Langages synchrones avec horloges périodiques

03 septembre 2006

Bertails Alexandre

Laboratoire de Recherche en Informatique d'Orsay
Équipe Démons

Sous la direction de Marc Pouzet

Table des matières

Chapitre 1

Présentation

1.1 Problématique

Les systèmes multimédias sont des exemples d'applications orientées flots de données pouvant nécessiter un traitement intensif de ces données. Ceux-ci peuvent nécessiter une grande puissance de calcul. Par exemple, une simple opération de lissage par un filtre linéaire 3×3 sur des images issues de la télévision haute définition nécessite 466560000 multiplications par seconde. Or aujourd'hui les applications sont plus compliquées qu'un simple filtre.

Des outils permettant de développer de tels systèmes existent. Dans le commerce, nous pouvons citer *Simulink* (*http ://www.mathworks.com/products/simulink/*). Dans le monde universitaire, on trouve *Ptolemy* [9], un outil développé à Berkeley, Université de Californie.

Les motivations de ce stage sont d'étudier les modèles et les langages permettant de traiter ce genre d'applications. De manière plus générale, nous nous intéressons à des applications nécessitant beaucoup de ressources et ayant des traits *réguliers* et *répétitifs*.

Notre objectif est d'étudier un langage de la famille des *langages synchrones* permettant de décrire des *applications périodiques* et de produire du code sous la forme de boucles imbriquées. En effet, ce genre de code semble plus adapté aux architectures matérielles actuelles.

1.2 Annonce du plan

Ce document répond à deux préoccupations. La première est de présenter le domaine auquel je me suis intéressé durant ce stage. Cette introduction décrira rapidement un cas concret révélateur des applications qui nous intéressent et présentera ensuite succintement quelques formalismes existants.

La partie suivante sera réservée à la description de deux de ces formalismes, à savoir les *Synchronous Data Flow* et StreamIt (partie 2.1, page 5) puis les *langages synchrones* (partie 2.2, page 11). Nous y présenterons les algorithmes et les particularités relatifs aux différents systèmes.

Dans la dernière partie (partie 3, page 15), après avoir défini les *horloges périodiques*, nous étudierons un algorithme de compilation des *langages synchrones* avec *horloges périodiques*.

1.3 Cas concret

En mars 2005, la TNT (**T**élévision **N**umérique **T**errestre) s'installe en France. Cela se traduit par la possibilité de faire parvenir aux clients des images de meilleure qualité. Mais certains traitements spécifiques sur ces images peuvent être nécessaires avant la phase d'affichage.

Dans un document interne [4], Phillips décrit le *Downscaler* dont le but est de transformer une image en haute résolution (1920x1080) vers une image en basse résolution (720x480). Ce système applique des filtres (convolutions) aux pixels avant de décimer l'image (choisir certains pixels parmi d'autres). Nous verrons une implémentation de ce système écrit dans un langage synchrone existant : Lucid Synchrone ([12] et [3]).

1.4 Quelques formalismes

Les *systèmes synchrones* forment un formalisme où le temps est logique et où on suppose que les temps d'exécutions sont négligeables. Divers formalismes existent pour décrire des systèmes synchrones. Ces formalismes peuvent être classés par familles :

- *programmation impérative classique* : on utilise des langages tels que C ou Java. Mais ces langages n'offrent aucune garantie de synchronisme par construction et le programmeur doit tout faire lui-même.
- les *réseaux de processus* : plusieurs formalismes peuvent être regroupés ici tels que les réseaux de Kahn [8], les Synchronous Dataflow [6] ou StreamIt [10].
- les *langages synchrones* : on y retrouve des langages comme Esterel [7], Lustre [2] ou Signal [1].

Comme annoncé plus haut, nous reviendrons sur certains de ces formalismes par la suite.

Remarque : les *réseaux de Kahn* décrivent la composition de processus déterministes communiquant par des files (FIFO). La taille de ces files n'est pas forcément connue statiquement. Les communications se font par envois de messages. On suppose que les files sont fiables et que le délai d'émission est borné. Bien qu'intéressant, ce modèle ne permet de prédire les temps de calcul ni la taille de la mémoire nécessaire.

Chapitre 2

Confrontation de deux modèles existants

Durant ce travail, nous avons cherché à nous placer dans les familles des *langages synchrones*. Cependant, nous nous sommes fortement inspirés des *Synchronous Dataflow* [6] pour étudier l'*aspect périodique*.

Ce chapitre présentera les *Synchronous Dataflow* (et leur variante StreamIt [10]) ainsi que les *langages synchrones*. Nous nous attarderons sur les diverses possibilités d'ordonnancements étudiées dans StreamIt.

En fin de chapitre, nous présenterons une implémentation du *downscaler* en Lucid Synchrone.

2.1 Synchronous Dataflow et StreamIt

2.1.1 Synchronous Dataflow

Présentation

Les *Synchronous Dataflow* [6] — ou SDF — sont un exemple de la programmation par diagrammes de blocs (tracé graphique des blocs de dépendances des processus).

Les SDF sont une version des réseaux de Kahn plus adaptée au temps réel car les taux de production/consommation de données sont connus à priori. Cette façon de faire semble particulièrement adaptée au traitement du signal.

Exemple

La figure 2.1 présente un exemple de SDF.

Les nœuds du graphe sont des nœuds de calcul. Il peuvent être écrits dans n'importe quel langage (par exemple C ou Java).

Les arêtes symbolisent les dépendances de données. A chaque exécution d'un nœud, des données présentes sur les arcs entrants sont consommées. Le nombre de données consommées est inscrit au bout de l'arc, à l'entrée du nœud. De la même façon, le nombre présent sur les arcs sortants représente le nombre de données produites.

