Abstraction d'horloges dans les systèmes synchrones

Marc Pouzet
LRI
Univ. Paris-Sud 11

Journées AFSEC – 26/01/2009

(travail réalisé avec Albert Cohen, Louis Mandel et Florence Plateau)

Langages dataflow synchrones

Modéliser/programmer le logiciel embarqué critique.

L'idée de Lustre :

- écrire directement des équations de suites vues comme des **spécifications exécutables**
- fournir un **compilateur** et des outils d'analyse dédiés pour produire du code

E.g, le filtre linéaire défini par :

$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$

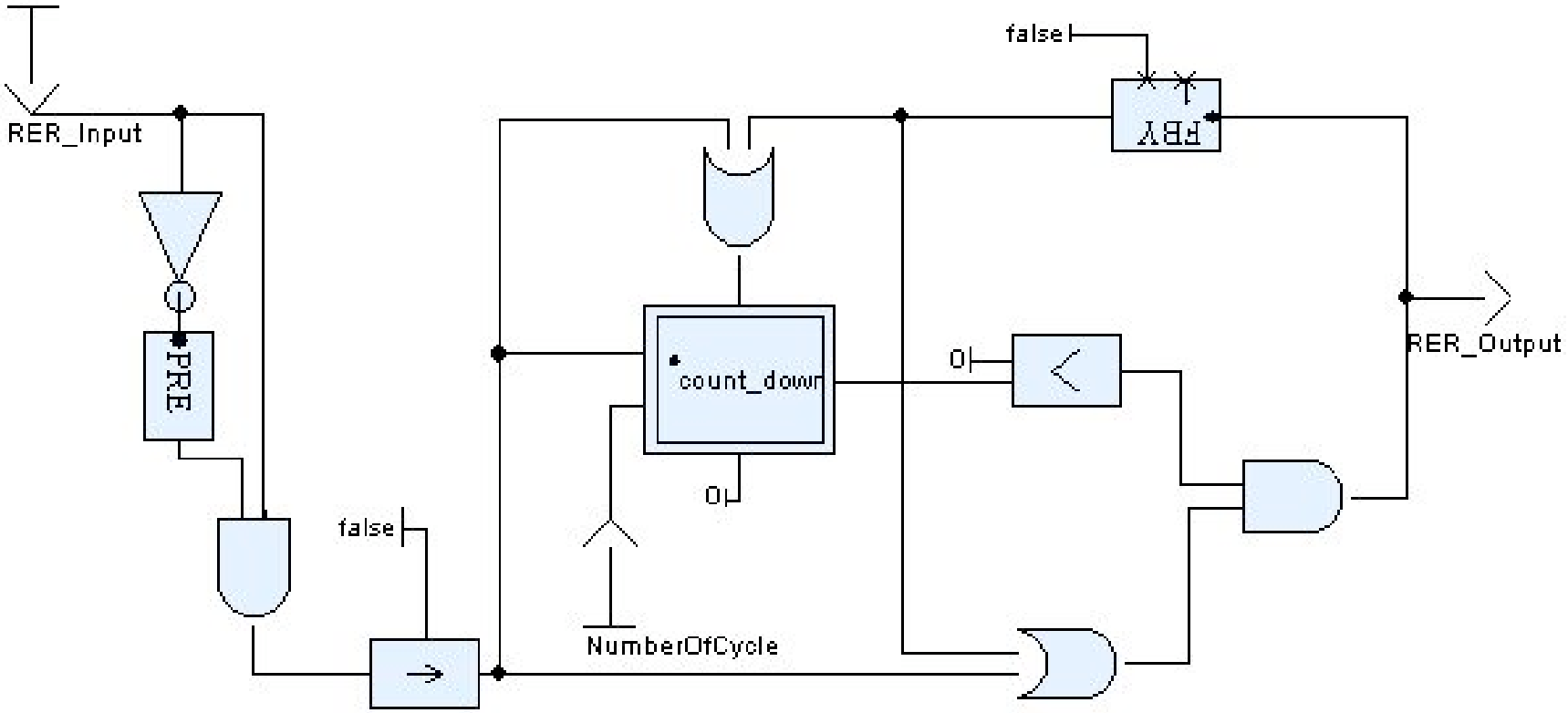
est programmé en écrivant :

```
Y = (0 -> a * pre(Y)) + b * X
```

on écrit des **invariants**

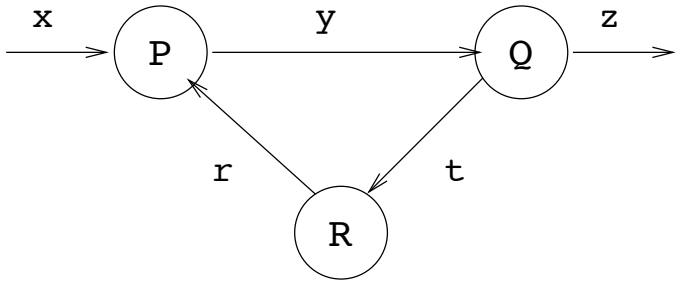
- d'autres opérateurs permettent de composer des processus lents et rapides (sous/sur-échantillonnage) ; non nécessairement périodique

Un exemple de planche SCADE



Sémantique dataflow

Principe de Kahn [IFIP'74] : Quelle est la sémantique d'un réseau de processus communiquant par FIFOs non-bornés (e.g., Unix pipe, sockets) ?



- communication par envoi de message (`send/wait`) dans des FIFOs
- canaux fiables, délais de communication bornés
- attente sur un canal seulement. Le programme suivant est **interdit**

```
if (A is present) or (B is present) then ...
```

- un processus = fonction continue de $(V^{\infty})^n \rightarrow (V'^{\infty})^m$.

Lustre :

- Lustre a une **sémantique de Kahn** (pas de test a l'absence)
- Un **système de types** dédié (calcul d'horloge) garantit l'existence d'une exécution synchrone sans buffer

Intérêt/Faiblesse du modèle

(+) : Sémantique simple : un processus définit une fonction (déterminisme) ; la composition est la composition de fonctions ;

(+) : Modularité : un réseau définit une fonction continue ; clos par composition et rebouclage

(+) : Invariance temporelle : pas de temps explicite ; sémantique invariante par ralentissement/accelération

(+) : Exécution distribuée asynchrone : ordonnanceur centralisé inutile

x	$=$	x_0		x_1		x_2	x_3	x_4	x_5	...
$f(x)$	$=$	y_0		y_1		y_2	y_3	y_4	y_5	...
$f(x)$	$=$	y_0			y_1	y_2		y_3		y_4 y_5 ...

Un modèle naturel pour le traitement vidéo (TV boxes) : Sally (Philips NatLabs), StreamIt (MIT), Xstream (ST-micro) et restrictions synchrones *à la SDF* (Ptolemy)

Un petit noyau dataflow

Un langage noyau muni de primitives dataflow

$$
\begin{array}{rcl}
e & ::= & e\ \texttt{fby}\ e \mid op(e, ..., e) \mid x \mid i \\
 & & \mid \texttt{merge}\ e\ e\ e \mid e\ \texttt{when}\ e \\
 & & \mid \lambda x.e \mid e(e) \mid \texttt{rec}\ x.e \\
op & ::= & + \mid - \mid \texttt{not} \mid ...
\end{array}
$$

- fonction ($\lambda x.e$), application ($e(e)$), point-fixe ($\texttt{rec}\ x.e$)
- constantes i et variables (x)
- primitives dataflow : $x\ \texttt{fby}\ y$ est le délai initialisé ; $op(e_1, ..., e_n)$ l'application point-à-point ; échantillonnage (`when`/`merge`).

