bowtie^{\sim} a_2 \\ b^1_2 - b^0_1 < 1 & \Rightarrow a_1 \preceq^{\sim} a_2 \quad \text{si } r_1 = r_2 \\ a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2 & \Leftrightarrow a_1 <^{\sim} a_2 \end{aligned}$$

L’observation de la figure 5 donne l’intuition de la correction de ces tests. Les mots des enveloppes  $a_1$  et  $a_2$  naviguent dans la zone située entre leurs deux droites respectives. Si ces droites sont de même pente, les mots restent à une distance bornée les uns des autres. Ils sont donc synchronisables. Si le chevauchement entre les deux enveloppes est suffisamment petit pour qu’il ne puisse pas y avoir un point du chronogramme où l’occurrence d’un 0 est permise dans  $a_1$  alors que l’occurrence d’un 1 est permise dans  $a_2$ , il ne peut y avoir d’entrelacement entre un mot de  $a_1$  et un mot de  $a_2$ . Les mots de  $a_1$  sont donc toujours au dessus de ceux de  $a_2$  et la relation de précédence est vérifiée.

**Taille du buffer.** Lorsque la relation d’adaptabilité entre deux horloges est vérifiée, on peut consommer au rythme de la seconde horloge un flot produit sur le rythme de la première horloge en introduisant un buffer de taille bornée. Une taille correcte pour le buffer établissant la communication peut elle aussi être calculée dans le domaine abstrait.

**Définition 8** (taille du buffer).

Soient  $a_1 = \langle b^0_1, b^1_1 \rangle (r_1)$  et  $a_2 = \langle b^0_2, b^1_2 \rangle (r_2)$  deux enveloppes telles que  $a_1 <^{\sim} a_2$ .

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

En effet, la taille de buffer nécessaire pour communiquer de toute horloge de  $a_1$  vers toute horloge de  $a_2$  est la taille de buffer nécessaire pour communiquer de l’horloge la plus précoce de  $a_1$  vers l’horloge la plus tardive de  $a_2$  (qui correspond approximativement à la distance verticale entre la droite  $\Delta^1$  de  $a_1$  et la droite  $\Delta^0$  de  $a_2$ ).

## 6. Résolution des contraintes de sous-typage

À l’issue du typage du corps d’un nœud, on obtient une contrainte de sous-typage par buffer du nœud. Pour pouvoir généraliser le type du nœud, il faut résoudre ce système de contraintes.

Par exemple, le type du nœud `good` avant généralisation est  $\alpha \rightarrow \alpha$  `on` (10) avec la contrainte  $\{\alpha$  `on` (10)  $<: \alpha$  `on` (01) $\}$  sur la variable  $\alpha$ . Quelle que soit l'instanciation de la variable de type  $\alpha$ , la contrainte est vérifiée si et seulement si (10)  $<:$  (01). Afin de vérifier cette contrainte d'adaptabilité, nous nous plaçons dans le domaine abstrait :

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

Les abstractions des mots (10) et (01) sont respectivement  $\langle 0, \frac{1}{2} \rangle (\frac{1}{2})$  et  $\langle -\frac{1}{2}, 0 \rangle (\frac{1}{2})$ . Ainsi, par application de la proposition 2, on a :

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

La contrainte de sous-typage du nœud `good` est toujours vérifiée. Son type est donc  $\forall \alpha. \alpha \rightarrow \alpha$  `on` (10).