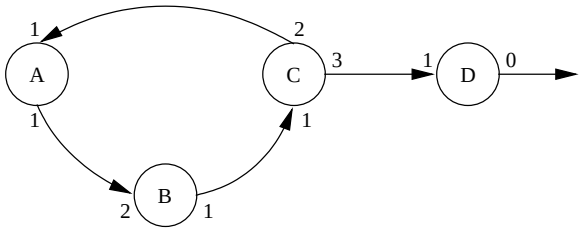

FIG. 2.1 – Exemple de diagramme SDF

Compilation

La compilation des SDF consiste à trouver statiquement un ordonnancement des nœuds qui garantisse une exécution sans erreur. Cet ordonnancement doit vérifier les contraintes suivantes :

- les données sont disponibles à l'activation d'un nœud
- les tampons alloués au début de l'exécution ne débordent jamais

Il existe une classe d'algorithmes prenant en entrée un SDF et calculant un *ordonnancement séquentiel admissible et périodique*. Ces algorithmes sont décrits dans [Static Scheduling of Synchronous Data Flow Programs for Digital SIgnal Processing, 1987].

Un SDF est codé par sa *matrice topologique* Γ donnant pour chaque nœud son taux de production (ou de consommation) représenté par un entier positif (ou négatif). Nous faisons correspondre les nœuds aux lignes et les arcs aux colonnes.

L'algorithme calcule un ordonnancement séquentiel $v(n)(i)$ indiquant si le nœud i est activé à l'instant n. Celui-ci renvoie aussi la taille des tampons relative à cet ordonnancement. Il ne garantit pas une latence minimale ni une taille des tampons minimale (de manière générale, on ne peut pas avoir ces deux propriétés en même temps). Par contre, il garantit le calcul de la période la plus courte.

L'algorithme maintient au cours du calcul l'information sur l'évolution de la taille des tampons au cours du temps dans $b(n)$. On a la relation $b(n+1) = b(n) + \Gamma * v(n)$.

Algorithme d'ordonnancement

- **Entrée** Γ et $b(0)$ (qui montre qu'on peut initialiser le système avec des données dans les tampons)
- **Sortie** b_{max} la taille des tampons sur les arcs et un ordonnancement valide

1. Trouver le plus petit vecteur d'entiers q tel que $\Gamma * q = 0$ (résolution des équations d'équilibre).
2. Former une liste arbitrairement ordonnée L des nœuds du système.
3. Choisir un nœud $i \in L$ exécutable.
4. Si chaque nœud i a été exécuté $q(i)$ fois, renvoyer b_{max} la taille maximale des tampons pour chaque arc au cours du déroulement de l'algorithme : pour tout nœud $i, b_{max}.(i) = max_{k=1...p} b(k).(i)$.
5. Si aucun nœud n'est exécutable, renvoyer une erreur (deadlock).
6. Sinon, retourner au point 3.

Exemple de compilation de SDF

En reprenant l'exemple précédemment donné, voici ce que peut calculer l'algorithme d'ordonnancement (en supposant que le système est initialisé avec une donnée sur l'arc $(B \rightarrow C)$) :

- Ordonnancement : $C - D - D - D - A - A - B$
- Taille des tampons :

Arc	Taille du tampon
$(A \rightarrow B)$	2
$(B \rightarrow C)$	1
$(C \rightarrow A)$	2
$(C \rightarrow D)$	3

2.1.2 StreamIt

Présentation

StreamIt est un sous-ensemble des SDF. Par rapport aux SDF, StreamIt impose les restrictions suivantes :

- chaque nœud de calcul a un seul fil d'entrée et un seul fil de sortie.
- la composition des flots n'est possible qu'au travers d'un certain nombre de constructeurs.

Cependant, il est possible pour un nœud de consulter des données qui n'ont pas encore été consommées. La syntaxe générale du langage est très proche de celle de Java.

Nous présentons maintenant les divers opérateurs du langage permettant de construire des flots.

filter Les nœuds de calcul (`filter`) contiennent le code à exécuter lorsque les données sont en nombre suffisant. On doit spécifier les taux d'entrées/sorties.

pipeline Le `pipeline` consiste juste en l'enchaînement d'un nombre arbitraire de flots.

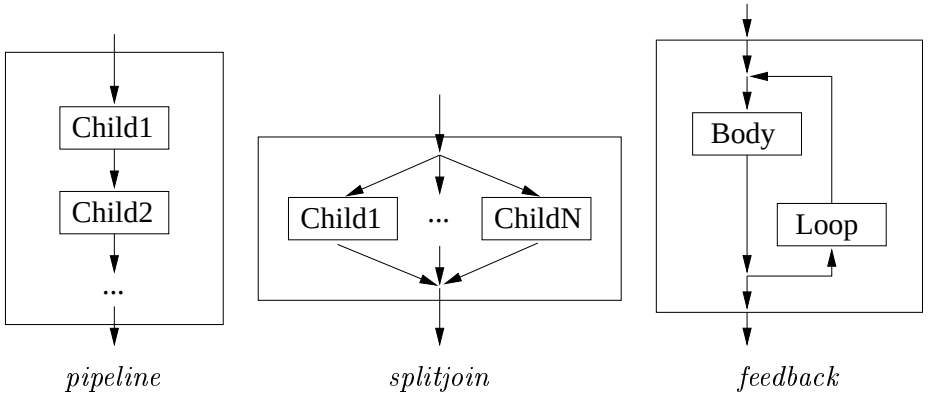

Fig. 2.2 – Les opérateurs

splitjoin Le `splitjoin` permet de répartir des données venant d'un fil vers un nombre arbitraire d'autres fils, reliés eux-même à d'autres nœuds. Les données issues de ces nœuds devront obligatoirement être réunies sur un même fil au moyen d'un recomposeur.

