Typage des horloges périodiques en Lucy-n

Louis Mandel & Florence Plateau

LRI, Université Paris-Sud 11
LIENS, École Normale Supérieure
INRIA
{mandel,plateau}@lri.fr

Résumé

Lucy-n est un langage permettant de programmer des réseaux de processus communiquant à travers des *buffers* de taille bornée. La taille des buffers et les rythmes d'exécution relatifs des processus sont calculés par une phase de typage appelée calcul d'horloge. Ce typage nécessite la résolution d'un ensemble de contraintes de sous-typage. L'an dernier, nous avons proposé un algorithme de résolution de ces contraintes utilisant des méthodes issues de l'interprétation abstraite. Cette année nous présentons un algorithme tirant profit de toute l'information contenue dans les types.

1. Introduction

Le modèle n-synchrone est un modèle de programmation flot de données. Il décrit des réseaux de processus qui s'exécutent de manière concurrente et communiquent à travers des buffers de taille bornée. Il allie concurrence, déterminisme et souplesse dans les communications. Ces qualités sont utiles en particulier pour la programmation d'applications multimédia [6].

Nous avons présenté l'an dernier le langage Lucy-n [10, 11], une extension n-synchrone de Lustre [4]. Lucy-n fournit une analyse statique qui infère les rythmes d'exécution des nœuds de calcul et la taille des buffers. Pour faire cela, il hérite de toute la tradition du *calcul d'horloge* des langages synchrones flot de données. Ce calcul est un système de type dédié qui vérifie qu'un réseau de processus peut être exécuté sans buffers [5]. Dans les langages synchrones, chaque flot est associé à une horloge qui définit les instants où une donnée est présente. Les horloges sont des mots binaires infinis où l'occurrence d'un `1` indique la présence d'une valeur sur le flot et l'occurrence d'un `0` indique l'absence de valeur. Voici un exemple de flot `x` et son horloge :

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

Le calcul d'horloge impose à chaque expression de vérifier une contrainte de type comme celle-ci :

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

Cette règle établi que dans l'environnement de typage H, l'expression $e_1 + e_2$ a une horloge du type ck_3 si e_1 a une horloge du type ck_1, e_2 une horloge du type ck_2 et si la contrainte $ck_1 = ck_2 = ck_3$ est vérifiée.[1] L'égalité des types garantit l'égalité des horloges des flots que l'on compose et donc que l'exécution ne nécessitera pas de buffers pour stocker les données en attente d'être traitées. Le calcul

[1] Les ensembles C_1 et C_2 contiennent les contraintes collectées durant le typage des expressions e_1 et e_2.

d'horloge des langages synchrones ne collecte donc que des contraintes d'égalité de type pour garantir une exécution sans buffers.

L'adaptation du calcul d'horloge pour le modèle n-synchrone nécessite l'introduction d'une règle de sous-typage permettant de typer la primitive de bufferisation. Si un flot dont l'horloge est de type ck peut être utilisé plus tard sur une horloge de type ck' en étant stocké dans un buffer de taille bornée, on dira que ck est un sous-type de ck', noté $ck \ \texttt{<:}\ ck'$:

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

Le calcul d'horloge de Lucy-n collecte donc à la fois des contraintes d'égalité et de sous-typage.

Pour résoudre ces contraintes, il faut être capable d'unifier des types ($ck_1 \ \texttt{=}\ ck_2$) et de faire en sorte qu'ils vérifient la relation de sous-typage ($ck_1 \ \texttt{<:}\ ck_2$). Un cas intéressant est celui où les horloges sont des mots binaires ultimement périodiques, c'est-à-dire composés d'un préfixe suivi de la répétition infinie d'un même motif. Dans ce cas, l'égalité et le sous-typage sont des propriétés décidables. L'an dernier, nous avons présenté un algorithme de résolution des contraintes sur les types d'horloges ultimement périodiques [10]. Cet algorithme utilisait une méthode d'abstraction des horloges [9]. L'abstraction des horloges permet de manipuler une spécification simple des instants de présence plutôt que les horloges elles mêmes, en oubliant des informations sur leur forme précise. Ceci a permis de mettre au point un algorithme de résolution des contraintes simple et efficace.

Cependant, la perte d'information engendrée par l'abstraction peut mener à échouer dans la résolution d'un système de contraintes ayant une solution. D'autre part, elle entraîne une sur-approximation de la taille des buffers nécessaires. Dans le cas où les horloges sont simples, on peut vouloir privilégier la qualité des résultats plutôt que la rapidité à les calculer.

On s'intéresse dans cet article à la mise en place d'un algorithme de résolution des contraintes sans abstraction des horloges. Cette problématique est une question difficile pour deux raisons. D'une part, elle dépend intrinsèquement de toutes les informations contenues dans les horloges. Si les préfixes et motifs périodiques des mots décrivant les horloges sont de grande taille, on aboutit rapidement à une explosion combinatoire. D'autre part, la gestion des comportements initiaux (décris par les préfixes des mots) est toujours délicate [2, 14] et souvent mise de coté [1]. Le traitement conjoint des comportements initiaux et périodiques est source de complexité et à notre connaissance, il n'existe pas de travaux réussissant à traiter ces deux phases de manière séparée.

Nous présentons le langage Lucy-n dans la section 2, puis son calcul d'horloge dans la section 3. La section 4 introduit les propriétés des mots périodiques utilisées dans la section 5 qui présente l'algorithme de résolution des contraintes. La section 6 discute des résultats obtenus sur deux exemples. La section 7 conclut cet article en présentant des perspectives de recherche.

2. Le langage

Dans cette section, nous présentons le langage Lucy-n en programmant un composant d'un encodeur GSM. Ce composant est un encodeur cyclique qui prend en entrée un flot de bits (représentant des échantillons de voix) et injecte dans ce flot 3 bits de redondance tous les 50 bits.

Le principe de l'encodeur cyclique est de considérer les 50 bits à encoder comme la suite des coefficients d'un polynôme p de degré 49 à coefficients sur le corps de Galois à deux éléments. Les bits de redondance sont les coefficients du reste de la division de p par un autre polynôme propre à l'encodeur, ici $X^3 + X + 1$. Le circuit classique [13] pour effectuer cette division est montré sur la figure 1. Dans cette figure, les opérateurs $\oplus$ représentent le ou exclusif et les carrés représentent des registres initialisés à `false`.

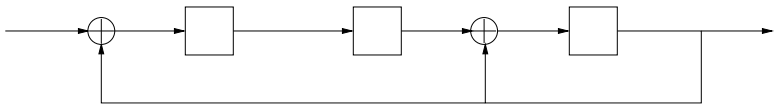
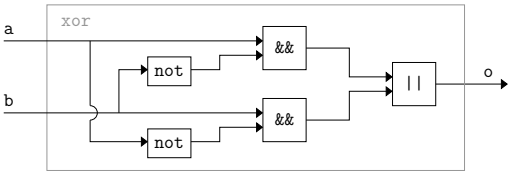


FIG. 1 – Circuit effectuant la division par $X^3 + X + 1$. Le flux d'entrée est la suite des cinquante coefficients du polynôme à diviser. Après la consommation du cinquantième bit d'entrée, tous les coefficients du polynôme quotient ont été produits en sortie et les registres contiennent les coefficients du polynôme reste.

L'opérateur ou exclusif se programme de la façon suivante en Lucy-n : [2]

```
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
```

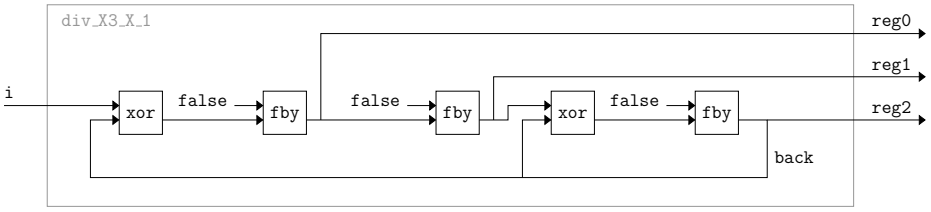


Le nœud `xor` prend en entrée deux flots `a` et `b` et calcule la valeur du flot de sortie `o`. La valeur de `o` est définie par l'équation `o = (a && (not b)) || (b && (not a))` où les opérateurs scalaires `&&`, `||` et `not` sont appliqués point à point aux flots d'entrée. Ainsi, si l'on applique le nœud `xor` sur les flots `x = true false true false false`... et `y = false false true true false`..., on obtient le flot `true false false true false`...