Ici, la résolution du système s'est résumée à de la vérification car les deux types reliés par la contrainte de sous-typage étaient exprimés en fonction de la même variable de type. Ce n'est pas toujours aussi simple. Par exemple, considérons le nœud suivant :

```

1 let node f (x, y, z) =
2   buffer (x when (11010))
3   + y when 0(00111)
4   + buffer (z when (01))
5   + buffer (z when 0(1100))

```

Le type de ce nœud avant généralisation est  $\alpha_1 \times \alpha_2 \times \alpha_3 \rightarrow \alpha_2$  `on` 0(00111) avec le système de contraintes  $C$  suivant :

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

En fonction des instanciations des variables de type  $\alpha_1$ ,  $\alpha_2$  et  $\alpha_3$ , ces contraintes peuvent ou non être vérifiées. Résoudre le système  $C$  consiste à trouver une substitution pour les variables assurant que les contraintes sont toujours vérifiées. Pour cela, il faut exprimer tous les types en fonction de la même variable. On pose  $\alpha_1 = \alpha$  `on`  $c_1$ ,  $\alpha_2 = \alpha$  `on`  $c_2$ ,  $\alpha_3 = \alpha$  `on`  $c_3$  avec  $c_1$ ,  $c_2$  et  $c_3$  des variables d'horloge dont on cherche des valeurs telles que le système suivant soit satisfait :

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

Ainsi, on a traduit le système de contrainte de sous-typage en un système de contraintes d'adaptabilité. Pour résoudre ce système, on peut chercher une solution dans le domaine abstrait :

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

Considérons la deuxième contrainte de ce dernier système. Si on pose  $\text{abs}(c_2) = \langle b^0_2, b^1_2 \rangle (r_2)$  et  $\text{abs}(c_3) = \langle b^0_3, b^1_3 \rangle (r_3)$  et on calcule les valeurs de  $\text{abs}((01)) = \langle -\frac{1}{2}, 0 \rangle (\frac{1}{2})$  et  $\text{abs}(0(00111)) = \langle -\frac{9}{5}, -\frac{3}{5} \rangle (\frac{3}{5})$ , alors on peut appliquer la définition de `on` $\sim$  pour réécrire cette deuxième contrainte en :

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

Puis, en utilisant la proposition 2, elle peut être décomposée en une contrainte de synchronisabilité ( $r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5}$ ) et une contrainte de précédence ( $(b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1$ ).

Si on applique ces transformations aux deux autres contraintes, on peut alors transformer le système de contraintes d'adaptabilité abstraites en deux systèmes de contraintes :

$$\text{synchronisabilité} \left\{ \begin{array}{l} r_1 \times \frac{3}{5} = r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5} \\ r_3 \times \frac{1}{2} = r_2 \times \frac{3}{5} \end{array} \right\} \text{ et précédence} \left\{ \begin{array}{l} (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_1 \times \frac{3}{5} + 0) < 1 \\ (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1 \\ (b^1_2 \times \frac{3}{5} - \frac{3}{5}) - (b^0_3 \times \frac{1}{2} - \frac{1}{2}) < 1 \end{array} \right\}$$

Dans le système de contraintes de synchronisabilité, on cherche des valeurs correctes pour les taux. Les  $r_i$  doivent donc être compris entre zéro et un d'après la définition 4. Pour résoudre ce système, on commence par réécrire chaque contrainte  $r_i \times q_i = r_j \times q_j$  sous la forme  $r_i = \frac{q_j}{q_i} \times r_j$  avec  $\frac{q_j}{q_i} \leq 1$ . Puis, on sature le système pour exprimer toutes les contraintes en fonction de la même variable. Enfin, on choisit le taux 1 pour cette variable, afin de maximiser le rythme d'exécution. Dans notre exemple, la solution du système que l'on obtient est  $r_1 = \frac{5}{6}$ ,  $r_2 = \frac{5}{6}$  et  $r_3 = 1$ .

Le système de contraintes de précédence nous permet de trouver les valeurs des  $b^0_i$  et  $b^1_i$ . Afin de ne chercher que des enveloppes non vides, nous ajoutons à ces contraintes des contraintes de non vacuité comme définies dans la proposition 1. Nous devons aussi ajouter les contraintes que les  $b^0_i$  sont négatifs ou nuls car nous avons appliqué la définition 6 pour calculer l'opération  $on \sim$ .