Primitives Dataflow

x	x_0	x_1	x_2	x_3	x_4	x_5
y	y_0	y_1	y_2	y_3	y_4	y_5
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
x `fby` y	x_0	y_0	y_1	y_2	y_3	y_4
h	1	0	1	0	1	0
x `when` h	x_0		x_2		x_4	
z		z_0		z_1		z_2
`merge` h x z	x_0	z_0	x_2	z_1	x_4	z_3

Echantillonnage :

- si h est une séquence booléenne, x `when` h produit une sous-suite de x
- `merge` h x z combine deux sous-suites

Sémantique de Kahn

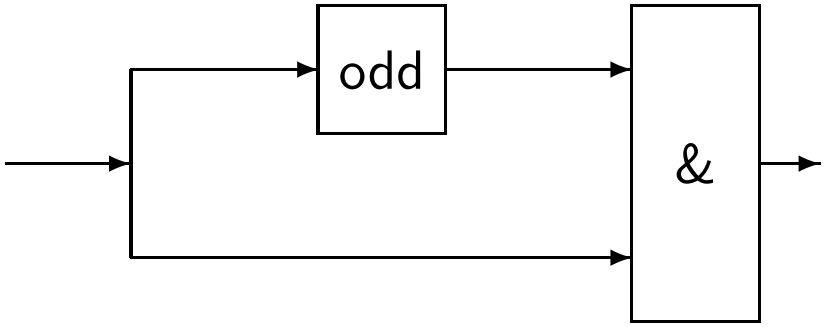
Chaque opérateur est interprété par une fonction sur les suites ($V^\infty = V^* + V^\omega$). E.g., si $x \mapsto s_1$ et $y \mapsto s_2$ alors la valeur de $x + y$ est $+^\# (s_1, s_2)$

$$
\begin{array}{lcl}
i^\# & = & i.i^\# \\
+^\# (x.s_1, y.s_2) & = & (x+y).+^\# (s_1, s_2) \\
(x.s_1)\,\texttt{fby}^\#\, s_2 & = & x.s_2 \\
x.s\,\texttt{when}^\#\, 1.c & = & x.(s\,\texttt{when}^\#\, c) \\
x.s\,\texttt{when}^\#\, 0.c & = & s\,\texttt{when}^\#\, c \\
\texttt{merge}^\#\, 1.c\, x.s_1\, s_2 & = & x.\texttt{merge}^\#\, c\, s_1\, s_2 \\
\texttt{merge}^\#\, 0.c\, s_1\, y.s_2 & = & y.\texttt{merge}^\#\, c\, s_1\, s_2
\end{array}
$$

Propriété : fonctions continues (ordre préfixe) ; conservé par composition.

Synchronisme

Que se passe t'il lors de la composition de suites produites à des rythmes différents ?



Si $x = (x_i)_{i \in \mathbb{N}}$ alors $\mathsf{odd}(x) = (x_{2i})_{i \in \mathbb{N}}$ et $x \& \mathsf{odd}(x) = (x_i \& x_{2i})_{i \in \mathbb{N}}$.

FIFOs non bornées !

- Ces programmes doivent être détectés et rejetés statiquement
- chaque opérateur est à mémoire finie alors que la composition ne l'est pas : toute la complexité (synchronisation) est cachée dans les canaux de communication
- La sémantique de Kahn ne modélise pas le temps, i.e., deux événements arrivent **en même temps**

Flots synchrones

Compléter les suites avec une représentation explicite de l'absence abs.

$$x : (V^{abs})^{\infty}$$

Horloge : l'horloge de x est une suite booléenne

$$\mathbb{B} = \{0, 1\}$$

$$\mathcal{CLOCK} = \mathbb{B}^{\infty}$$

$$\begin{array}{lcl} \texttt{clock}\ \epsilon & = & \epsilon \\ \texttt{clock}\ (abs.x) & = & 0.\texttt{clock}\ x \\ \texttt{clock}\ (v.x) & = & 1.\texttt{clock}\ x \end{array}$$

Suites synchrones :

$$ClStream(V, cl) = \{s / s \in (V^{abs})^{\infty} \wedge \texttt{clock}\ s \leq_{prefix} cl\}$$

Autre codage possible : $x : (V \times \mathbb{N})^{\infty}$

Data-flow Primitives

Constante :

$$\begin{aligned} i^{\#}(\epsilon) &= \epsilon \\ i^{\#}(1.cl) &= i.i^{\#}(cl) \\ i^{\#}(0.cl) &= abs.i^{\#}(cl) \end{aligned}$$

Application point-à-point :

Les arguments doivent être synchrones, i.e., de même horloge

$$\begin{aligned} +^{\#}(s_1, s_2) &= \epsilon \text{ if } s_i = \epsilon \\ +^{\#}(abs.s_1, abs.s_2) &= abs.+^{\#}(s_1, s_2) \\ +^{\#}(v_1.s_1, v_2.s_2) &= (v_1 + v_2).+^{\#}(s_1, s_2) \end{aligned}$$

Définitions partielles

Ces fonctions ne sont pas totales. Que se passe t'il lorsqu'un élément est présent et l'autre absent ?

Contraindre le domaine :

$$(+) : \forall cl : \mathcal{CLOCK}.\, ClStream(\texttt{int}, cl) \times ClStream(\texttt{int}, cl) \rightarrow ClStream(\texttt{int}, cl)$$

i.e., $(+)$ est une fonction attend deux entrées de même horloge cl et produit une suite d'horloge cl.

Ces conditions supplémentaires sont des **types** : les programmes ne les respectant pas sont rejetés.

Remarque : Les types et les (types d') horloges peuvent être spécifés séparément :

- $(+) : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ ← **son type**
- $(+) :: \forall cl.cl \times cl \rightarrow cl$ ← **son horloge**

Dans la suite, nous considérerons seulement l'horloge.

Echantillonnage

$$
\begin{aligned}
s_1 \,\texttt{when}^{\#}\, s_2 &= \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(abs.s) \,\texttt{when}^{\#}\, (abs.c) &= abs.s \,\texttt{when}^{\#}\, c \\
(v.s) \,\texttt{when}^{\#}\, (1.c) &= v.s \,\texttt{when}^{\#}\, c \\
(v.s) \,\texttt{when}^{\#}\, (0.c) &= abs.x \,\texttt{when}^{\#}\, c
\end{aligned}
$$

$$
\begin{aligned}
\texttt{merge}\, c\, s_1\, s_2 &= \epsilon \text{ if one of the } s_i = \epsilon \\
\texttt{merge}\, (abs.c)\, (abs.s_1)\, (abs.s_2) &= abs.\texttt{merge}\, c\, s_1\, s_2 \\
\texttt{merge}\, (1.c)\, (v.s_1)\, (abs.s_2) &= v.\texttt{merge}\, c\, s_1\, s_2 \\
\texttt{merge}\, (0.c)\, (abs.s_1)\, (v.s_2) &= v.\texttt{merge}\, c\, s_1\, s_2
\end{aligned}
$$

Exemples

$base = (1)$	1	1	1	1	1	1	1	1	1	1	1	1	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	...
$h = (10)$	1	0	1	0	1	0	1	0	1	0	1	0	...
$y = x$ `when` h	x_0		x_2		x_4		x_6		x_8		x_{10}	x_{11}	...
$h' = (100)$	1		0		0		1		0		0	1	...
$z = y$ `when` h'	x_0						x_6					x_{11}	...
k			k_0		k_1				k_2		k_3		...
`merge` h' z k	x_0		k_0		k_1		x_6		k_2		k_3		...

```
let clock five =
  let rec f = true fby false fby false fby false fby f in f
let node stutter x = o where
  rec o = merge five x ((0 fby o) whenot five) in o
```