Il existe deux politiques pour distribuer les données :

– `duplicate` : toutes les données arrivant sont recopiées sur tous les fils de sortie.
– `roundrobin` : on peut spécifier comment seront distribuées les données (ex : 3 pour le premier fils, 6 pour le deuxième, etc.).

Les flots sont obligatoirement recomposés selon la politique du `roundrobin`.

feedback Le constructeur `feedback` permet de faire reboucler des flots sur eux-mêmes. Il est parfois nécessaire d'initialiser la boucle en fournissant des données initiales. Cela se fait au moyen du mot clé `enqueue`.

Exemple : suite de Fibonacci

Le calcul de la suite de Fibonacci peut prendre la forme de la figure 2.3.

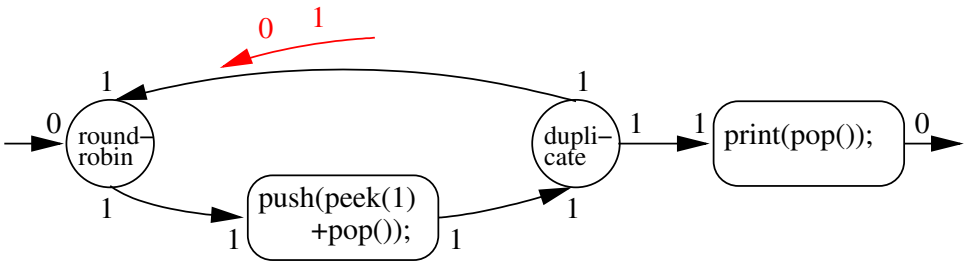

FIG. 2.3 – Diagramme du programme de la suite de Fibonacci

La boucle de retour permet de se souvenir de la précédente valeur calculée. Le nœud `duplicate` ré-injecte dans cette boucle une copie de la dernière donnée calculée et envoie une autre copie dans l'afficheur.

Cela correspond au code `StreamIt` suivant :

```
void->void pipeline Fib {
    add feedbackloop {
        join roundrobin(0, 1);
        body PeekAdd();
        loop Identity<int>();
        split duplicate;
        enqueue 0;
        enqueue 1;
    };
    add IntPrinter();
}

int->int filter PeekAdd {
    work push 1 pop 1 peek 2 {
        push(peek(1) + pop());
    }
}
```

```
int->void filter IntPrinter {
    work pop 1 {
        print(pop());
    }
}
```

Compilation

Il est possible de compiler StreamIt en suivant la méthode de compilation utilisée par les SDF. Cela nécessiterait d'expanser tous les nœuds. Cependant, les restrictions par rapport aux SDF nous donnent quelques avantages. Nous illustrerons les stratégies de compilation au travers de l'exemple suivant :

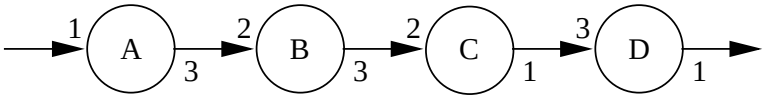

Fig. 2.4 – Exemple illustrant la compilation

Équations d'équilibre Si on note n_A, n_B, n_C et n_D le nombre d'activations de A, B, C et D, on obtient les équations d'équilibre suivantes :

$$\begin{aligned} 3.n_A &= 2.n_B \\ 3.n_B &= 2.n_C \\ 1.n_C &= 3.n_D \end{aligned}$$

La plus petite solution de ce système est :

$$\begin{aligned} n_A &= 4 \\ n_B &= 6 \\ n_C &= 9 \\ n_D &= 3 \end{aligned}$$

Les équations d'équilibre nous apprennent que A doit être exécuté 4 fois, B 6 fois, C 9 fois et D 3 fois. Trois grands algorithmes d'ordonnancement sont possibles (*cf.* [11]).

Single Appearance Schedule L'*ordonnancement à apparition unique* donne une taille de code minimale : chaque nœud n'apparaît qu'une seule fois, inséré dans une boucle.

On obtient l'ordonnancement suivant : $(4A)3((2B)(3C)D)$.

Voici les tailles des tampons obtenus :

Arc	Taille du tampon
$(A \rightarrow B)$	12
$(B \rightarrow C)$	18
$(C \rightarrow D)$	9

En contrepartie de la taille minimale du code, on obtient une taille maximale pour les tampons. De même, la latence (temps entre l'entrée et la sortie d'une donnée dans le *pipeline*) est grande : 9.

Notons que cet ordonnancement n'est pas toujours possible dans le cas de l'opérateur `feedback`.

Push Schedule L'*ordonnancement à consommation immédiate* garantit que les données seront consommées dès que cela sera possible.

Cet ordonnancement nous donne $(2E)CDB(2F)B(2C)2D$ où $E = ABC$ et $F = CE$.

Voici les tailles des tampons obtenus :

Arc	Taille du tampon
$(A \rightarrow B)$	4
$(B \rightarrow C)$	4
$(C \rightarrow D)$	6

Une conséquence immédiate est que la taille des tampons et la latence seront minimales (8 pour la latence).

Les deux stratégies précédentes peuvent être considérées comme extrêmes. Il en existe une troisième offrant un compromis entre les deux.