Le système de contraintes ainsi obtenu peut se ramener à un système d'inéquations linéaires qui peut être résolu avec un outil tel que GLPK [7]. On trouve le résultat suivant comme solution au système de contraintes d'adaptabilité abstraites :

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

Par définition de la relation  $<: \sim$ , tout mot dans les enveloppes que nous venons d'inférer sont des solutions du système de contraintes d'adaptabilité d'origine. Donc par exemple, les mots  $c_1 = (011111)$ ,  $c_2 = (111110)$  et  $c_3 = (1)$  sont une solution. En appliquant la substitution  $\{\alpha_1 \leftarrow \alpha \text{ on } c_1, \alpha_2 \leftarrow \alpha \text{ on } c_2, \alpha_3 \leftarrow \alpha \text{ on } c_3\}$  dans le type de  $\mathbf{f}$ , on obtient un type qui sera toujours correct quelle que soit l'instanciation de  $\alpha$ . Le type ainsi obtenu est :

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

**Taille de buffer.** Une fois le système de contraintes résolu, les types qui étaient à gauche et à droite de chaque contrainte de sous-typage sont exprimés en fonction de la même variable d'horloge. Ces types correspondant aux types des flots d'entrée et de sortie des buffers, on peut maintenant calculer la taille des buffers. Sur l'exemple du nœud  $\mathbf{f}$ , les buffers lignes 2, 4 et 5 sont respectivement de taille 2, 1 et 2.

## 7. Implantation

Nous avons programmé en OCAML un typeur pour le langage LUCY-N. Sa particularité est d'être paramétré par le langage d'horloges à utiliser. En effet, comme nous l'avons dit dans la section 2, les expressions d'horloges peuvent être définies avec tout langage d'horloges s'évaluant vers des mots binaires infinis. Ainsi, le typeur implante le système de type manipulant les horloges présentées dans cet article mais pas seulement.

Le typeur est un foncteur paramétré par un module qui définit le langage d'horloges et qui contient les fonctions suivantes :

**Égalité :** Une fonction `equal` qui teste l'égalité de deux horloges et une fonction `unify` qui à partir des deux expressions d'horloges  $ce_1$  et  $ce_2$  retourne deux nouvelles expressions  $ce'_1$  et  $ce'_2$  telles que  $ce'_1 \text{ on } ce_1 = ce'_2 \text{ on } ce_2$ .

**Adaptabilité :** Une fonction `adaptable` qui teste l'adaptabilité de deux horloges et une fonction `solve` qui à partir d'un système de contraintes d'adaptabilité retourne une instanciation des variables qui satisfait le système.

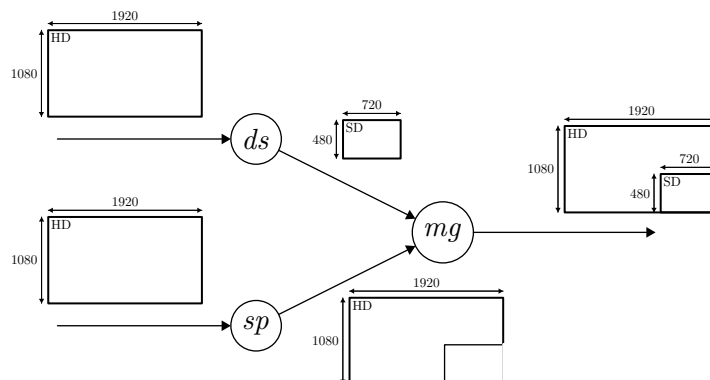


FIG. 6 – *Picture in Picture* : Les entrées de l’application sont deux vidéos Haute Définition qui arrivent sous forme de flots de pixels. La taille de la première est diminuée en utilisant un *Downscaler*. Des pixels de la seconde sont supprimés par échantillonnage. Puis les deux vidéos sont fusionnées.

**Taille de buffer** : Enfin, une fonction `buffer_size` qui à partir de deux horloges adaptables retourne la taille de buffer suffisante pour faire communiquer des flots qui ont ces horloges.