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

La définition du nœud `xor` est suivie par deux informations qui sont inférées automatiquement par le typeur de Lucy-n : le type de données (*`val xor : (bool * bool) -> bool`*) et le type d'horloges (*`val xor :: forall 'a. ('a * 'a) -> 'a`*). Le type d'horloges indique ici que les deux flots d'entrée doivent arriver au même rythme *`'a`* et que le flot de sortie sera produit sur ce rythme *`'a`*. Le caractère polymorphe du type permet d'appliquer le nœud `xor` sur tout couple de flots de même horloge, quelle que soit cette horloge.

Comme le langage Lucy-n fournit une primitive de registre initialisé appelée `fby`, nous sommes donc maintenant en mesure de programmer le circuit de la figure 1.




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

On voit ici que les flots `reg0`, `reg1` et `reg2` sont définis de façon mutuellement récursive. Par exemple, l'équation `reg2 = false fby reg1` signifie que `reg2` vaut `false` au premier instant, et vaut la valeur précédente de `reg1` aux instants suivants.

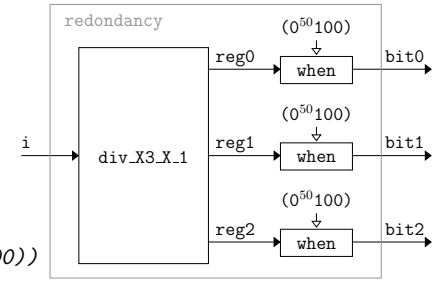
[2] Nous montrons à droite la représentation du programme sous forme de schéma flot de données.

Afin de pouvoir utiliser le circuit de la figure 1 sur plus d'un polynôme, il faut réinitialiser les trois registres à `false` après l'arrivée des coefficients de chaque polynôme à diviser (c'est à dire tous les 50 bits d'entrée ici). Le contenu des registres est le résultat d'un ou exclusif entre l'arc de rétroaction `back` et le registre précédent (ou le flot d'entrée pour le premier registre). Pour les réinitialiser à `false`, on doit donc injecter trois `false` en entrée et trois `false` à la place des valeurs de rétroaction, et cela tous les 50 bits d'entrée.

Le type d'horloges du nœud `div_X3_X_1` ainsi modifié[3] est le suivant :
`val div_X3_X_1 :: forall 'a. 'a on (1^50 0^3) -> ('a * 'a * 'a).`
La notation `(1^50 0^3)` représente la répétition infinie du mot binaire $1^{50}0^3$ avec 1^{50} la concaténation de cinquante `1` et 0^3 la concaténation de trois `0`. Le type d'horloges du flot d'entrée est `'a on (1^50 0^3)`. Cela signifie que par rapport à un rythme de référence `'a` définissant les instants, le flot d'entrée doit être présent aux 50 premiers instants puis absent pendant 3 instants (le diviseur a ainsi le temps de réinitialiser ses trois registres). On traite donc une division tous les 53 instants du rythme `'a`. La valeur des registres est quand à elle produite en sortie à chaque instant de ce rythme de référence. Dans la suite de cette section, il sera implicite que tous les comportements décrits se répètent de manière périodique.

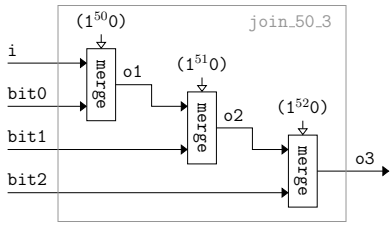
Le reste de la division étant contenu dans les registres après l'arrivée du $50^{ième}$ bit d'entrée, le nœud qui calcule les bits de redondance doit échantillonner la sortie de `div_X3_X_1` pour ne conserver que les valeurs des registres au $51^{ième}$ instant. Pour cela, on dispose de l'opérateur `when`. Celui-ci prend en paramètre un flot et une horloge, et ne conserve les valeurs du flot que quand l'horloge vaut `1`. Pour ne conserver que la $51^{ième}$ valeur parmi des suites de 53 bits, on utilise l'horloge d'échantillonage (0^{50}`100`) :

```
let node redondancy 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 redondancy : bool -> (bool * bool * bool)
val redondancy ::
  forall 'a. 'a on (1^50 0^3) ->
     ('a on (0^50 100) * 'a on (0^50 100) * 'a on (0^50 100))
```



Pour ajouter les 3 bits de redondance à la suite du flot à coder, on utilise l'opérateur `merge` servant à fusionner des flots. Celui-ci prend une horloge et deux flots en paramètre. Lorsque l'horloge vaut `1` c'est le premier flot qui est transmis en sortie et le second doit être absent. L'inverse se produit lorsque l'horloge vaut `0`. Pour intégrer le premier bit de redondance (`bit0`) après les 50 bits de données, on utilise donc l'horloge de fusion (1^{50}`0`). On obtient un flot de 51 bits. On utilise alors l'horloge (1^{51}`0`) pour intégrer le deuxième bit de redondance, et enfin l'horloge (1^{52}`0`) pour le troisième.

```
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
```



Nous verrons dans la section 4 que le type d'horloges de `join_50_3` est équivalent à :

$$\texttt{join\_50\_3} : \forall\alpha.\,(\alpha\ \texttt{on}\ (1^{50}000) * \alpha\ \texttt{on}\ (0^{50}100) * \alpha\ \texttt{on}\ (0^{50}010) * \alpha\ \texttt{on}\ (0^{50}001)) \rightarrow \alpha$$

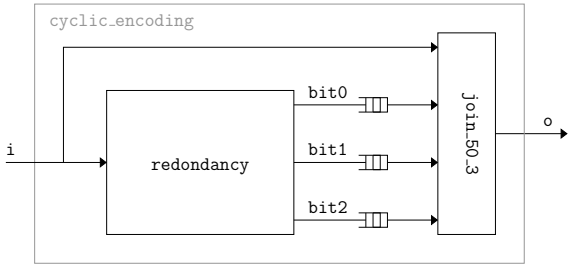
Ce type exprime que le flot de données doit arriver pendant 50 instants puis être absent

[3] disponible à l'adresse `http://www.lri.fr/~plateau/jfla11/gsm.ls.html`

pendant 3 instants, que le flot contenant le premier bit de redondance doit arriver au $51^{ième}$ instant, celui contenant le deuxième bit de redondance au $52^{ième}$ instant et celui contenant le troisième bit au $53^{ième}$ instant.

Pour coder le nœud d'encodage cyclique, il faut maintenant utiliser le nœud `redondancy` pour calculer les 3 bits de redondance et le nœud `join_50_3` pour les intégrer au flot d'entrée. Mais les bits de redondance sont produits trop tôt (à l'instant 51) par rapport au moment où ils sont attendus en entrée du nœud `join_50_3` (respectivement aux instants 51, 52 et 53). Il faut donc stocker ces données, à l'aide de l'opérateur `buffer` :

```
39 let node cyclic_encoding i = o where
40   rec (bit0, bit1, bit2) = redondancy 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
```

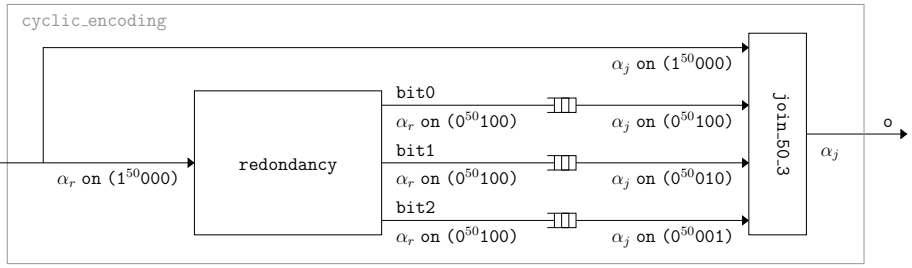


Le typeur calcule automatiquement la taille des buffers nécessaires. On voit que le buffer stockant le premier bit de redondance est inutile, la taille inférée est 0. En effet ce bit est produit au $51^{ième}$ instant, instant auquel il est utilisé par `join_50_3` pour être intégré au flot de sortie. Les deux autres bits de redondance sont eux aussi produits au $51^{ième}$ instant, mais ils sont utilisés plus tard. Il faut donc un buffer de taille 1 pour stocker le deuxième bit durant 1 instant et un buffer de taille 1 pour stocker le troisième bit durant 2 instants, en attendant leur utilisation.