Phased Minimum Latency Schedule L'*ordonnancement à latence minimale par phase* peut se voir comme un *Single Appearance Schedule* par phase, où chaque phase correspond à la production d'une donnée (ou plutôt *groupe de données* selon le taux de sortie) en sortie du *pipeline*.

On obtient l'ordonnancement suivant : $(2A)(2B)(3C)D(2E)$ où $E = A(2B)(3C)D$.

On a les propriétés suivantes :
- La taille des tampons est inférieure à celle du *Single Appearance Schedule*.

Arc	Taille du tampon
$(A \rightarrow B)$	6
$(B \rightarrow C)$	6
$(C \rightarrow D)$	3

- La latence est la même que dans le *Push Schedule* : 8.

StreamIt a choisi le *Phased Minimum Latency Schedule* pour son algorithme d'ordonnancement parce que la forme particulière des graphes est adaptée à cet algorithme (cela est dû aux restrictions imposées par les opérateurs).

Algorithme d'ordonnancement de StreamIt L'algorithme d'ordonnancement de StreamIt est simple. Il s'agit d'appliquer le *Phased Minimum Latency Schedule* de façon récursive : pour chaque opérateur, on suppose que ses fils sont déjà ordonnés (l'appel récursif a aussi calculé les taux des entrées/sorties), puis on fait l'ordonnancement pour cet opérateur avec du *Phased Minimum Latency Schedule*.

2.1.3 Forces et limitations

Le travail bibliographique et l'écriture de programmes en StreamIt nous ont permis de dresser une liste des principaux avantages du langage :

- Le modèle de programmation est connu depuis longtemps (réseaux de Kahn et SDF). Actuellement, ce modèle semble convenir pour la description de système de flots à haute performance.
- L'ordonnancement des nœuds peut être fait statiquement et de manière efficace avec l'aide de l'algorithme *Phased Minimum Latency Schedule*. Le code généré est sous la forme de boucles imbriquées et il n'y a jamais de tests de disponibilité des données à l'exécution.
- Le langage reste modulaire puisqu'on peut inférer les taux des entrées/sorties lors de la composition des flots.

Cependant, quelques points obscurcissent le tableau :

- On ne peut parler que de systèmes périodiques.
- Il est difficile de décrire des systèmes reconfigurables. On ne peut pas gérer les événements extérieurs. Ce problème devrait être résolu avec le *teleport messaging* qui sera disponible dans la prochaine version du langage. Après lecture des articles parus à ce sujet, nous interprétons le besoin de cette technique comme un exemple des limitations du modèle standard.
- Les combinateurs sont limités et il est parfois difficile de construire certains flots. Exemple : imaginons que nous voulions partager une information entre deux nœuds éloignés : il n'est pas facile de brancher ces nœuds puisque cela nécessite de casser le flot et d'utiliser un `splitjoin`, auquel on doit indiquer les taux de redistribution des données.
- Les nœuds sont limités à un fil en entrée et un fil en sortie. On peut bien sûr former des structures (au sens de C) pour les rassembler, mais on ne peut pas avoir des fils avec des taux différents sur une même entrée/sortie.

2.2 Le modèle synchrone

2.2.1 Présentation

Le modèle synchrone est une version *0-buffer* des réseaux de Kahn où l'activation des nœuds est gardée par des *horloges*. *0-buffer* signifie qu'il est possible d'exécuter le programme sans mécanisme de synchronisation par tampons, mais rien n'empêche d'en utiliser. C'est le modèle de l'électronique. Nous prendrons comme langage exemple Lucid Synchrone [12].

Les types de base manipulés dans ces langages synchrones sont les types issus des langages hôtes étendus aux flots.

Le langage permet de décrire l'évolution des valeurs d'un flot dans le temps. Ainsi, on peut demander à se souvenir de la valeur passée d'un flot (opérateur `pre` dans Lucid Synchrone).

On peut aussi décrire des systèmes évoluant avec plusieurs horloges (et donc plusieurs rythmes). Ces horloges peuvent être échantillonnées (rythmes plus lents) ou sur-échantillonnées (rythmes plus rapides). Dans Lucid Synchrone, cela correspond aux opérateurs `when` et `merge`.

Les horloges peuvent être arbitrairement compliquées. Par exemple, on peut construire une horloge évoluant en même temps que l'énumération des entiers naturels et faisant coïncider ses

tics sur les puissances de 2.

Enfin, les langages synchrones se prettent bien à la vérification et à la preuve de propriétés sur les programmes synchrones. On peut par exemple garantir l'exécution temps réel.

2.2.2 Exemple : suite de Fibonacci

De la même façon que nous avons illustré StreamIt avec un programme calculant la suite de Fibonacci, nous faisons de même avec Lucid Synchrone :

```
let node fibonacci fst snd = fibo where
  rec fibo = fst -> pre_fibo + pre pre_fibo
  and pre_fibo = snd -> pre fibo

let node main () =
  let fibo = fibonacci 0 1 in
  print_int fibo; print_newline ()
```

`fibonacci` est constitué de deux nœuds internes mutuellement récursifs : `pre_fibo` et `fibo`. `fst` et `snd` représentent deux flots d'entiers passés en paramètres mais le programme ne s'intéresse ici qu'à leurs premières valeurs.

`pre_fibo` est le flot `fibo` décalé dans le temps et initialisé avec la première valeur de `snd`. `fibo` vaut donc bien la somme de ses dernières et avant-dernières valeurs. Cette valeur au premier instant est la première valeur de `fst`.