`stutter`$(nat) = 0.0.0.0.1.1.1.1.2.2.2.2.3.3...$

Echantillonnage et horloges

- $x\,\texttt{when}^{\#}\,y$ est défini lorsque x et y ont la même horloge cl
- l'horloge de $x\,\texttt{when}^{\#}\,c$ est notée $cl\,\texttt{on}\,c$: "c avance au rythme de cl"

$$
\begin{array}{lcl}
s\,\texttt{on}\,c & = & \epsilon \text{ if } s = \epsilon \text{ or } c = \epsilon \\
(1.cl)\,\texttt{on}\,(1.c) & = & 1.cl\,\texttt{on}\,c \\
(1.cl)\,\texttt{on}\,(0.c) & = & 0.cl\,\texttt{on}\,c \\
(0.cl)\,\texttt{on}\,(abs.c) & = & 0.cl\,\texttt{on}\,c
\end{array}
$$

On obtient :

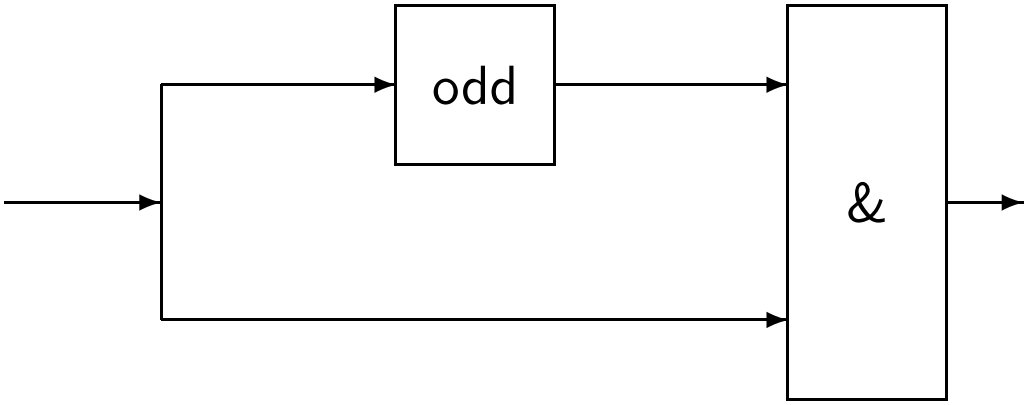
$$\texttt{when} : \forall cl.\forall x : cl.\forall c : cl.cl\,\texttt{on}\,c$$

$$\texttt{merge} : \forall cl.\forall c : cl.\forall x : cl\,\texttt{on}\,c.\forall y : cl\,\texttt{on}\,\texttt{not}\,c.cl$$

Pour tout cl, si l'entrée x est d'horloge cl et la seconde c est d'horloge cl alors $x\,\texttt{when}\,c$ est d'horloge $cl\,\texttt{on}\,c$.

Vérifier le synchronisme

Le programme précedent est maintenant rejeté statiquement.



C'est un problème de **typage**

```
let odd x = x when half
let non_synchronous x = x & (odd x)
                            ^^^^^^^

This expression has clock 'a on half,
but is used with clock 'a
```

Dans les langages synchrones (e.g., Lustre, Signal), on considère seulement **l'égalité d'horloges**

Du synchrone pur au N-synchrone

- La comparaison d'horloges est limitée à l'égalité, i.e., "deux flots composés sont synchrones ou pas"
- Peut-on composer des flots non exactement synchrone mais d'horloges "proches" ?
- Prendre en compte des phénomènes e gîgues, modéliser des temps d'exécution en restant conservatif
- Donner plus de liberté au compilateur, générer (si besoin) du code synchrone

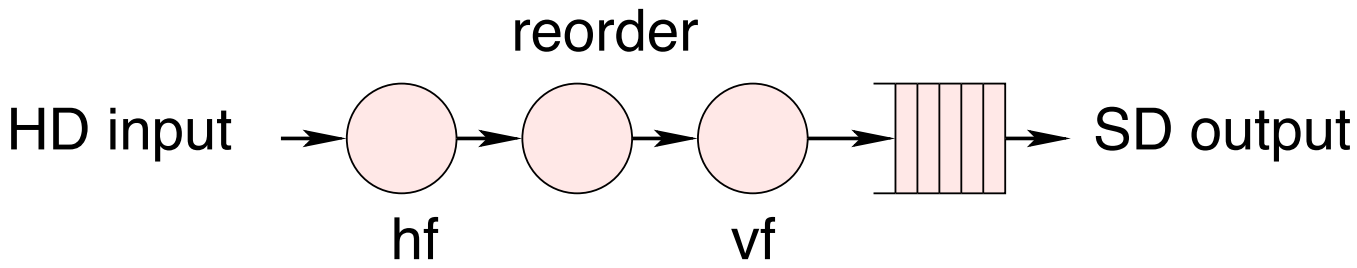
Un exemple typique : Downscaler

image haute résolution (HD) $\rightarrow$ définition standard (SD)

1920×1080 pixels 720×480

filtre horizontal : nombre de pixels par lignes de 1920 pixels à 720 pixels,

filtre vertical : nombre de lignes de 1080 à 480



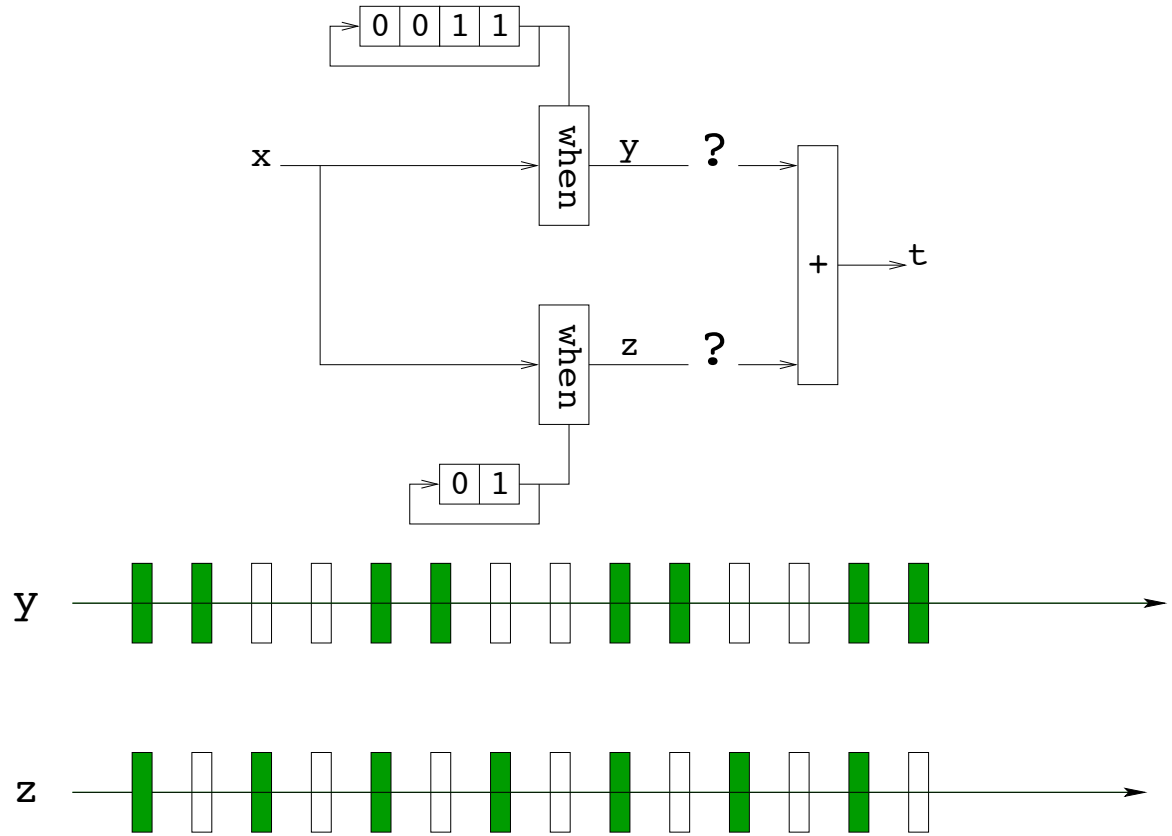
Contraintes de temps-réel

les processus d'entrée/sortie : 30Hz.