Avant de calculer les tailles de buffer nécessaires, le typeur doit inférer les rythmes d'exécution des différents nœuds de manière à ce que les données soient produites à l'instant où elles sont consommées quand il n'y a pas de buffers, et de manière à ce qu'il n'y ait pas de lecture dans un buffer vide et pas d'accumulation infinie de données là où il y a des buffers.

3. Calcul d'horloge

Nous avons vu dans la section 1 que chaque expression d'un programme doit satisfaire une contrainte de type.[4] Considérons par exemple le nœud `cyclic_encoding` que nous avons programmé dans la section précédente.



Si l'on nomme α_i le type d'horloges de l'entrée `i`, l'expression `redondancy i` génère la contrainte $\alpha_i = \alpha_r$ `on` $(1^{50}0^3)$. En effet, une fois instancié avec une variable fraîche α_r, le type d'horloges du nœud `redondancy` est α_r `on` $(1^{50}0^3) \rightarrow (\alpha_r$ `on` $(0^{50}100) \times \alpha_r$ `on` $(0^{50}100) \times \alpha_r$ `on` $(0^{50}100))$, donc le

[4] Les règles du calcul d'horloge sont détaillées dans [14].

type de son entrée doit être égal à α_r `on` $(1^{50}0^3)$. L'équation `(bit0, bit1, bit2) = redondancy i` ajoute alors à l'environnement que `bit0`, `bit1` et `bit2` sont de type α_r `on` $(0^{50}100)$.

De même, l'application de `join_50_3` ajoute des contraintes sur les types de ses entrées. Une fois instancié avec une variable fraîche α_j, le type d'horloges du nœud `join_50_3` est $(\alpha_j$ `on` $(1^{50}0^3) \times \alpha_j$ `on` $(0^{50}100) \times \alpha_j$ `on` $(0^{50}010) \times \alpha_j$ `on` $(0^{50}001)) \rightarrow \alpha_j$. Ce type impose la contrainte que le type de la première entrée (ici α_i) soit égal à α_j `on` $(1^{50}0^3)$ et les types des autres entrées (ici α_r `on` $(0^{50}100)$) soient respectivement sous-types de α_j `on` $(0^{50}100)$, α_j `on` $(0^{50}010)$ et α_j `on` $(0^{50}001)$. Pour ces dernières entrées, on n'impose pas l'égalité mais seulement le sous-typage car celles-ci sont consommées à travers des buffers. Pour finir, l'équation `o = join_50_3 (...)` ajoute à l'environnement l'information que l'horloge de `o` est du type α_j, type de retour de `join_50_3`.

Le nœud `cyclic_encoding` a donc le type d'horloges $\alpha_i \rightarrow \alpha_j$, avec les contraintes suivantes :

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

Pour finir le typage de ce nœud et être capable de calculer la taille des buffers, il faut trouver une solution à ce système de contraintes. On cherche donc des instanciations pour les variables de type α_i, α_r et α_j telles que les contraintes soient toujours satisfaites. Ces instanciations doivent être des types d'horloge Lucy-n, c'est à dire de la forme : $ck \quad ::= \quad \alpha \mid ck$ `on` p avec p un mot binaire ultimement périodique.

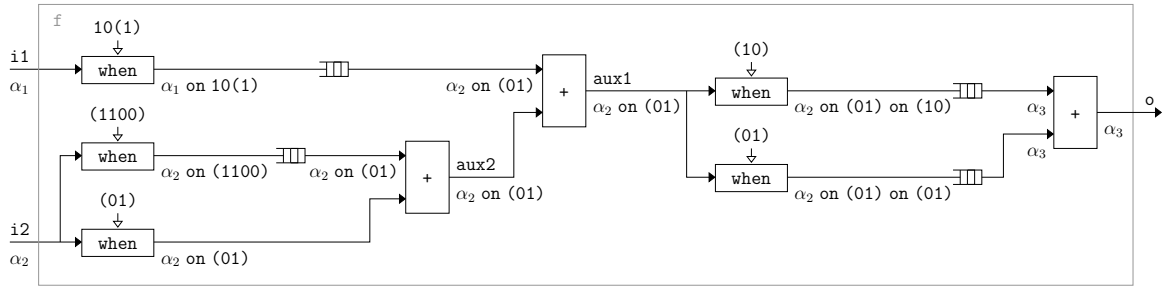
Ici, pour résoudre le système de contraintes, on peut commencer par traiter les contraintes d'égalité en choisissant la substitution suivante : $\theta = \{\alpha_i \leftarrow \alpha$ `on` $(1^{50}0^3)$; $\quad \alpha_r \leftarrow \alpha$; $\quad \alpha_j \leftarrow \alpha$; $\}$. En effet, une fois cette substitution appliquée au système A, on obtient :

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

Remarque 1. *Il est à noter qu'il n'existe pas d'algorithme d'unification glouton qui soit complet. En effet, il n'y a pas d'unificateur le plus général sur les types d'horloges : le choix de l'unificateur de deux types dont les variables sont α_1 et α_2 dépend de toutes les contraintes existant sur ces variables. Par conséquent, pour que l'algorithme soit complet, il faut résoudre toutes les contraintes de manière globale. La relation de sous-typage étant antisymétrique, une façon simple de faire est de traiter chaque contrainte d'égalité comme deux contraintes de sous-typage ($ck_1 = ck_2 \Leftrightarrow (ck_1 <: ck_2) \wedge (ck_2 <: ck_1)$). Dans l'exemple que nous traitons, l'unification gloutonne structurelle menant à une solution, nous l'avons utilisée pour plus de concision. Pour plus de précisions à ce propos, se référer à la section 5.2 de [14].*

Nous avons donc ramené notre système de contraintes à un système contenant uniquement des contraintes de sous-typage. On peut constater que dans ce système, toutes les contraintes sont exprimées en fonction de la même variable de type. Comme nous l'avons montré dans [14, 10], il est alors possible de simplifier les opérateurs `on` par la gauche.