La compilation nous donne les types suivants :

```
val fibonacci :  int -> int => int
val fibonacci :: 'a -> 'a -> 'a
val main :  unit => unit
val main :: 'a -> 'b
```

Pour chaque nœud, on obtient en premier le type des valeurs contenues dans ses flots puis les horloges associées. `fibonacci` prend en entrée deux flots d'entiers et rend un flot d'entiers. Les horloges doivent être les mêmes pour ses trois flots. `main` n'attend pas d'entrée et ne rend pas de valeurs. En conséquence, ses horloges d'entrées/sorties peuvent être quelconques.

2.2.3 Compilation

Chaque nœud du programme est relié à l'horloge de base (*ie.* le rythme le plus rapide du système). Chaque nœud gère la mémoire et les horloges de ses nœuds fils. Si une horloge indique qu'un nœud doit être activé, il est alors appelé et ainsi de suite de manière récursive. Le nœud ainsi appelé peut alors modifier sa mémoire.

Une boucle infinie se charge de
- lire les entrées
- produire les sorties
- modifier l'état du système

au rythme de l'horloge la plus rapide.

2.2.4 Forces et limitations

Lucid Synchrone et Lustre sont très intéressants pour les points suivants :
- Ils possèdent une sémantique bien définie.
- Ils expriment beaucoup mieux le temps-réel.
- Ils peuvent servir à décrire des circuits. On peut faire de la synthèse de circuits à partir de Lustre.
- Ils sont adaptés pour faire de la preuve (*model-checking*, preuve intéractive).
- Il est naturel de parler en terme de fréquence d'activation.
- On peut définir des horloges arbitrairement compliquées et on ne reste donc pas cantonné aux systèmes périodiques.
- Le système reste modulaire dans la mesure où les nœuds compilés sont disponibles au travers de leurs types et horloges.

Cependant :
- Il est parfois utile de vouloir parler en terme de taux de production/consommation et de vouloir connecter des nœuds qui n'ont pas *exactement* ces mêmes taux.
- Le système de types d'horloges des langages synchrones traditionnels n'est pas assez souple. L'égalité d'horloges y est syntaxique (deux horloges sont égales si elles ont le même nom ou si elles sont les composées d'horloges égales). Cette souplesse est présente dans un formalisme nouveau : le N-synchrone (*cf.* [5]).

2.3 Implémentation du *downscaler*

J'ai implémenté le *downscaler* décrit par Phillips dans Lucid Synchrone. Afin de pouvoir le tester, j'ai implémenté :
- un protocole de communication vidéo.
- le *downscaler* lui-même.
- un afficheur vidéo.

Du point de vue de l'utilisateur, l'application prend en entrée une image dans un format haute-définition (1920x1080) et l'affiche à l'écran dans un format basse-définition (720x480).

Protocole de communication vidéo

Le protocole de communication vidéo choisi est le CCIR656 [1]. C'est un protocole série très utilisé par les constructeurs de capteurs vidéos. C'est un *protocole série* qui a la particularité que la dimension des images n'est jamais transmise au module récepteur. Ainsi, les dimensions des images peuvent changer dynamiquement.

Voici une description rapide des différents signaux numériques utilisés par le protocole :
- **Synchronisation horizontale** : symbolise un retour-ligne.
- **Synchronisation verticale** : symbolise un retour-image.
- **Bus de données** : transporte la sérialisation des pixels.

[1] Ce protocole est largement décrit sur le net. Les capteurs OV620 d'Omnivision l'implémentant, voici un lien qui permet de trouver les informations nécessaires : *http ://www.parallax.com/dl/docs/prod/robo/cmucamomnivis.pdf*

Normalement, on utilise aussi un *fil supplémentaire* fournissant une information de pertinance sur les pixels transmis. Cela permet de détecter les instants pendant lesquels sont réalisées les synchronisations. Je l'ai implémenté dans mon module mais je ne m'en sers pas.

Downscaler

Il y a une première passe dans un filtre de convolution (ici un simple filtre isotropique). J'ai pris des coefficients simples car Phillips n'a pas transmis d'informations à ce sujet. Ces coefficients peuvent même être différents dans certaines zones de l'image.

La sélection des pixels se fait en sous-échantillonnant l'horloge. Sur les lignes, on échantillone avec l'horloge (10100100). On a maintenant des lignes de 720 pixels. Sur les colonnes, on échantillone avec l'horloge ($0^{720}1^{720}0^{720}0^{720}1^{720}0^{720}0^{720}1^{720}$).

Afficheur vidéo

Pour afficher les images, j'ai interfacé CamlImage avec Lucid Synchrone. On écrit dans le *framebuffer* de l'image à la réception de chaque pixel. La zone d'affichage est raffraîchie lorsqu'une image complète a été réceptionnée.

Commentaires

J'ai écrit plusieurs versions équivalentes de cette application. Je voulais me faire une idée des styles de programmation offerts par Lucid Synchrone. En particulier, j'ai utilisé les *automates* qui me permettaient de créer facilement un système à états réagissant aux signaux de synchronisation. De même, j'ai utilisé les *signaux* — ici au sens de Lucid Synchrone — pour y encapsuler les pixels. Cela me permettait d'exprimer plus facilement leur présence. Par contre, il devenait plus difficile de gérer les signaux de synchronisation. Finalement, je pense que les horloges seules sont suffisantes pour ce genre de problèmes.

Pour plus de simplicité, les pixels et les signaux de synchronisation sont échantillonnés avec les mêmes horloges. Dans tous les cas, il est indispensable de s'assurer que ces signaux sont bien générés.

Le code est donné en annexe.