Pour le moment, deux langages d’horloges ont été expérimentés : les horloges périodiques et les horloges abstraites. Mais pour chacun de ces langages, il peut y avoir plusieurs combinaisons des fonctions d’égalité et d’adaptabilité. Ce que nous avons présenté ici est la combinaison du langage des horloges périodiques (`pbw`) avec de l’égalité interprétée (`interp`) [5] et de l’adaptabilité sur des horloges abstraites (`abs`). Les exemples de cet article ont donc été typés en tapant la commande : `lucync -ce pbw -unif interp -solver abs fichier.ls`.

## 8. Application : le *Picture in Picture*

Montrons sur un exemple d’application multimédia, le *Picture in Picture*, le comportement du typeur de LUCY-N. Cette application est décrite sur la figure 6. Elle est programmée de la manière suivante par le nœud `picture_in_picture_end`.

```

52 let clock incrust_end =
53   (0^(1920 * (1080 - 480)) {0^1200 1^720}^480)
54
55 let node picture_in_picture_end (p1, p2) = o where
56   rec small = buffer(downscaler p1)
57   and big = (p2 whenot incrust_end)
58   and o = merge incrust_end small big

```

L’horloge `incrust_end` détermine la provenance des pixels de l’image résultat, dans le cas où l’on veut incruster la petite image en bas à droite de la grande, comme sur la figure 6. Elle vaut 1 quand c’est la petite image qu’il faut afficher, et 0 quand c’est la grande. La notation `{0^1200 1^720}^480` est un raccourci pour la répétition de 480 fois le motif `0^1200 1^720`. La petite image est obtenue par application du nœud `downscaler`, dont le code est fourni en annexe. Il est composé d’un filtre horizontal, qui applique une convolution puis un échantillonnage par une horloge `hf`, réduisant la taille des lignes de 1920 vers 720, et d’un filtre vertical effectuant lui aussi une convolution et réduisant par échantillonnage par une horloge `vf` la taille des colonnes de 1080 vers 480. Pour effectuer la convolution, le filtre vertical a besoin de disposer des pixels situés au dessus et en dessous du pixel traité, ce qui implique que la production des pixels de sortie commence une ligne (720 pixels) après la consommation des pixels en entrée. Le type d’horloges inféré pour le nœud `downscaler` est :

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

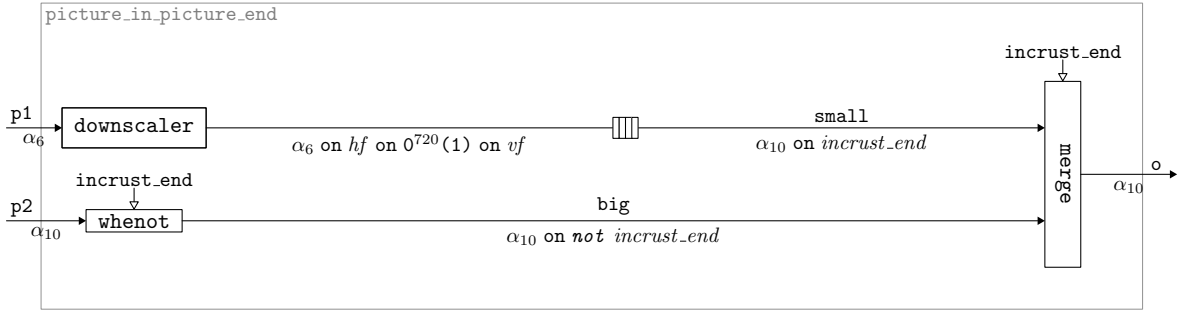


FIG. 7 – Le nœud `picture_in_picture_end`. Les types d’horloges avant résolution des contraintes de sous-typage sont indiqués sous les fils.