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




```
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. 2 – Le nœud `f` et son schéma flot de données. Ce dernier est annoté avec les types obtenus après résolution des contraintes d'égalité, afin de ne montrer que les contraintes de sous-typage.

Il existe des cas où les contraintes de sous-typage ne sont pas exprimées en fonction de la même variable de type. Par exemple, le programme de la figure 2 génère l'ensemble de contraintes de sous-typage suivant, où seule la deuxième contrainte se simplifie :

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

On peut toujours se ramener au cas où toutes les contraintes sont exprimées en fonction de la même variable de type. Pour cela, on introduit des variables de mot notées c_n et on substitue chaque variable α_n par α `on` c_n. Ici, en appliquant la substitution $\theta = \{\alpha_1 \leftarrow \alpha \texttt{ on } c_1;\ \alpha_2 \leftarrow \alpha \texttt{ on } c_2;\ \alpha_3 \leftarrow \alpha \texttt{ on } c_3;\}$, on obtient le système suivant :

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

Les systèmes de contraintes ainsi obtenus ne sont plus des systèmes de contraintes sur les types, mais des systèmes de contraintes sur les mots binaires ultimement périodiques. L'opérateur ***on*** et la relation <: sur les mots binaires, que nous appelons relation d'*adaptabilité*, sont définis dans la section suivante. L'algorithme qui infère les mots binaires ultimement périodiques c_n qui satisfont les contraintes d'adaptabilité est quant à lui présenté dans la section 5.

4. Algèbre des mots périodiques

Les horloges étant représentées par des mots binaires infinis ultimement périodiques, nous avons besoin d'établir dans cette section des définitions et propriétés sur ces derniers. Les preuves de ces propriétés sont disponibles dans le chapitre 4 de [14]. Toutes les propriétés et remarques énoncées ici sont utilisées dans la section 5.

4.1. Mots binaires ultimement périodiques

Nous notons w les mots binaires infinis ($w ::= \texttt{0}w \mid \texttt{1}w$), u ou v les mots binaires finis ($u, v ::= \texttt{0}u \mid \texttt{1}u \mid \varepsilon$), $|u|$ la taille de u et $|u|_{\texttt{1}}$ le nombre de `1` qu'il contient. Notre analyse s'intéresse aux instants de présence des données sur les flots. Elle manipule donc principalement les indices des `1` dans les mots :

Définition 1 (indice du $j^{ième}$ `1` de w : $\mathcal{I}_w(j)$)**.**
Soit w un mot contenant une infinité de `1`*.*

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

Par exemple l'indice du $3^{ième}$ `1` de $w_1 = \texttt{11010}\,\texttt{11010}\ldots$ est 4.

Remarque 2 (indices croissants)**.**
La fonction $\mathcal{I}_w$ est strictement croissante ($\forall j \geq 1,\ \mathcal{I}_w(j) < \mathcal{I}_w(j+1)$).

Remarque 3 (indices suffisants)**.**
Il en découle que l'indice du $j^{ième}$ `1` *est supérieur ou égal à j ($\forall j \geq 1,\ \mathcal{I}_w(j) \geq j$).*

Comme annoncé dans la section 1, nous nous intéressons ici aux horloges ultimement périodiques $u\texttt{(}v\texttt{)}$, c'est à dire composées d'un mot fini u en préfixe, suivi de la répétition infinie d'un même mot fini v :

Définition 2 (mot binaire ultimement périodique p)**.** $p = u\texttt{(}v\texttt{)} \stackrel{def}{\Leftrightarrow} p = uw$ avec $w = vw$

Par exemple, $p = \texttt{1101(110)} = \texttt{1101 110 110 110}\ \ldots$ Nous noterons $p.u$ le préfixe de p (ici `1101`) et $p.v$ son motif périodique (ici `110`).

Le taux de p est le ratio entre le nombre de `1` et la taille de son motif périodique :

Définition 3 (taux de p)**.** $rate(p) = \dfrac{|p.v|_{\texttt{1}}}{|p.v|}$

Dans la suite, nous ne considérons que des mots de taux non nul, c'est-à-dire dont le motif périodique contient au moins un `1` (ces mots contiennent une infinité de `1` et sont les horloges de flots présents infiniment souvent).

Les deux remarques suivantes garantissent la bonne formation d'un mot périodique.

Remarque 4 (périodicité)**.** *Deux occurrences successives d'un même* `1` *du motif périodique sont séparées d'une distance égale à la taille de ce motif ($\forall j > |p.u|_{\texttt{1}},\ \mathcal{I}_p(j + |p.v|_{\texttt{1}}) = \mathcal{I}_p(j) + |p.v|$).*

Remarque 5 (taille suffisante)**.** *La taille d'un mot fini v est supérieure ou égale au nombre d'éléments compris entre les indices du premier et du dernier* `1` *de v. C'est en particulier le cas pour la taille du motif périodique d'un mot infini p ($|p.v| \geq 1 + \mathcal{I}_p(|p.u|_{\texttt{1}} + |p.v|_{\texttt{1}}) - \mathcal{I}_p(|p.u|_{\texttt{1}} + 1)$).*

Un même mot ultimement périodique peut être représenté d'une infinité de façons différentes. Par exemple `(10)` = `(1010)` = `1(01)` = ... Il existe une forme normale qui est la représentation avec le préfixe et le motif périodique les plus courts possibles. Cependant, nous verrons dans la suite que pour certaines opérations, il est plus pratique de mettre les opérandes sous une forme plus longue que leurs formes normales respectives. On pourra par exemple ajuster les formes des opérandes pour qu'elles soient de même taille, ou qu'elles contiennent le même nombre de `1`, ou encore que le nombre de `1` de la première soit égal à la taille de la seconde.

Remarque 6. *Voici les manipulations possibles sur la forme d'un mot binaire ultimement périodique :*
- *Augmenter la taille du préfixe : par exemple, on peut rallonger le préfixe de* $p =$ `1101(110)` *de deux éléments. On obtient p sous la forme* `1101 11(0 11)`*. Augmenter la taille du préfixe permet d'augmenter son nombre de* `1`*.*
- *Répéter le motif périodique : par exemple on peut tripler la taille du motif périodique de* $p =$ `1101(110)` *(et donc tripler son nombre de* `1`*). On obtient p sous la forme* `1101(110 110 110)`*.*

4.2. Relation d'adaptabilité

Définissons maintenant la relation $<:$ (appelée *adaptabilité*). La relation $w_1 <: w_2$ est vérifiée si et seulement si un flot d'horloge w_1 peut-être être stocké dans un buffer de taille bornée puis consommé au rythme de l'horloge w_2. Pour cela, il faut qu'il n'y ait pas d'accumulation infinie de données en attente dans le buffer, ni de lectures dans le buffer lorsqu'il est vide. Le caractère borné du nombre de valeurs présentes dans le buffer au cours de l'exécution est garanti par la relation de *synchronisabilité* entre les mots w_1 et w_2 (notée $w_1 \bowtie w_2$), qui vérifie que les nombres asymptotiques de lectures et d'écritures sont égaux. L'absence de lectures dans un buffer vide est garanti par la relation de *précédence* entre les mots w_1 et w_2 (notée $w_1 \preceq w_2$), qui vérifie que la $j^{ième}$ écriture dans le buffer a toujours lieu avant la $j^{ième}$ lecture. La relation d'adaptabilité est la conjonction des relations de précédence et de synchronisabilité.

Deux mots w_1 et w_2 sont synchronisables si la différence entre le nombre d'occurrences de `1` dans w_1 et le nombre d'occurrences de `1` dans w_2 est bornée. Pour tester la relation de synchronisabilité sur des mots périodiques, il suffit de vérifier que leur motif périodique comporte la même proportion de `1`.

Proposition 1 (test de synchronisabilité)**.**

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

Un mot w_1 précède un mot w_2 si l'occurrence de son $j^{ième}$ `1` a toujours lieu avant ou en même temps que celle du $j^{ième}$ `1` de w_2. Pour tester la relation de précédence sur des mots périodiques synchronisables, il suffit de vérifier cette condition jusqu'à ce que le comportement périodique "commun" soit atteint.

Proposition 2 (test de précédence)**.**
Soient p_1 *et* p_2 *tels que* $p_1 \bowtie p_2$*. Soit* $h = \max(|p_1.u|_1, |p_2.u|_1) + \ ppcm\ (|p_1.v|_1, |p_2.v|_1)$.

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