Chapitre 3

Compilation des langages synchrones avec horloges périodiques

On constate qu'il existe beaucoup de langages permettant de décrire des *systèmes réguliers*, c'est-à-dire dont les opérations se répètent périodiquement. De même, on constate que les systèmes de ce genre sont particulièrement nombreux.

Les langages synchrones permettent d'ores et déjà de concevoir de tels systèmes. Les horloges, qui commandent l'activation des divers nœuds, sont alors qualifiées de *périodiques*.

Ainsi, une horloge périodique c est un flot booléen qui correspond à la définition suivante :

$$\exists p \in \mathbb{N}, \forall x \in \mathbb{N}, c(x+p) = c(x)$$

Cependant, les méthodes de compilation des langages synchrones restent classiques, la difficulté principale restant la génération des horloges. Ces compilateurs ne cherchent pas à tirer partie du caractère périodique des horloges et on constate que dans certains cas, on peut faire mieux.

L'étude de StreamIt nous a donné des idées pour la compilation des langages synchrones.

Nous étudions dans cette partie une méthode de compilation des *langages synchrones* avec *horloges périodiques*.

3.1 Horloges périodiques

Langage des horloges périodiques Nous présentons ici le langage des *horloges périodiques*.Voici l'expression régulière d'une horloge :

$$clock ::= 0^*1(0|1)^*$$

Le 1 symbolise l'activation d'un nœud et le 0 l'absence d'activation. Par exemple, voici une horloge active un coup sur deux : 10.

Période Si une horloge s'écrit $c_0c_1 \dots c_{p-1}$ alors on dira que sa période est p.

Composition des horloges périodiques ck_1 *on* ck_2 désigne la composition d'horloge. Cela signifie que l'horloge résultante est basée sur ck_2 qui a été échantillonée sur ck_1.

Par exemple, prenons $ck_1 = (101)$ et $ck_2 = (01)$. L'horloge résultante est (001). Les chronogrammes sont donnés dans la figure 3.1.

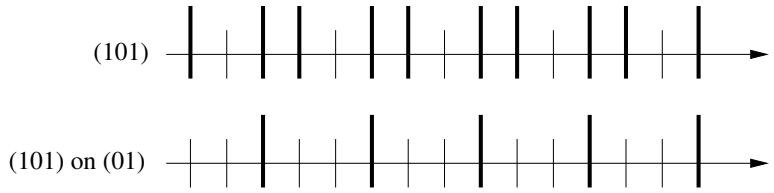

FIG. 3.1 – Chronogrammes pour la composition d'horloges

Types d'horloges Les types d'horloges sont simplement les types d'horloges de base étendus aux produits et aux fonctions (comme en OCAML). De même, on peut généraliser les horloges avec des schémas d'horloges.

Prenons par exemple un nœud qui, à un couple de flots en entrée, renvoie un flot d'horloge moitié. Voici son type : $\forall\alpha, (\alpha \times \alpha) \rightarrow (\alpha\ on\ (10))$.

3.2 Langage

Nous considèrerons le langage d'équations suivant :

$$
\begin{array}{rcl}
e & ::= & x \\
 & & \mid i \\
 & & \mid \texttt{op}\ (e, e) \\
 & & \mid \texttt{op}\ e \\
 & & \mid e\ \texttt{fby}\ e \\
 & & \mid e\ \texttt{when}\ clock \\
 & & \mid \texttt{merge}\ clock\ e\ e \\
eq\_list & ::= & e\ .\ eq\_list \\
 & & \mid e\ . \\
clock & ::= & 0^*1(0|1)^*
\end{array}
$$

Typage Nous supposerons que le langage est équipé d'un *système de typage d'horloge*.

3.3 Ordonnancement cyclique

3.3.1 Vocabulaire et notations

Nous distinguerons par la suite deux types de graphes : le *graphe d'appel* et le *graphe de dépendances*. Le *graphe d'appel* est le graphe issu directement d'un programme de notre langage, étiqueté par les horloges. Le *graphe de dépendances* est le graphe des dépendances de données par appels des nœuds et déplié dans le temps.

Dans ce manuscrit, un *nœud* désignera un nœud au sens de notre langage (comme dans LUSTRE ou dans LUCID SYNCHRONE) et un *sommet* désignera un nœud dans un des graphes de dépendances.

Dans les algorithmes présentés, nous conserverons les conventions suivantes pour le nommage des variables :
- n désignera le nœud pour lequel on recherche un ordonnancement
- i désignera tout nœud fils – ou sous-nœud – de n
- p désignera la période requise par l'ordonnancement. C'est un multiple de la période de l'horloge inférée par le calcul d'horloge.
- p_i désignera la période requise pour l'ordonnancement du nœud i.
- t représentera un instant d'activation d'un nœud. On notera en index le nœud concerné.

Enfin, un ordonnancement sera donné sous la forme d'une suite de couples $(i, vect)$ où i est un sous-nœud de n et $vect$ est un vecteur associant à chaque nœud j dont dépend i l'indice relatif où on trouvera la bonne valeur de j nécessaire à l'exécution de i.