Comme la sortie du nœud `downscaler` est connectée à l’entrée du nœud de fusion par l’intermédiaire d’un buffer (voir la figure 7), le type du nœud `picture_in_picture_end` avant généralisation est  $(\alpha_6 \times \alpha_{10}) \rightarrow \alpha_{10}$ , avec la contrainte suivante sur  $\alpha_6$  et  $\alpha_{10}$  (notées ‘a6 et ‘a10) :

```
{ 'a6 on hf on 0^720 (1) on vf <: 'a10 on incrust_end (* line 56, characters 14-35 *) }
```

Le résultat de l’abstraction de `hf on 0^720 (1) on vf` est  $\langle -720, \frac{481}{3} \rangle (\frac{1}{6})$  et celui de l’abstraction de `incrust_end` est  $\langle -192200, 0 \rangle (\frac{1}{6})$ . Trouver une solution au système de contraintes ci-dessus revient par conséquent à trouver des enveloppes `h6` et `h10` qui satisfassent le système :

```
{ h6 on~ <-720, 481/3>(1/6) <:~ h10 on~ <-192200, 0>(1/6) }
```

Les valeurs calculées pour `h6` et `h10` sont  $\langle 0, 0 \rangle (1)$  et  $\langle -4315, -4315 \rangle (1)$ . Donc le type inféré est :

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

Cela signifie que si l’image destinée à être réduite arrive en continu à partir du premier instant, l’image destinée à recevoir l’incrustation doit arriver en continu après un délai de 4315 instants, et l’image résultat sera produite en continu après ce même délai, soit approximativement le temps de recevoir deux lignes d’images Haute Définition. Nous avons vu que le nœud `downscaler` introduit une ligne de délai, et le nœud de fusion n’en introduit pas. Le type inféré est donc correct, mais il surestime d’un peu plus d’une ligne le délai nécessaire avant la production des premières sorties. Ceci est lié à l’abstraction des horloges. Cependant, cette méthode est ici satisfaisante, car la méthode de résolution sans abstraction existante nécessite une journée de calcul.

La taille de buffer calculée est très satisfaisante : 192 240 places nécessaires contre 191 970 pour la résolution sans abstraction, soit un surcoût de stockage de moins d’une ligne de petite image.

## 9. Conclusion

Le générateur de code pour LUCY-N n’a pas encore été implanté. Néanmoins, grâce aux informations de typage, la traduction des programmes n-synchrones en des programmes purement synchrones sera aisée. Les buffers pourront être implantés comme des nœuds synchrones qui sont paramétrés par leur taille et leurs horloges.

Pour le moment LUCY-N est un prototype qui permet d’expérimenter des calculs d’horloge pour la programmation dans le modèle n-synchrone. Cet aspect expérimental se reflète dans son implantation sous forme de foncteur. Dans cet article, nous avons présenté un langage d’horloges périodiques et un système de résolution de contraintes qui manipule les horloges abstraites de [10] pour LUCY-N. L’architecture choisie nous permettra de mettre en place facilement d’autres langages d’horloges et systèmes de résolution associés.

## Références

- [1] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
- [2] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley, 1993.
- [3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM, 1987.
- [4] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, May 1996.
- [5] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems*. In *ACM International Conference on Principles of Programming Languages*, January 2006.
- [6] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, October 2003.
- [7] Glpk. Gnu linear programming kit. <http://www.gnu.org/software/glpk/>.
- [8] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974.
- [9] E. Lee and D. Messerschmitt. Synchronous dataflow. *IEEE Trans. Comput.*, 75(9), 1987.
- [10] L. Mandel and F. Plateau. Abstraction d’horloges dans les systèmes synchrones flot de données. In *Vingtièmes Journées Francophones des Langages Applicatifs*, February 2009.
- [11] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.
- [12] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA, 1995.
- [13] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris-Sud 11, January 2010.
- [14] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: [www.lri.fr/~pouzet/lucid-synchrone](http://www.lri.fr/~pouzet/lucid-synchrone).

## A. Code du *Downscaler*

```

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

```