On vérifie que la condition de précédence est satisfaite jusqu'au $h^{ième}$ `1`. Voici une explication du choix de cette borne h. Utilisons la remarque 6 pour ajuster les composantes respectives de p_1 et p_2 au même nombre de `1`. On obtient deux mots p'_1 et p'_2 (égaux à p_1 et p_2) dont le préfixe contient $\max(|p_1.u|_1, |p_2.u|_1)$ éléments `1` et dont le motif périodique contient ppcm $(|p_1.v|_1, |p_2.v|_1)$ éléments `1`. Après un parcours de $h = |p'_1.u|_1 + |p'_1.v|_1 = |p'_2.u|_1 + |p'_2.v|_1$ éléments `1` de p'_1 et p'_2, on recommence donc simultanément le parcours de leur motif périodique. Comme ces motifs ont le même taux, on se retrouve alors exactement dans la même situation qu'au début du premier parcours des motifs périodiques. Si la condition est vérifiée jusqu'au $h^{ième}$ élément `1`, elle l'est donc pour toujours.

La relation d'*adaptabilité* est la conjonction des relations de synchronisabilité et de précédence.

Proposition 3 (test d'adaptabilité)**.** $p_1 <: p_2 \ \Leftrightarrow\ p_1 \bowtie p_2 \ \wedge\ p_1 \preceq p_2$

4.3. Taille des buffers

Pour calculer la taille d'un buffer, on s'intéresse au nombre de valeurs qui sont écrites et lues dans celui-ci au cours de l'exécution. Pour cela, on va considérer les fonctions de cumul des horloges.

Ces fonctions comptent le nombre d'instants de présence de valeurs sur un flot depuis le début de l'exécution :

Définition 4 (fonction de cumul de w : $\mathcal{O}_w$)**.**[5]

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

Considérons un buffer qui prend en entrée un flot sur le rythme w_1, et fournit en sortie ce 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 ($\mathcal{O}_{w_2}(i)$). La taille de buffer nécessaire et suffisante est le nombre maximal de valeurs présentes dans le buffer durant l'exécution. Pour calculer la taille de buffer sur des mots périodiques, il suffit de calculer la valeur maximale de cette différence jusqu'à ce que le comportement périodique "commun" soit atteint.

Proposition 4 (taille du buffer)**.**
Soient p_1 et p_2 telles que $p_1 <: p_2$. Soit $H = \max(|p_1.u|, |p_2.u|) + \ ppcm\,(|p_1.v|, |p_2.v|)$.

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

4.4. Horloges échantillonnées

L'opérateur ***on*** sur les horloges donne le rythme d'un flot échantillonné. Il permet d'exprimer l'horloge de sortie de l'opérateur `when` qui laisse passer certaines valeurs selon une condition w_2 d'un flot consommé sur un rythme w_1 :

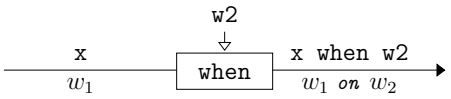


FIG. 3 – Si `x` est d'horloge w_1, `x when w2` est d'horloge w_1 ***on*** w_2.

Définition 5 (opérateur ***on***)**.**
$$\begin{aligned} \texttt{0}w_1\ \textbf{\textit{on}}\ w_2 &\stackrel{def}{=} \texttt{0}(w_1\ \textbf{\textit{on}}\ w_2) \\ \texttt{1}w_1\ \textbf{\textit{on}}\ \texttt{1}w_2 &\stackrel{def}{=} \texttt{1}(w_1\ \textbf{\textit{on}}\ w_2) \\ \texttt{1}w_1\ \textbf{\textit{on}}\ \texttt{0}w_2 &\stackrel{def}{=} \texttt{0}(w_1\ \textbf{\textit{on}}\ w_2) \end{aligned}$$

Par exemple, si $w_1 = \texttt{11010111}\ldots$ et $w_2 = \texttt{101100}\ldots$, alors w_1 ***on*** $w_2 = \texttt{10010100}\ldots$.
Considérons l'échantillonnage d'un flot `x` d'horloge w_1 par la condition w_2 :

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

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

On peut constater ici que si `x` est présent (l'élément de w_1 vaut `1`), alors on consulte le prochain élément de l'horloge d'échantillonnage (w_2). Si cet élément vaut `1` alors la valeur de `x` est conservée par l'échantillonnage et le flot `x when w2` est présent (w_1 ***on*** w_2 vaut `1`). Dans le cas contraire, la valeur de `x` est supprimée par l'échantillonnage et le flot `x when w2` est donc absent (w_1 ***on*** w_2 vaut `0`). Enfin, si `x` est absent (i.e. w_1 vaut `0`), le flot `x when w2` est absent (i.e. w_1 ***on*** w_2 vaut `0`) et on ne consulte pas l'horloge d'échantillonage (w_2).

[5] La notation $w[i]$ désigne le $i^{ième}$ élément du mot w.

Le calcul des tailles du préfixe et du motif périodique du résultat d'une opération **on** est un peu compliqué. Le cas le plus simple est celui où le nombre de 1 du préfixe et du motif périodique du premier mot correspondent à la taille du préfixe et du motif périodique du second :

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

Pour tous mots p_1 et p_2 respectant ces conditions, les tailles du préfixe et du motif périodique de p_1 **on** p_2 sont celles de p_1. En effet, le premier parcours du motif périodique de p_1 commence en même temps que celui du motif périodique de p_2, et c'est là que commence le motif périodique de p_1 **on** p_2. Les parcours suivants du motif périodique des deux mots recommencent en même temps, et c'est aussi là que recommence le comportement périodique du résultat. Le nombre de 1 du préfixe et du motif périodique de p_1 **on** p_2 est alors égal au nombre de 1 de ceux de p_2.

Proposition 5. *Soient p_1 et p_2 telles que $|p_1.u|_1 = |p_2.u|$ et $|p_1.v|_1 = |p_2.v|$. Alors :*

$$\begin{array}{lcllcl} |(p_1 \textbf{ on } p_2).u| & = & |p_1.u| & |(p_1 \textbf{ on } p_2).u|_1 & = & |p_2.u|_1 \\ |(p_1 \textbf{ on } p_2).v| & = & |p_1.v| & |(p_1 \textbf{ on } p_2).v|_1 & = & |p_2.v|_1 \end{array}$$

Nous avons vu dans la remarque 6 qu'il est possible d'augmenter la taille et le nombre de 1 du préfixe et du motif périodique des mots. Ainsi, on peut toujours ajuster les opérandes du **on** afin qu'ils aient la forme de celles du cas simple ci-dessus.

Remarque 7. *Il n'est en fait pas indispensable d'allonger explicitement la taille des opérandes, on peut se contenter de calculer la taille qu'ils auraient après cet ajustement pour avoir la taille du résultat. Si l'on veut effectuer l'opération p_1 **on** p_2, on suppose donc qu'on mette p_1 sous la forme p'_1 et p_2 sous la forme p'_2 telles que $|p'_1.u|_1 = |p'_2.u|$ et $|p'_1.v|_1 = |p'_2.v|$. Le résultat de p_1 **on** p_2 est de la taille de p'_1 et son nombre de 1 est celui de p'_2.*

Enfin, il n'est pas nécessaire de calculer le mot w_1 **on** w_2 pour avoir l'indice de son $j^{ième}$ 1. En effet, si le $j^{ième}$ 1 de w_2 est le $i^{ième}$ élément de w_2 ($\mathcal{I}_{w_2}(j) = i$) , alors le $j^{ième}$ 1 de w_1 **on** w_2 est à l'indice du $i^{ième}$ 1 de w_1 ($\mathcal{I}_{w_1 \textbf{ on } w_2}(j) = \mathcal{I}_{w_1}(i)$).

Proposition 6 (indice du $j^{ième}$ 1 de w_1 **on** w_2)**.**

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

Nous disposons maintenant de tous les outils algébriques nécessaires à la mise en place de l'algorithme de résolution des contraintes d'adaptabilité sur les mots périodiques.

5. Algorithme de résolution des contraintes d'adaptabilité

Nous avons vu dans la section 3 que le calcul d'horloge nécessite la résolution d'un système de contraintes d'adaptabilité. Après calcul des **on** sur les opérandes connues, ces contraintes sont soit de la forme $p_x <: p_y$, soit de la forme c_x **on** $p_x <: c_y$ **on** p_y, où p_x et p_y sont des mots connus de taux non nul et c_x et c_y des mots inconnus. Résoudre le système consiste à trouver des valeurs pour ces inconnus telles que les contraintes du système sont toujours satisfaites.

Tout comme les contraintes d'unification, les contraintes d'adaptabilité ne peuvent être résolues de manière gloutonne. En effet, pour un taux donné, il n'existe pas un mot périodique qui soit plus petit que tous les autres (ou plus grand que tous les autres) pour la relation $\preceq$[6] et donc pour la relation $<:$.

[6]Les mots (1) et (0) constituent une exception à cette règle. En effet, pour tout p, (1) $\preceq p \preceq$ (0)

Par exemple : $\ldots <: \texttt{1}^{999}\texttt{(10)} <: \ldots <: \texttt{1(10)} <: \texttt{(10)} <: \texttt{0(10)} <: \ldots <: \texttt{0}^{999}\texttt{(10)} <: \ldots$ Par conséquent, si on trouve c_1 et c_2 satisfaisant une contrainte c_1 *on* $p_1 <: c_2$ *on* p_2, alors l'infinité de solutions suivantes conviennent aussi :

- $\texttt{1}^{d_1} c_1$ et $\texttt{0}^{d_2} c_2$, quels que soient d_1 et d_2 ;