HD pixels arrivent au rythme $30 \times 1920 \times 1080 = 62,208,000 Hz$

SD pixels au rythme $30 \times 720 \times 480 = 10,368,000 Hz$ (6 fois plus lent)

Trop restrictif pour les applications vidéo

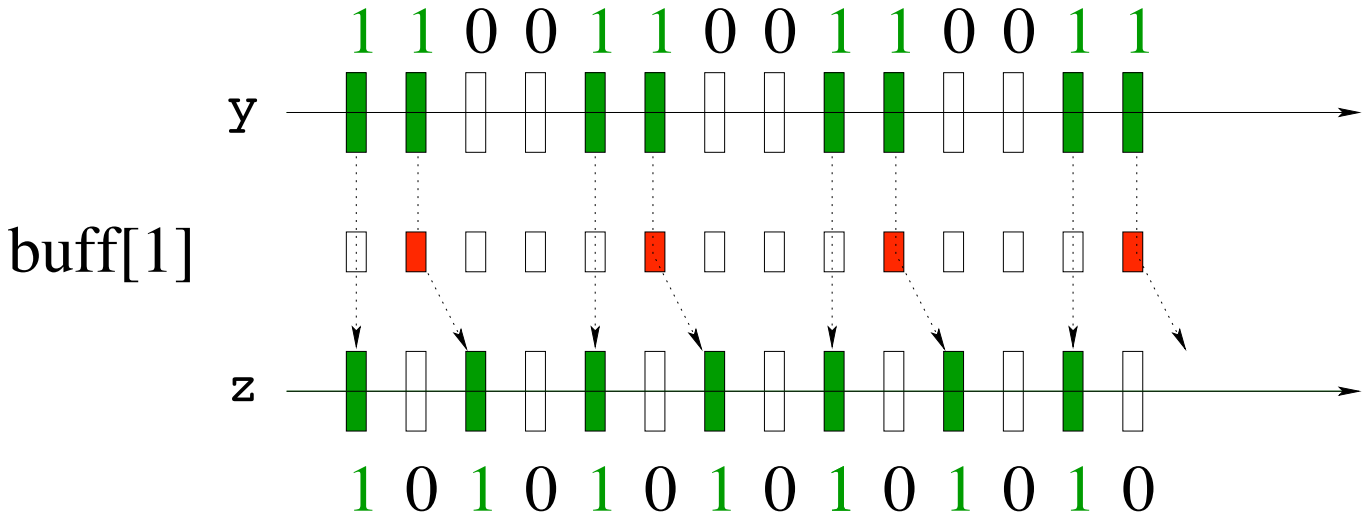


- les flots doivent être synchrones lors de la composition
- l'ajout de buffer (à la main) difficile et source d'erreurs
- le calculer automatiquement et générer du code synchrone ?

relacher la contrainte de synchronisme et les règles d'horloge associées

Réseaux de Kahn N-Synchrones

- proposer un modèle basé sur une notion de synchronisme relaché
- compilable vers du code synchrone
- composer des programmes qui peuvent être rendus synchrones par l'ajout de buffers



– basé sur l'utilisation de *suites infinies ultimatemment périodiques*

– une relation de précédence entre horloges $ck_1 <: ck_2$

Suites binaires infinies ultimement périodiques

$\mathbb{Q}_2$ pour l'ensemble des infinite periodic binary words.

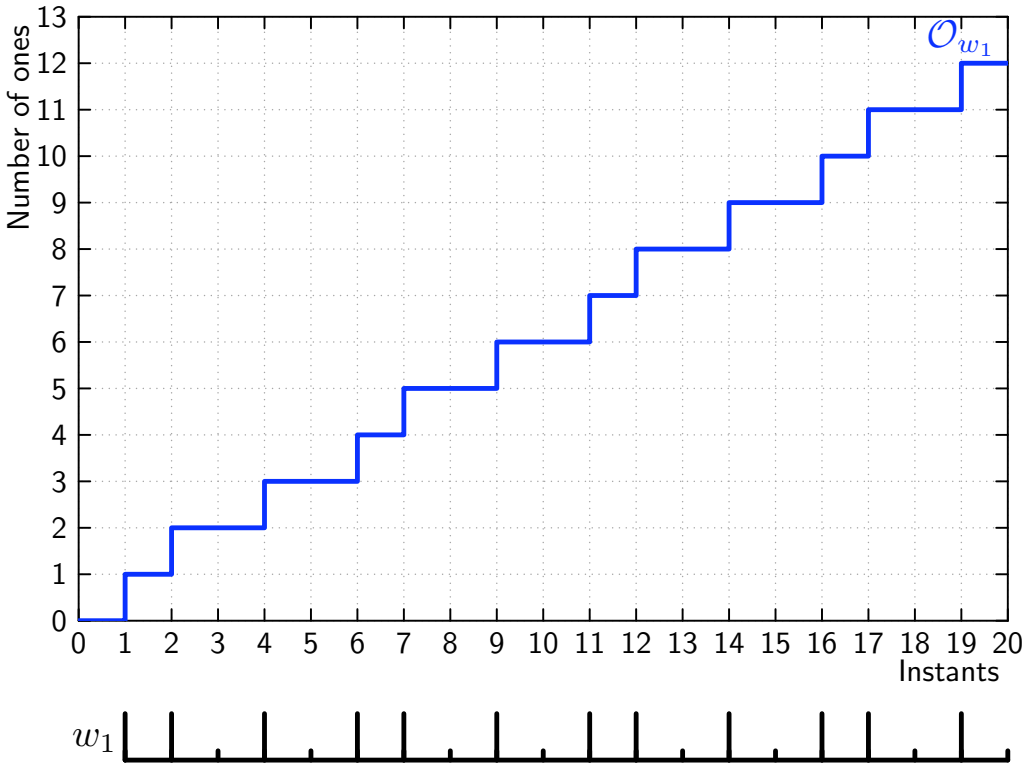
$$\begin{aligned} (01) &= 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ 01\ldots \\ 0(1101) &= 0\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101\ 1101\ldots \end{aligned}$$

– 1 pour la présence
– 0 pour l'absence

Définition :