3.3.2 Algorithme d'ordonnancement cyclique

Entrée Un nœud n issu d'un programme synchrone $\mathcal{P}$
Sorties Un 4-uplet $(p, \mathcal{O}^{init}, \mathcal{O}^{cycl}, \mathcal{M})$ où
- p est la période de réaction (valable aussi pour la phase d'initialisation)
- $\mathcal{O}^{init}$ est un ordonnancement des nœuds internes à n pour la phase d'initialisation
- $\mathcal{O}^{cycl}$ est un ordonnancement des nœuds internes à n pour la phase cyclique
- $\mathcal{M}$ est la mémoire nécessaire pour une exécution compatible avec $\mathcal{O}^{init}$ et $\mathcal{O}^{cycl}$

Algorithme

1. Calculer la période p
2. Construire le graphe Γ^{init} des dépendances de données pour la phase d'*initialisation*
3. Construire le graphe Γ^{cycl} des dépendances de données pour la phase *cyclique*
4. Effectuer un tri topologique sur Γ^{init} et récupérer l'ordonnancement $\mathcal{O}^{init}$
5. Effectuer un tri topologique sur Γ^{cycl} et récupérer l'ordonnancement $\mathcal{O}^{cycl}$
6. Calculer la taille de la mémoire $\mathcal{M} = (\mathcal{M}_i)_i$ nécessaire à chaque nœud i
7. Retourner $(p, \mathcal{O}^{init}, \mathcal{O}^{cycl}, \mathcal{M})$

3.3.3 Calcul de la période p

La période p désigne la période nécessaire à l'exécution des ordonnancements $\mathcal{O}^{init}$ et $\mathcal{O}^{cycl}$ (période de réaction) et satisfaisant toutes les périodes p_i des nœuds internes i.

Nous appelerons *période d'initialisation* la période associée à $\mathcal{O}^{init}$ et *période cyclique* la période associée à $\mathcal{O}^{cycl}$.

Une *réaction* définit un cycle dans le sens où l'ordonnancement sera exactement le même entre deux périodes de réaction.

La période p est calculée de la façon suivante :

$$p = ppcm(p_i)_{i \ noeud \ interne}$$

3.3.4 Construction des graphes de dépendances

Représentation d'un graphe

Un nœud du graphe Γ^x où $x \in \{init, cycl\}$ sera représenté par un couple (i, t) où
- i représente un sous-nœud de n
- t est un entier représentant un temps d'activation de i. t est tel que $t \in [\![1, p]\!]$

Opérations sur le graphe

On se donne les primitives suivantes :
- *empty* le graphe vide
- *add_node* Γ (i, t) ajoute le nœud (i, t) au graphe Γ
- *add_edge* Γ (i_1, t_1) (i_2, t_2) ajoute un arc orienté entre (i_1, t_1) et (i_2, t_2) dans le graphe Γ

Construction du tableau intermédiaire des instants d'activation

Le tableau intermédiaire $tab[i][t]$ est un tableau de booléens contenant les instants d'activation des nœuds i. c_i désignera l'horloge associée au nœud i. Le calcul d'horloge nous assure que tab sera le même pour les phases d'initialisation et cyclique.

```
1: for all i node in n do
2:   for all t ∈ [[1, p]] do
3:     tab[i][t] ← false
4:   end for
5: end for
6: for all i node in n do
7:   for all t ∈ [[1, p_i]] do
8:     if c_i[t] = true (ie : i is active at instant t) then
9:       for all t' ∈ [[0, p/p_i − 1]] do
10:        tab[i][p_i × t' + t] ← true
11:      end for
12:    end if
13:  end for
14: end for
```

Notons que ce tableau est commun aux graphes d'initialisation et cyclique.

Graphe de dépendances de la phase cyclique

Instanciation des sommets Cette opération consiste juste à parcourir tab et à instancier un sommet lorsqu'on y trouve la valeur *true*.

```
1: Γ ← empty
2: for all i node in n do
3:   for all t ∈ [[1, p]] do
4:     if tab[i][t] then
5:       add_node Γ (i, t)
6:     end if
7:   end for
8: end for
```

Notons que cette opération parcourt tous les sous-nœuds de n or certains de ces sous-nœuds ne seront appelés que dans l'une ou l'autre des phases d'initialisation ou cyclique. Ces sous-nœuds seront traduits par des sommets inaccessibles dans le graphe de dépendances.

Ajout des arcs de dépendances On supposera que le code du nœud se présente dans une forme adaptée au traitement que nous allons proposer, c'est-à-dire que tous les calculs intermédiaires sont affectés à des nœuds temporaires. On travaillera donc par cas sur les opérateurs du langages, qui devront se trouver à chaque fois derrière une affectation.

On s'occupe dans un premier temps des *dépendances temporelles immédiates* à l'intérieur d'un même nœud. Cela traduit simplement le fait que la valeur d'un nœud à l'instant t doit être calculée avant la valeur à l'instant $t+1$.

1: **for all** i node in n **do**
2: $t_f \leftarrow first\ activation\ instant\ of\ i$
3: **for all** $t \in [\![t_f + 1, p]\!]$ **do**
4: **if** $tab[i][t]$ **then**
5: $t_p \leftarrow precedent\ activation\ instant\ of\ i$
6: $add\_edge\ \Gamma\ (i, t_p)\ (i, t)$
7: **end if**
8: **end for**
9: **end for**

Nous raisonnons maintenant par cas sur les opérateurs du langage.

- $\mathbf{x = x'} \rightarrow$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x', t)\ (x, t)$
  3: **end for**
- $\mathbf{x = x''\ fby\ x'} \rightarrow$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: **if** t is not the first instant in p **then**
  3: $t' \leftarrow$ immediate precedent instant of t
  4: $add\_edge\ \Gamma\ (x', t')\ (x, t)$
  5: **end if**
  6: **end for**
- $\mathbf{x = x'\ when\ pe} \rightarrow$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x'][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x', t)\ (x, t)$
  3: **end for**
- $\mathbf{x = op\ (x_1, x_2)} \rightarrow$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x_1, t)\ (x, t)$
  3: $add\_edge\ \Gamma\ (x_2, t)\ (x, t)$
  4: **end for**
- $\mathbf{x = merge\ pe\ x_1\ x_2} \rightarrow$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x_1][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x_1, t)\ (x, t)$
  3: **end for**
  4: **for all** $t \in [\![1, p]\!]$ such that $tab[x_2][t]$ is **true do**

5: $add\_edge\ \Gamma\ (x_2, t)\ (x, t)$
6: **end for**

Graphe de dépendances de la phase d'initialisation

Instanciation des sommets Cette opération est la même que dans le cas du graphe de dépendance cyclique. La remarque concernant les sommets inaccessibles est la même ici.