- $\texttt{1}^{d_1} c_1$ et $\texttt{1}^{d_2} c_2$, quels que soient d_1 et d_2 tels que $d_1 \geq d_2$;
- $\texttt{0}^{d_1} c_1$ et $\texttt{0}^{d_2} c_2$, quels que soient d_1 et d_2 tels que $d_1 \leq d_2$.

La technique consistant à choisir le mot le plus grand selon la relation d'adaptabilité pour une variable située toujours à droite dans les contraintes, et le mot le plus petit pour une variable située toujours à gauche ne peut donc pas s'appliquer dans notre cas. L'inférence de valeurs satisfaisant les contraintes doit forcément être faite de manière globale afin de choisir des mots suffisamment grands, et/ou suffisamment petits pour satisfaire toutes les contraintes.

5.1. Simplification du système de contraintes

Une contrainte de la forme $p_x <: p_y$ ne contenant pas de variables, on ne peut pas influencer sa valeur de vérité. Pour ces contraintes, il suffit de vérifier que la relation d'adaptabilité est satisfaite en appliquant le test de la proposition 3. Si la relation est satisfaite, la contrainte est toujours vérifiée et peut donc être supprimée du système. Dans le cas contraire, le système ne peut être satisfait.

Reprenons l'exemple du `cyclic_encoding`. Le système de contraintes d'adaptabilité à résoudre était le suivant :

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

En appliquant le test d'adaptabilité, on peut vérifier que chacune de ces contraintes est toujours satisfaite. Ici, après simplification, le système est vide.

Dans le cas général, après simplification, le système ne contient que des contraintes sur des expressions avec des inconnues. Par exemple, le système de contraintes de sous-typage C' de la section 3 se ramène au système de contraintes d'adaptabilité suivant (après calcul des *on* et simplification du système) :

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

On peut observer qu'après simplification, toutes les expressions du système contiennent une inconnue.

5.2. Résolution du système de contraintes

L'objectif est de résoudre un système de contraintes d'adaptabilité comme A' qui a la forme suivante :

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

Les valeurs p_1, p_2, p'_2, p''_2, p_3, p'_3 sont des mots binaires ultimement périodiques connus de taux non nul. Les variables c_1, c_2, c_3 sont les inconnues du système. Résoudre le système consiste à associer aux inconnues des mots binaires périodiques de taux non nuls tels que les contraintes soient vérifiées.

Remarque 8. *Si l'on s'autorise à associer aux inconnues des mots de taux nuls, tous les systèmes ont une solution. Par exemple, l'instanciation* $\forall n.\ c_n = \texttt{(0)}$ *est une solution triviale à tous les systèmes. Des solutions de taux nul reviennent à n'exécuter le réseau qu'un nombre fini d'instants. Nous écartons de telles solutions.*

Une contrainte d'adaptabilité $c_x \ \texttt{on}\ p_x <: c_y \ \texttt{on}\ p_y$ se ramène à une contrainte de synchronisabilité et une contrainte de précédence :

$$\begin{array}{rcll} c_x \ \texttt{on}\ p_x <: c_y \ \texttt{on}\ p_y & \Leftrightarrow & \{ \text{ par la proposition 3 } \} & \\ & & (c_x \ \texttt{on}\ p_x \bowtie c_y \ \texttt{on}\ p_y) \ \wedge \ (c_x \ \texttt{on}\ p_x \preceq c_y \ \texttt{on}\ p_y) & \end{array}$$

La contrainte de synchronisabilité est réécrite de la manière suivante :

$$\begin{array}{rcll} c_x \ \texttt{on}\ p_x \bowtie c_y \ \texttt{on}\ p_y & \Leftrightarrow & \{ \text{ par la proposition 1 } \} & rate(c_x \ \texttt{on}\ p_x) = rate(c_y \ \texttt{on}\ p_y) \\ & \Leftrightarrow & \{ \text{ par la définition 3 } \} & \frac{|(c_x \ \texttt{on}\ p_x).v|_1}{|(c_x \ \texttt{on}\ p_x).v|} = \frac{|(c_y \ \texttt{on}\ p_y).v|_1}{|(c_y \ \texttt{on}\ p_y).v|} \end{array}$$

La contrainte de précédence peut quant à elle être réécrite ainsi :

$$\begin{array}{rcll} c_x \ \texttt{on}\ p_x \preceq c_y \ \texttt{on}\ p_y & \Leftrightarrow & \{ \text{ par la proposition 2 } \} & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x \ \texttt{on}\ p_x}(j) \leq \mathcal{I}_{c_y \ \texttt{on}\ p_y}(j) \\ & \Leftrightarrow & \{ \text{ par la proposition 6 } \} & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) \end{array}$$

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

On s'intéresse donc à la taille et au nombre de `1` des préfixes et motifs périodiques des $c_x \ \texttt{on}\ p_x$ et des $c_y \ \texttt{on}\ p_y$. Dans la suite nous utiliserons l'indice n pour parler indifféremment des indices x et y.

Nous avons vu dans la section 4 (proposition 5) que la taille et le nombre de `1` du préfixe et du motif périodique de $c_n \ \texttt{on}\ p_n$ s'expriment facilement en fonction des tailles et nombres de `1` des préfixes et motifs périodiques de c_n et de p_n dans le cas suivant :
- si $|c_n.u|_1 = |p_n.u|$, alors $|(c_n \ \texttt{on}\ p_n).u|_1 = |p_n.u|_1$ et $|(c_n \ \texttt{on}\ p_n).u| = |c_n.u|$.
- si $|c_n.v|_1 = |p_n.v|$, alors $|(c_n \ \texttt{on}\ p_n).v|_1 = |p_n.v|_1$ et $|(c_n \ \texttt{on}\ p_n).v| = |c_n.v|$.

Nous allons choisir de nous ramener à ce cas simple.

Pour cela, nous ajustons les mots connus du système afin que la propriété suivante soit vérifiée : pour toute inconnue c_n, tous les mots p_n, p'_n, ... tels que $c_n \ \texttt{on}\ p_n$, $c_n \ \texttt{on}\ p'_n$, ... apparaissent dans le système ont la même taille de préfixe ($|p_n.u| = |p'_n.u| = \ldots$) et la même taille de motif périodique ($|p_n.v| = |p'_n.v| = \ldots$). Cette opération est toujours possible (grâce à la remarque 6) et ne change pas la sémantique du système.

On peut ensuite faire le choix suivant pour le nombre de `1` des inconnues c_n.

Choix 1 (nombres de `1` des c_n)**.**

$$|c_n.u|_1 = |p_n.u| = |p'_n.u| = \ldots$$
$$|c_n.v|_1 = |p_n.v| = |p'_n.v| = \ldots$$

Remarque 9. *Nous n'avons pas la preuve que ce choix pour le nombre de* `1` *des c_x et c_y ne mène jamais à un échec sur des systèmes ayant une solution. Nous en discuterons dans la section 5.3.*

Grâce à ce choix, nous savons exprimer la taille et le nombre de `1` des préfixes et motifs périodiques des $c_n \ \texttt{on}\ p_n$ en fonction de ceux des c_n et p_n. La contrainte de synchronisabilité devient :

$$\begin{array}{rcll} c_x \ \texttt{on}\ p_x \bowtie c_y \ \texttt{on}\ p_y & \Leftrightarrow & \{ \text{ par la proposition 5 } \} & \\ & & \dfrac{|p_x.v|_1}{|c_x.v|} = \dfrac{|p_y.v|_1}{|c_y.v|} & (1) \\ & \Leftrightarrow & |p_y.v|_1 \times |c_x.v| \ = |p_x.v|_1 \times |c_y.v| & \end{array}$$

La contrainte de précédence devient quant à elle :

$$\begin{array}{rcll} c_x \ \texttt{on}\ p_x \preceq c_y \ \texttt{on}\ p_y & \Leftrightarrow & \{ \text{ par la proposition 5 } \} & \\ & & \forall j,\ 1 \leq j \leq h,\ \mathcal{I}_{c_x}(\mathcal{I}_{p_x}(j)) \leq \mathcal{I}_{c_y}(\mathcal{I}_{p_y}(j)) & (2) \\ & & \text{avec } h = \max(|p_x.u|_1, |p_y.u|_1) + \ \text{ppcm}\ (|p_x.v|_1, |p_y.v|_1) & \end{array}$$

Par exemple, ajustons le système A' afin que tous les échantillonneurs d'une même variable soient de même taille :

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

On choisit maintenant comme nombre de `1` pour les c_n la taille de leurs échantillonneurs :

$$|c_1.u|_1 = 2 \quad |c_1.v|_1 = 1 \qquad |c_2.u|_1 = 0 \quad |c_2.v|_1 = 4 \qquad |c_3.u|_1 = 0 \quad |c_3.v|_1 = 1$$

Par la formule (1), les contraintes de synchronisabilité se ramènent à un système d'équations linéaires sur la taille des motifs périodiques des c_n :

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

Par la formule (2), les contraintes de précédence se ramènent quant à elles à un système d'inéquations linéaires sur les indices des `1` dans les c_n :