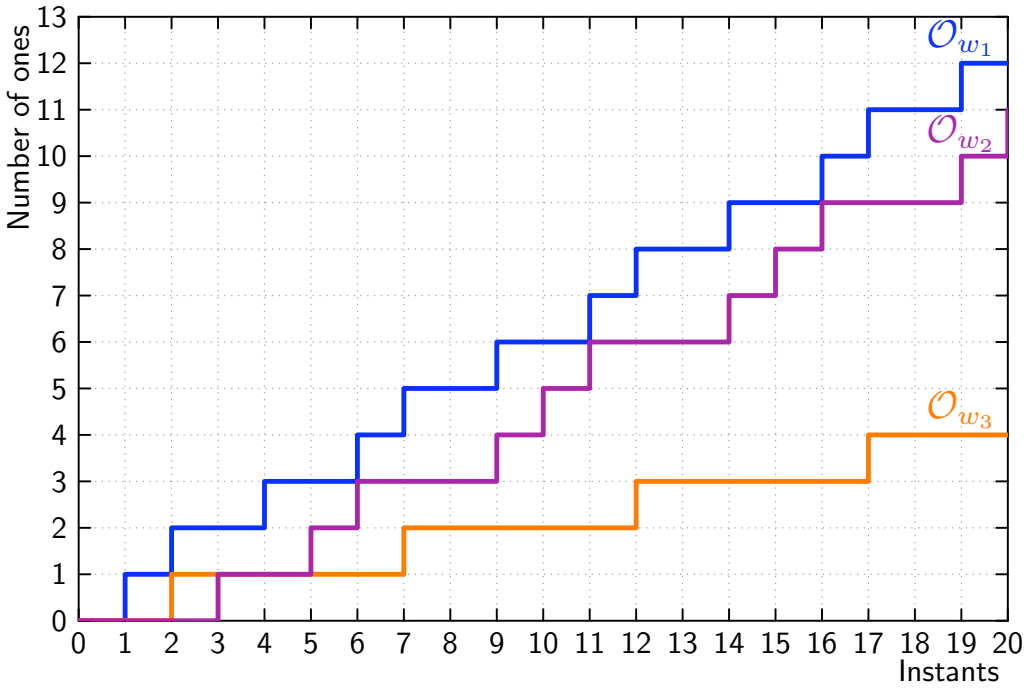
$$w ::= u(v) \quad \text{where } u \in (0+1)^* \text{ and } v \in (0+1)^+$$

Horloges et mots binaires infinis



$\mathcal{O}_w(i) =$ fonction de cumul des 1 de w

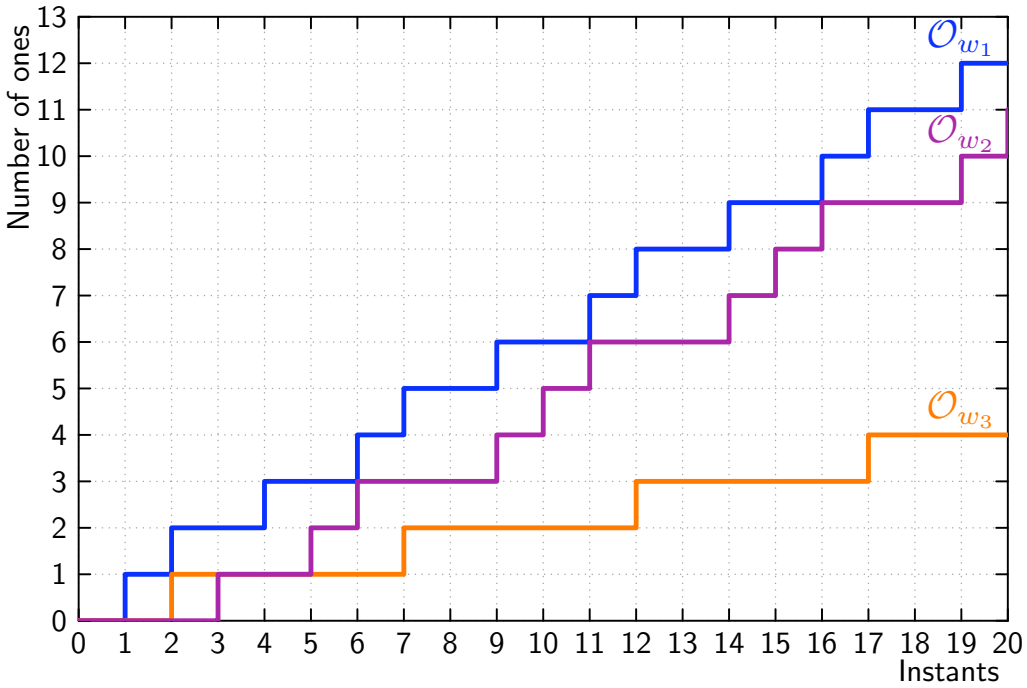
Horloges et mots binaires infinis



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

sous-typage $w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$

Horloges et mots binaires infinis



buffer	$size(w_1, w_2) = \max_{i \in \mathbb{N}}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$
sous-typage	$w_1 <: w_2 \overset{def}{\Leftrightarrow} \exists n \in \mathbb{N}, \forall i,\ 0 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq n$
synchronisabilité	$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z}, \forall i,\ b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$
précédence	$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall i,\ \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$

Systèmes multi-horloge ("clock gating")

$$c \quad ::= \quad w \mid c \text{ on } w \qquad w \in (0+1)^\omega$$

c on w est une **sous-horloge** de c, obtenue en avançant dans w au rythme de c.
E.g., $1(10)$ on $(01) = (0100)$.

base	1	1	1	1	1	1	1	1	1	1	...	(1)
p_1	1	1	0	1	0	1	0	1	0	1	...	$1(10)$
base on p_1	1	1	0	1	0	1	0	1	0	1	...	$1(10)$
p_2	0	1		0		1		0		1	...	(01)
(base on p_1) on p_2	0	1	0	0	0	1	0	0	0	1	...	(0100)

Pour les horloges ultimement périodiques, la précédence, la synchronisabilité et l'égalité sont décidables (mais cher)

Retour au langage

Synchrone pur :

- se ramène à un système de type *à la ML* (e.g., SCADE 6)
- égalité structurelle (horloges non nécessairement périodiques)

$$\frac{H \vdash e_1 : ck \qquad H \vdash e_2 : ck}{H \vdash op(e_1, e_2) : ck}$$

N-Synchrone :

- on ajoute une règle de **sous-typage** :

$$\text{(SUB)} \quad \frac{H \vdash e : ck \text{ on } w \quad w <: w'}{H \vdash e : ck \text{ on } w'}$$

- définit les points de synchronisation où un buffer doit être utilisé

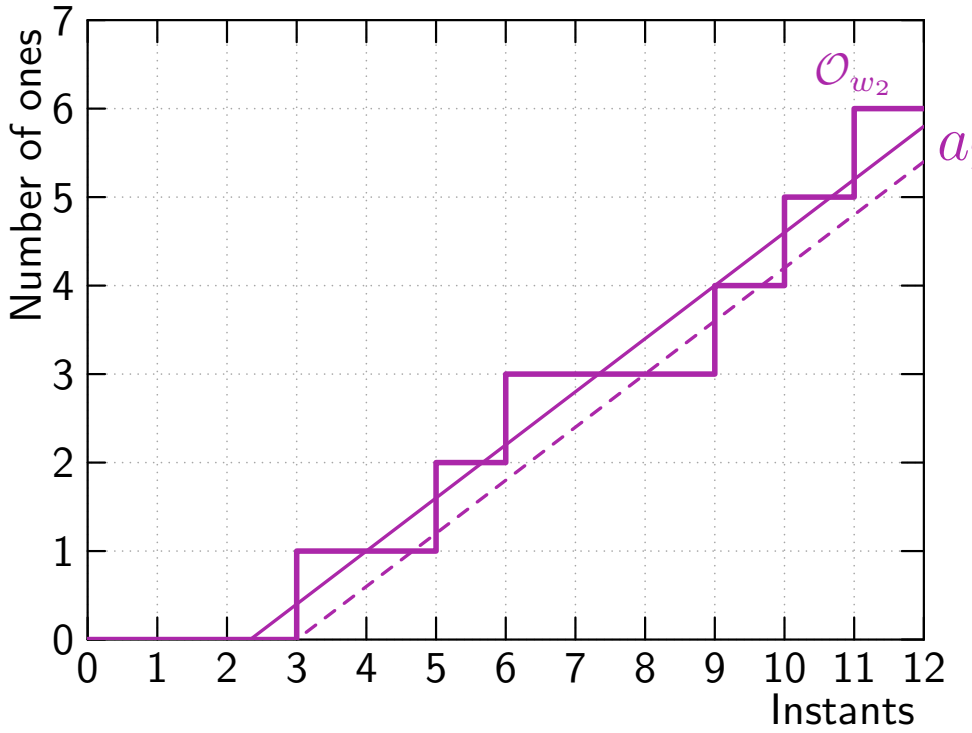
Que peut-on dire des systèmes non périodiques ?