$$\left\{ \begin{array}{lcl} \forall j,\ 1 \leq j \leq 3, & \mathcal{I}_{c_1}(\mathcal{I}_{\texttt{10(1)}}(j)) & \leq \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0101)}}(j)) \\ \forall j,\ 1 \leq j \leq 1, & \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0100)}}(j)) & \leq \mathcal{I}_{c_3}(\mathcal{I}_{\texttt{(1)}}(j)) \\ \forall j,\ 1 \leq j \leq 1, & \mathcal{I}_{c_2}(\mathcal{I}_{\texttt{(0001)}}(j)) & \leq \mathcal{I}_{c_3}(\mathcal{I}_{\texttt{(1)}}(j)) \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{lcl} \mathcal{I}_{c_1}(1) & \leq & \mathcal{I}_{c_2}(2) \\ \mathcal{I}_{c_1}(3) & \leq & \mathcal{I}_{c_2}(4) \\ \mathcal{I}_{c_1}(4) & \leq & \mathcal{I}_{c_2}(6) \\ \mathcal{I}_{c_2}(2) & \leq & \mathcal{I}_{c_3}(1) \\ \mathcal{I}_{c_2}(4) & \leq & \mathcal{I}_{c_3}(1) \end{array} \right\} \quad (\mathit{Prec})$$

À partir de là, pour trouver la valeur de nos inconnues c_n, nous allons trouver leur taille et la position de leurs `1`. Ainsi, les tailles $|c_n.v|$ et les indices $\mathcal{I}_{c_n}(j)$ sont les nouvelles inconnues de notre problème. Pour que celles-ci représentent effectivement des mots bien formés, elles doivent satisfaire les contraintes des remarques 2 à 5. S'ajoutent donc aux systèmes de contraintes de synchronisabilité *Sync* et de précédence *Prec* les quatre systèmes suivants qui portent sur les $\mathcal{I}_{c_n}(j)$ recherchés : [7]

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

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

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

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

On utilise enfin un solveur généraliste de contraintes d'inéquations linéaires pour résoudre notre système :

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

La résolution du tel système associé à A' produit les résultats suivants :

$$|c_1.v| = 2 \qquad |c_2.v| = 4 \qquad |c_3.v| = 4$$

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

Grâce à ces informations et à la connaissance des nombres de `1` précédemment choisis pour les préfixes et motifs périodiques des c_n, nous pouvons construire la solution suivante pour notre système A' :

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

[7] On utilisera la notation $\mathcal{I}_{c_n}(j) \in S$ pour désigner l'occurrence de l'inconnue $\mathcal{I}_{c_n}(j)$ dans S.

On connaît maintenant tous les types d'horloges du système de la figure 2. Cela nous fournit d'une part le type d'horloges du nœud `f` qui est $\forall\alpha,\ \alpha$ `on` $c_1 \times \alpha$ `on` $c_2 \rightarrow \alpha$ `on` c_3, c'est-à-dire $\forall\alpha,\ \alpha$ `on 11(10)` $\times\ \alpha$ `on (1)` $\rightarrow \alpha$ `on (0`^{3}`1)`. Cela nous permet d'autre part de calculer les tailles des buffers. Par exemple, on sait que l'horloge d'écriture dans le premier buffer est du type α `on` c_1 `on 10(1)` $= \alpha$ `on 11(10) on 10(1)` et que l'horloge de lecture est du type α `on` c_2 `on (01)` $= \alpha$ `on (1) on (01)`. Selon la proposition 4, la taille nécessaire pour ce buffer est $size($`11(10) on 10(1), (1) on (01)`$) = 1$.

5.3. Correction et complétude de l'algorithme

Nous montrons ici que notre algorithme est correct et ne contient qu'une potentielle source d'incomplétude.

D'une part, la simplification du système présentée dans la section 5.1 est correcte et complète car elle ne fait qu'appliquer le test d'adaptabilité sur les mots binaires ultimement périodiques qui est correct et complet.

D'autre part, l'algorithme de résolution présenté dans la section 5.2 repose sur la réécriture pas à pas du système de contraintes d'adaptabilité en un système d'inéquations linéaires sur les indices des `1` et sur les tailles des mots recherchés. Par les propositions de la section 4, chacune des réécritures transforme le système de contraintes en un système équivalent. La seule étape faisant exception à cette règle est le choix du nombre de `1` dans les mots recherchés c_n. Ce choix n'impacte pas la correction de l'algorithme : il est correct de chercher une solution parmi un sous-ensemble de tous les mots possibles. Par contre, ce choix fait potentiellement perdre la complétude. Il est possible que les solutions soient à l'extérieur du sous-ensemble de mots considéré.

Enfin, considérons la phase de changement d'inconnue où l'on décide de chercher les indices des `1` et la taille des mots plutôt que les mots eux-mêmes. Cette phase est correcte et complète grâce à l'ajout des contraintes *Per*, *Size*, *Init* et *Incr* qui garantissent que les nouvelles inconnues représentent bien des indices de `1` et des tailles de mots périodiques bien formés.

Il est à noter que pour des raisons d'optimisation, les indices des `1` non contraints par les contraintes d'adaptabilité n'apparaissent pas dans le système d'inéquations linéaires. Lors de la construction des mots résultats, ces indices sont choisis en respectant les mêmes contraintes de bonne formation que les indices des `1` qui apparaissent dans le système d'inéquations.

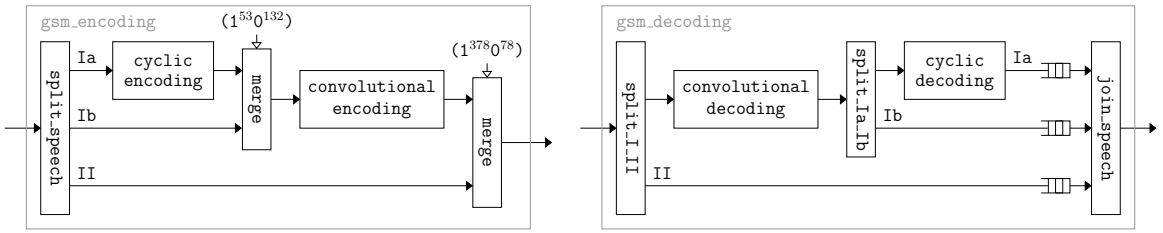
Pour conclure, l'algorithme est correct et sa seule source d'incomplétude est le choix du nombre de `1` des c_n. Nous pensons que ce choix est judicieux : pour l'instant il a toujours mené à une solution sur des systèmes satisfaisables. Par contre, nous savons qu'on peut parfois trouver des solutions permettant une exécution plus rapide du nœud en choisissant un multiple du nombre de `1` choisi ici.

6. Discussion

Dans cette section, nous comparons à travers des exemples la méthode de résolution mise en place dans la section précédente (que nous nommerons résolution concrète) à la méthode à base d'abstraction présentée dans [10] (que nous nommerons résolution abstraite).

L'algorithme de résolution concrète nous a permis de typer un extrait du protocole d'encodage/décodage de canal GSM. Le principe de l'encodeur/décodeur est décrit dans la figure 4 et son code est disponible à l'adresse `http://www.lri.fr/~plateau/jfla11`. Cet exemple illustre les avantages de la résolution concrète.

L'encodeur est représenté sur la figure 4(a). Les différents nœuds contiennent des buffers, mais on peut remarquer qu'ils sont connectés sans utiliser de buffers. Pour typer ce nœud, l'unification globale mentionnée dans la remarque 1 est indispensable. Elle permet de trouver un rythme de consommation



(a) L'encodeur de canal GSM.

(b) Le décodeur de canal GSM.

Dans le protocole GSM [8], la voix est divisée en échantillons de 20ms, qui sont chacun numérisés par une trame de 260 bits. Ces trames sont ensuite encodées, afin de pouvoir détecter et éventuellement corriger les erreurs qui pourraient les atteindre lors de la transmission radio.
Les bits d'une trame sont divisés en trois classes, selon l'importance qu'ils ont dans la qualité de la voix numérisée. La classe Ia contient les 50 premiers bits qui sont les bits les plus importants, la classe Ib les 132 bits suivants qui sont modérément importants et la classe II les 78 derniers bits qui sont les moins importants. Les 50 bits de classe Ia passent à travers l'encodeur cyclique que nous avons programmé dans la section 2. Puis les 132 bits de classe Ib sont concaténés aux 50 bits de classe Ia suivis de leurs 3 bits de redondance cyclique. Les 185 bits ainsi obtenus sont encodés par un encodeur convolutionnnel qui construit un paquet de 378 bits auxquels on concatène finalement les 78 bits de classe II.
Le décodeur suit un schéma symétrique. Il commence par séparer les bits de classe I des bits de classe II qui n'ont pas été encodés. Les bits de classe I sont ensuite traités par un décodeur convolutionnnel qui détecte et corrige des erreurs. Un décodeur cyclique est enfin appliqué sur les bits de classe Ia pour vérifier qu'il ne subsiste plus d'erreur dans cette partie de la trame (si des erreurs subsistent, le décodeur cyclique retransmet les bits de classe Ia de la trame précédente).

Fig. 4 – Extrait du protocole d'encodage/décodage de canal GSM.

du flot d'entrée satisfaisant l'ensemble des contraintes imposées par les trois branches de traitements appliqués à ce flot.