- La même idée : du synchrone avec des spécification de propriétés entre horloges. Garanties d'absence de deadlock et buffer borné.
- Le calcul **exact** avec les horloges périodiques ne marche pas en pratique (et est inutile). E.g., (10100100) on $0^{3600}(1)$ on $(101001001) = 0^{9600}(10^4 10^7 10^7 10^2)$
- Motivations :

  1. traiter le cas de motifs longs. Eviter le calcul exact.
  2. Comment parler de d'horloges a peu près periodiques ? Par exemple α on w où $w = 00.(\ (10) + (01)\ )^*$
     (e.g. $w = 00\ 01\ 10\ 01\ 01\ 10\ 01\ 10 \ldots$)

Idée : manupiler des ensembles d'horloges. Se ramener à un problème arithmétique.

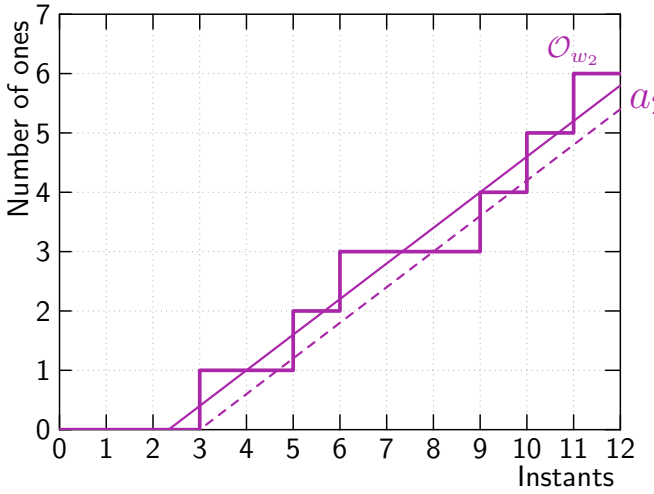
Abstraction de mots binaires infinis : (b^0, b^1, r)

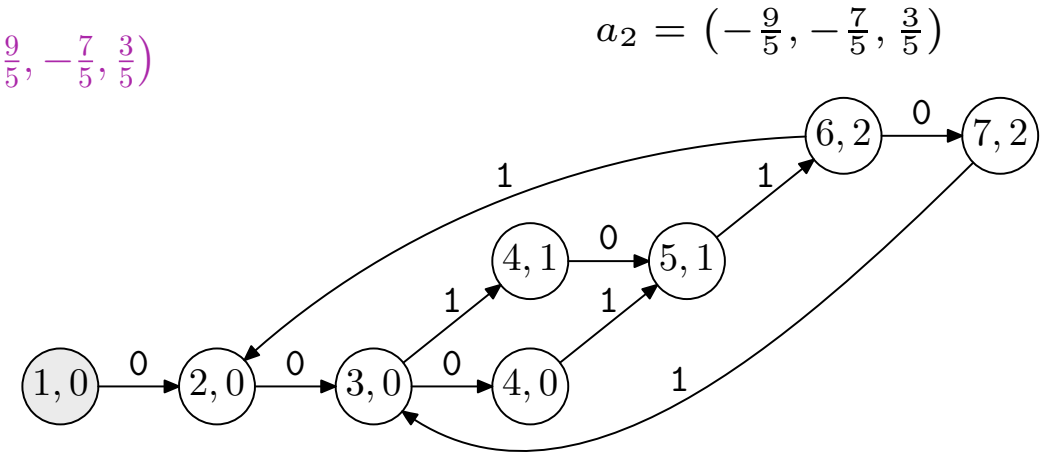


$a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$

$$concr\left(\left(b^0, b^1, r\right)\right) \overset{def}{\Leftrightarrow} \left\{ w,\ \forall i \geq 1,\ \wedge \begin{array}{lcl} w[i] = 1 & \Rightarrow & \mathcal{O}_w(i-1) < r \times i + b^1 \\ w[i] = 0 & \Rightarrow & \mathcal{O}_w(i-1) \geq r \times i + b^0 \end{array} \right\}$$

Horloges abstraites et automates





- Ensemble d'états $\{(i,j) \in \mathbb{N}^2\}$: coordonnées dans le chronogramme
- Nombre fini de classes d'équivalences
- Fonction de transition δ :

$$\delta(1,(i,j)) = nf(i+1,j+1) \quad \text{si } j < r \times i + b^1$$
$$\delta(0,(i,j)) = nf(i+1,j) \quad \text{si } j \geq r \times i + b^0$$

- Permet de vérifier/générer des horloges

Génération d'horloges

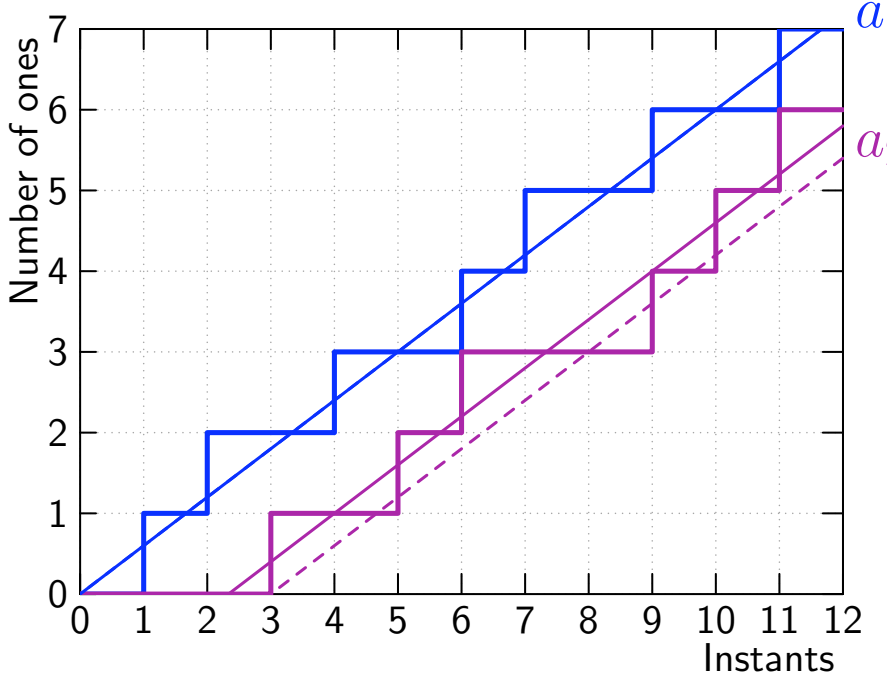
```
type rat = { num: int; den: int; }

let norm { num = n; den = l; } i j =
  if i >= l && j >= n then (i - l, j - n) else (i, j)

let node generate choice (b0, b1, r) = clk where
  rec i, j = (1,0) fby norm r (i+1) (if clk then j + 1 else j)
  and one = (rat_of_int j) <: (r *: (rat_of_int i) +: b1)
  and zero = (rat_of_int j) >=: (r *: (rat_of_int i) +: b0)
  and clk = choice one zero

let node early a = generate (fun x y -> x) a
let node late a = generate (fun x y -> not y) a
```

Relations abstraites



$a_1 = \left(0, 0, \frac{3}{5}\right)$

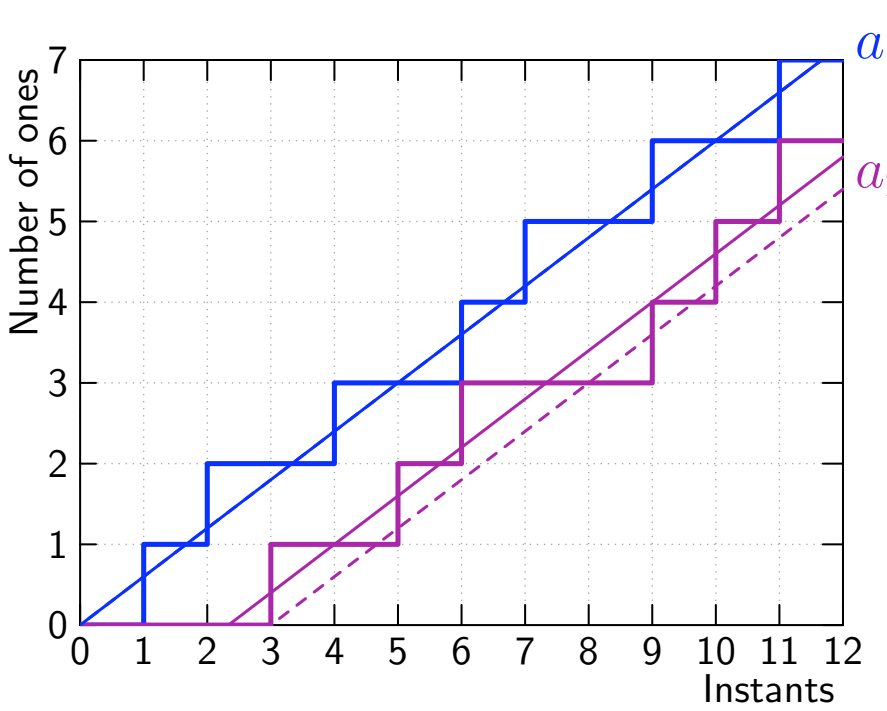
$a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$

Synchronisabilité :

$$\left(b^0{}_1, b^1{}_1, r_1\right) \bowtie^{\sim} \left(b^0{}_2, b^1{}_2, r_2\right) \Leftrightarrow r_1 = r_2$$

- proposition : $abs(c_1) \bowtie^{\sim} abs(c_2) \Leftrightarrow c_1 \bowtie c_2$

Relations abstraites



$a_1 = \left(0, 0, \frac{3}{5}\right)$

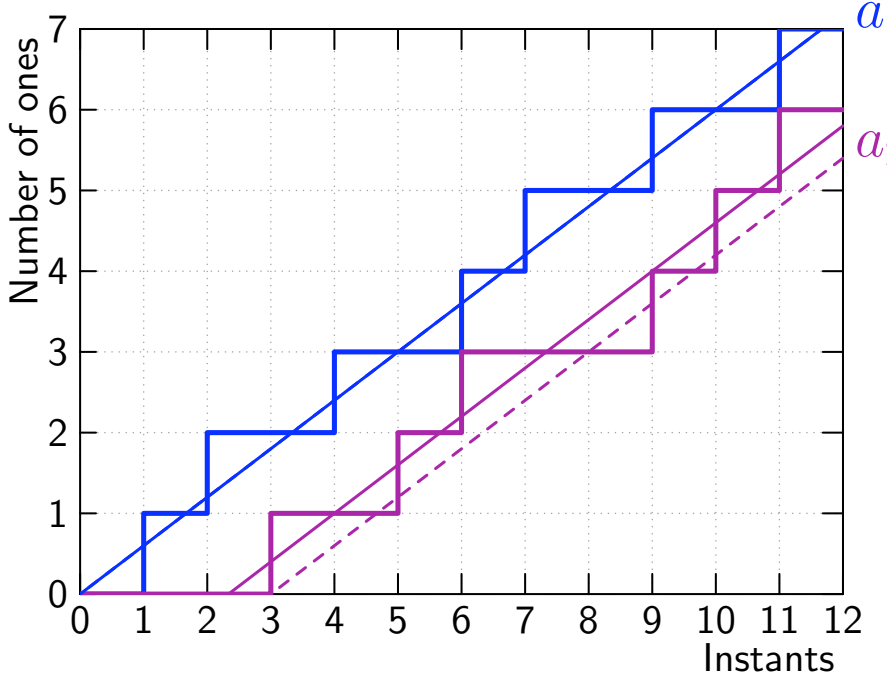
$a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$

Précédence :

$$\left(b^0{}_1, b^1{}_1, r_1\right) \preceq^{\sim} \left(b^0{}_2, b^1{}_2, r_2\right) \Rightarrow b^0{}_1 \geq b^1{}_2$$

- proposition : $abs(c_1) \preceq^{\sim} abs(c_2) \Rightarrow c_1 \preceq c_2$

Relations abstraites



$a_1 = \left(0, 0, \frac{3}{5}\right)$

$a_2 = \left(-\frac{9}{5}, -\frac{7}{5}, \frac{3}{5}\right)$

Sous-typage :

$$a_1 <:^{\sim} a_2 \Leftrightarrow a_1 \bowtie^{\sim} a_2 \wedge a_1 \preceq^{\sim} a_2$$

Buffer :

$$size(a_1, a_2) = \lceil {b^1}_1 - {b^0}_2 \rceil$$

Opérateurs abstraits

Horloges composées : $c ::= w \mid \mathit{not}\ w \mid c\ \mathit{on}\ c$

- Abstraction d'une horloge composée :

$$\begin{aligned} abs(\mathit{not}\ w) &= \mathit{not}^{\sim}\ abs(w) \\ abs(c_1\ \mathit{on}\ c_2) &= abs(c_1)\ \mathit{on}^{\sim}\ abs(c_2) \end{aligned}$$

- Propriété de correction :

$$\begin{aligned} \mathit{not}\ w &\in concr(\mathit{not}^{\sim}\ abs(w)) \\ c_1\ \mathit{on}\ c_2 &\in concr(abs(c_1)\ \mathit{on}^{\sim}\ abs(c_2)) \end{aligned}$$

- Définition de l'opérateur $\mathit{on}^{\sim}$:

$$\left(b^0{}_1, b^1{}_1, r_1\right)\ \mathit{on}^{\sim}\ \left(b^0{}_2, b^1{}_2, r_2\right) = \left(b^0{}_{12}, b^1{}_{12}, r_1 \times r_2\right)$$

$$\text{avec} \quad b^0{}_{12} = b^0{}_1 \times r_2 + b^0{}_2 \quad \text{et} \quad b^1{}_{12} = b^1{}_1 \times r_2 + b^1{}_2 + \delta$$
$$\text{et} \quad b^1{}_1 \leq 0 \quad \text{et} \quad b^1{}_2 \leq 0$$

Un peu de Coq

Correction de l'opérateur "on"

```
Property on_absj_correctness:
  forall (w1:ibw) (w2:ibw),
  forall (a1:abstractionj) (a2:abstractionj),
  forall H_wf_a1: well_formed_abstractionj a1,
  forall H_wf_a2: well_formed_abstractionj a2,
  forall H_a1_eq_absj_w1: in_abstractionj w1 a1,
  forall H_a2_eq_absj_w2: in_abstractionj w2 a2,
  forall H_wf_b1_1: Qden (absj_b1 a1) = Qden (absj_r a1),
  forall H_wf_b1_2: Qden (absj_b1 a2) = Qden (absj_r a2),
  in_abstractionj (on w1 w2) (on_absj a1 a2).
```

Formalisation et preuve des propriétés prouvées en Coq (5000 LOC) : toutes celles de l'abstraction proposée dans [APLAS 08] ; Un peu moins pour celle présentée ici.