Si l'on utilise l'algorithme de résolution abstraite, on ne peut pas traiter les contraintes d'unification comme deux contraintes de sous-typage. En effet, une paire de contraintes inverses ne peut généralement pas être satisfaite à cause de la perte d'information liée à l'abstraction. Pour pouvoir typer ce nœud avec l'algorithme de résolution abstraite, il faut donc ajouter des buffers pour stocker les flots `Ia`, `Ib` et `II`, afin de transformer les contraintes d'unification en contraintes de sous-typage. Les buffers inutilement estimés nécessaires par la résolution abstraite sont de taille 50, 132 et 78.

Cet exemple montre que l'algorithme de résolution concrète permet de traiter des programmes nécessitant un ordonnancement fin des nœuds. Cet avantage est aussi illustré par la famille des programmes contenant des cycles avec peu de valeurs d'initialisation. Là aussi, l'algorithme avec abstraction échoue à cause de son approximation là où l'algorithme concret trouve des ordonnancements.

Le décodeur est représenté sur la figure 4(b). On peut constater que les flots `Ia`, `Ib` et `II` sont stockés dans des buffers. Les tailles nécessaires inférées pour ces buffers par l'algorithme concret sont respectivement 1, 132 et 156. Avec l'algorithme abstrait, elles sont de 51, 264 et 234. L'algorithme concret divise presque par deux le nombre total de places de buffer estimées nécessaires.

Cet exemple montre que dans le cas où des buffers sont nécessaires, l'algorithme de résolution concrète permet d'évaluer plus finement la taille suffisante pour ces buffers.

Notons cependant que la résolution abstraite garde son intérêt dans le cas d'applications comme l'application vidéo *Picture in Picture* [10]. L'algorithme concret met plusieurs jours à calculer le type. En effet, la taille des mots du système étant de l'ordre de deux millions d'éléments, notre algorithme génère un système d'inéquations linéaires contenant un nombre de variables et de contraintes de cet

ordre de grandeur. Des systèmes de cette taille ne sont pas supportés efficacement par les outils de résolution d'inéquations linéaires comme Glpk [7]. Rappelons aussi que l'algorithme avec abstraction présente l'intérêt de s'appliquer à des systèmes contenant des mots qui ne sont pas exactement périodiques [9].

Selon le type de systèmes, il peut donc être approprié de choisir l'un ou l'autre des algorithmes. Lorsque les mots périodiques sont bien équilibrés, c'est à dire lorsque les `1` sont répartis de manière assez régulière (c'est le cas de ceux du *Picture in Picture*), l'algorithme avec abstraction donne des résultats satisfaisants et en un temps court. Par contre, celui-ci échoue lorsqu'il y a des contraintes difficiles à satisfaire : unification globale nécessaire, cycles dans les contraintes. Lorsque les mots sont mal équilibrés, c'est à dire lorsque les `1` arrivent par rafales (comme ceux du GSM), mais qu'ils ne sont pas trop longs (seulement des centaines d'éléments), alors il est plus avantageux d'utiliser l'algorithme concret : il a moins de risques de rejeter des systèmes ayant une solution, et calcule de manière plus fine les tailles de buffers nécessaires.

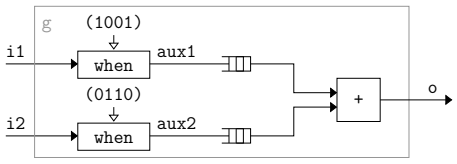
7. Conclusion

Dans cet article, nous avons présenté un algorithme de typage des horloges périodiques en Lucy-n qui tire profit de toute l'information contenue dans les horloges du programme. Le cœur de cet algorithme est la résolution des contraintes de sous-typage. Pour être capables de résoudre ces contraintes, nous avons établi des propriétés sur les mots binaires ultimement périodiques. En appliquant ces propriétés, les contraintes à résoudre se traduisent en un système d'inéquations linéaires qui peut être résolu par des techniques classiques.

Bien qu'il soit moins rapide que l'algorithme présenté l'année dernière, notre nouvel algorithme permet de réussir à typer des programmes où l'échec est inévitable avec l'algorithme de résolution abstraite. En particulier, l'algorithme de résolution concrète permet de typer des programmes modélisant des circuits insensibles à la latence [3]. Les types obtenus pour les différents nœuds d'un tel programme fournissent leurs instants d'activation, construisant ainsi un ordonnancement statique [2, 12] du circuit modélisé. Les perspectives de recherche dans cette direction sont intéressantes de deux points de vue. D'une part, nous pouvons espérer bénéficier des résultats issus de cette communauté pour proposer des ordonnancements équilibrés de programmes Lucy-n, c'est-à-dire répartissant au mieux les instants d'activation des nœuds au cours du temps. D'autre part, nous pensons pouvoir apporter à cette communauté une approche plus compositionnelle que les travaux existants.

Un grand avantage de l'algorithme de résolution concrète est qu'il laisse toute la liberté lors de la résolution des contraintes entre la bufferisation et le ralentissement. Considérons l'exemple suivant :

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



Si l'on donne le type $\forall\alpha.\,(\alpha \times \alpha) \to \alpha$ `on (01)` au nœud `g`, les buffers nécessaires à son exécution sont de taille 1. Mais si l'on décide d'exécuter le système plus lentement en donnant le type $\forall\alpha.(\alpha$ `on (011110)` $\times\ \alpha$ `on (110011)`$) \to \alpha$ `on (010010)` au nœud `g`, il peut alors être exécuté sans buffers. Ces résultats sont tous deux solutions du système d'inéquations linéaires généré par notre algorithme. C'est le choix de la fonction d'objectif pour la résolution des inéquations linéaires qui va mener à l'une ou l'autre des solutions. Dans cet article, nous avons choisi des exécutions au plus tôt. Une perspective intéressante est de comprendre comment adapter le choix de la fonction d'objectif et du nombre de `1` des mots recherchés au critère que l'on cherche à optimiser : délai introduit dans la chaîne de traitements, taille des buffers ou encore vitesse d'exécution.

Remerciements Nous souhaitons remercier chaleureusement Gwenaël Delaval qui nous a fourni l'exemple de l'encodeur/décodeur GSM. Merci aussi à Jean-Christophe Filliâtre qui nous a donné l'idée de diminuer par compression du graphe de contraintes le nombre d'inéquations linéaires à résoudre. Enfin, merci aux rapporteurs dont les commentaires nous ont permis d'améliorer la qualité de ce document.

Références

[1] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-Static Dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, February 1996.

[2] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Formal Methods for Scheduling of Latency-Insensitive Designs. *EURASIP Journal on Embedded Systems*, Issue 1(ISSN:1687-3955), January 2007.

[3] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits ans Systems*, 20(9):1059–1076, September 2001.

[4] 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.

[5] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, May 1996.

[6] 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.

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

[8] Xavier Lagrange, Philippe Godlewski, and Sami Tabbane. *Réseaux GSM : des principes à la norme*. Hermès Science, Paris, 5ème édition edition, 2000.

[9] Louis Mandel and Florence Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. In *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 2009)*, Saint-Quentin sur Isère, France, January 2009. INRIA.

[10] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n : une extension n-synchrone de Lustre. In *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA 2010)*, Vieux-Port La Ciotat, France, January 2010.

[11] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-synchronous extension of Lustre. In *Tenth International Conference on Mathematics of Program Construction (MPC 2010)*, Québec, Canada, June 2010.

[12] Jean-Vivien Millo. *Ordonnancements Périodiques dans les Réseaux de Processus : Application à la Conception Insensible aux Latences*. PhD thesis, Université de Nice-Sophia Antipolis, December 2008.

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

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