Applications

Abstraction d'horloges périodiques avec de longs motifs :

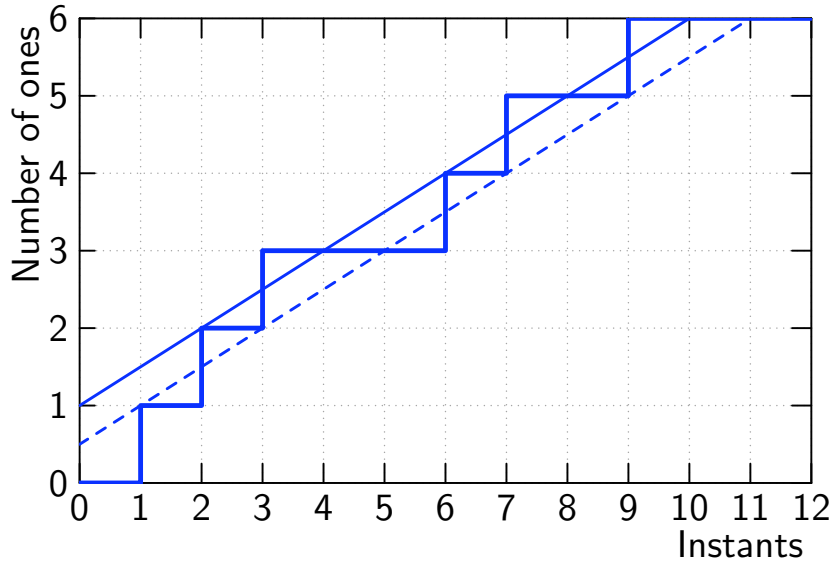
- exemple du Downscaler

  $abs((10100100)\ \mathit{on}\ 0^{3600}(1)\ \mathit{on}\ (1^{720}0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720}))$

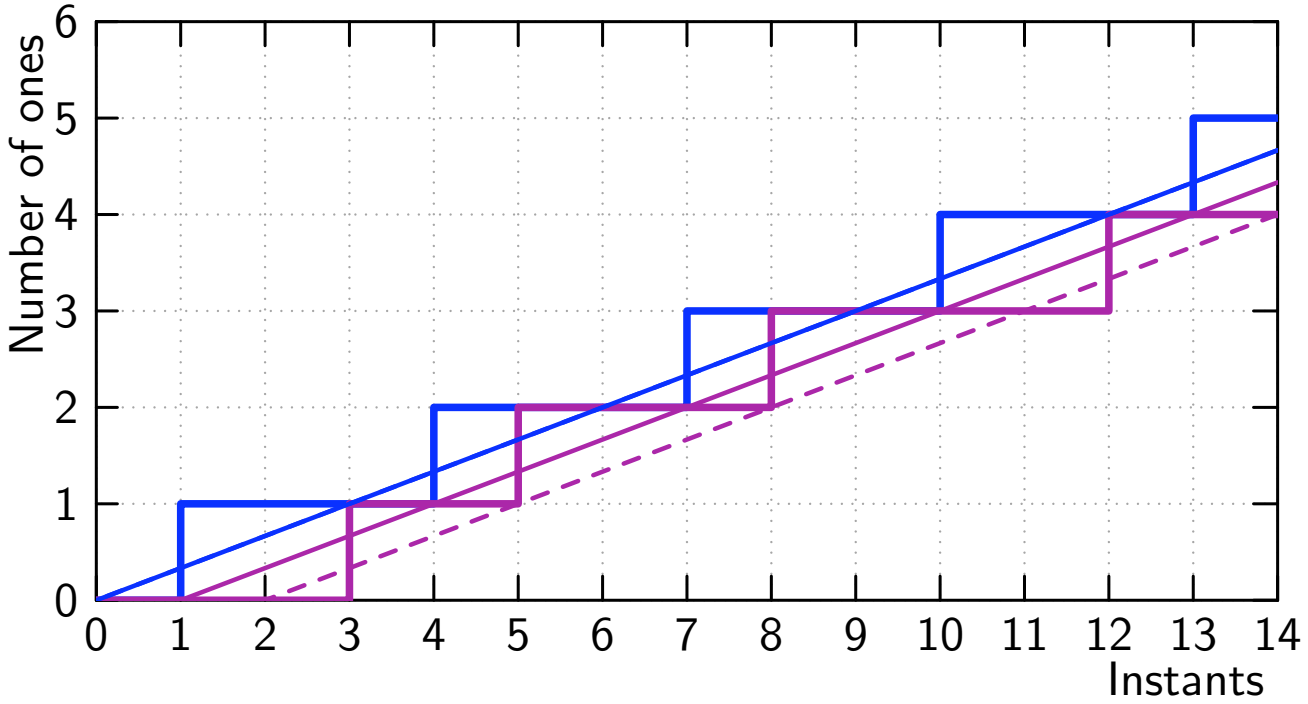
  $= \left(0, 0, \frac{3}{8}\right)\ \mathit{on}\sim\ (-3600, -3600, 1)\ \mathit{on}\sim\ \left(-400, \frac{4312}{9}, \frac{4}{9}\right) = \left(-2000, -1120, \frac{1}{6}\right)$

Modélisation de la gigue :

- exemple : $w \in 11.(\ (10) + (01)\ )^{\omega}$ peut être spécifiée par $\left(\frac{1}{2}, 1, \frac{1}{2}\right)$



Applications



Modélisation du temps d'exécution :

- f doit être exécutée tous les trois cycles et son exécution prend entre deux et trois cycles
- $f :: \forall \alpha.\alpha \ \mathtt{on}^{\sim} \left(0, 0, \frac{1}{3}\right) \rightarrow \alpha \ \mathtt{on}^{\sim} \left(-\frac{2}{3}, -\frac{1}{3}, \frac{1}{3}\right)$

Conclusion

- Beaucoup d'abstraction considérées, peu ont de bonnes propriétés vis-à-vis de l'opération `on`
- Beaucoup des idées présentées ici ont été étudiées dans le **Network Calculus** et le **Real-Time Calculus**
- Ces deux théories permettent de définir des enveloppes plus fines ; l'application à notre cas (e.g., `on`) n'est pas claire.
- On veut mélanger du synchrone "classique" et "souple" au sein d'un même programme. Modéliser de la gîgue sur une horloge quelconque, e.g.,

$$(-5, 0, c) = \{w/c <: w <: 0^5.c\}$$

- Un grand nombre de propositions ont été prouvées en Coq (toutes celles d'[APLAS 08], quelques unes à faire pour la présente)
- implémentation en cours