Ajout des arcs de dépendances Nous ferons les mêmes suppositions que dans le cas de la phase cyclique. Les dépendances temporelles sont calculées exactement de la même façon. Il nous reste à présenter les algorithmes traitant les ajouts des arcs pour chaque opérateur du langage. Ces algorithmes seront les mêmes, sauf dans le cas du `fby` qui prendra maintenant en compte le premier instant d'activation.

- $\mathbf{x = x' \rightarrow}$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x', t)\ (x, t)$
  3: **end for**
- $\mathbf{x = x'' \ fby\ x' \rightarrow}$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: **if** t is the first instant in p **then**
  3: $add\_edge\ \Gamma\ (x'', t)\ (x, t)$
  4: **else**
  5: $t' \leftarrow$ immediate precedent instant of t
  6: $add\_edge\ \Gamma\ (x', t')\ (x, t)$
  7: **end if**
  8: **end for**
- $\mathbf{x = x' \ when\ pe \rightarrow}$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x'][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x', t)\ (x, t)$
  3: **end for**
- $\mathbf{x = op\ (x_1, x_2) \rightarrow}$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x_1, t)\ (x, t)$
  3: $add\_edge\ \Gamma\ (x_2, t)\ (x, t)$
  4: **end for**
- $\mathbf{x = merge\ pe\ x_1\ x_2 \rightarrow}$
  1: **for all** $t \in [\![1, p]\!]$ such that $tab[x_1][t]$ is **true do**
  2: $add\_edge\ \Gamma\ (x_1, t)\ (x, t)$
  3: **end for**
  4: **for all** $t \in [\![1, p]\!]$ such that $tab[x_2][t]$ is **true do**
  5: $add\_edge\ \Gamma\ (x_2, t)\ (x, t)$
  6: **end for**

3.3.5 Ordonnancement et mémoire

Dans cette partie, nous nous intéressons à la génération de l'ordonnancement des nœuds. Cette phase repose en grande partie sur un tri topologique des sommets des graphes de dépendances de données. Nous prouverons dans un premier temps que les graphes obtenus sont bien

acycliques. Ensuite, nous donnerons l'algorithme d'ordonnancement qui prend aussi en compte la mémoire utilisée.

Graphes acycliques

Nous démontrons ici que les graphes Γ^{init} et Γ^{cycl} sont *acycliques*.

D'après les algorithmes d'ajout des arcs pour chaque graphe, on remarque qu'il n'y a que deux types d'arcs possibles :

- les arcs selon l'axe du temps : par construction, il sont forcément orientés dans le sens croissant du temps et il n'est pas possible d'obtenir un arc *en arrière*.
- les arcs issus des dépendances d'appel sur un même instant t : supposons qu'il existe un cycle $(i,t) \rightarrow (i_1,t) \rightarrow \cdots \rightarrow (i_n,t) \rightarrow (i,t)$. Cela signifierait que le nœud i dépend instantanément de lui-même. Ce cas est impossible parce que garanti par le *typage d'horloge*.

Algorithme d'ordonnancement

Chaque graphe de dépendances obtenu précédement nous donne un ordre d'exécution des calculs intermédiaires pour produire les sorties. Il nous suffit donc d'effectuer un *tri topologique* pour obtenir un ordonnancement.

Nous pouvons distinguer deux grandes stratégies de parcours du graphe lors du tri topologique :

- en privilégiant un parcours selon les *appels de nœuds*, on essaie de consommer les données dès lors qu'elles sont disponibles et donc on réduit la consommation de mémoire. En réalité, l'ordonnancement produit serait exactement le même que celui qu'on obtiendrait par la simulation de l'exécution par les techniques de compilations actuelles. Cet ordonnancement ne nous intéresse donc pas ici.
- en privilégiant un parcours selon l'*axe du temps*, on recherche la localité du calcul. Cependant, la taille de la mémoire demandée sera maximale.

Tri topologique La subtilité du tri topologique se situe dans le parcours de la liste d'adjacence de chaque sommet. Notons qu'il faut définir un ordre sur les nœuds pour faciliter plus tard la factorisation de l'ordonnancement. Retenons que la quantité de mémoire nécessaire dépendra directement de la stratégie de parcours du graphe dans le tri topologique.

Gestion de la mémoire L'algorithme d'ordonnancement donne la quantité de mémoire nécessaire à l'exécution de l'initialisation et d'un cycle. Dans la gestion de la mémoire, nous distinguerons la mémoire qui sera nécessaire à l'exécution d'un cycle (mémoire *intra-réaction*) et la mémoire nécessaire au passage d'un cycle à un autre (mémoire *inter-réactions*).

Mémoire intra-réaction Elle correspond par définition à $\mathcal{M}$. Il sera nécessaire de l'initialiser avant d'effectuer l'ordonnancement $\mathcal{O}^{cycl}$.

Mémoire inter-réactions Elle correspond à la mémoire nécessaire à la gestion des `pre` entre deux réactions.

Données vivantes Les *données vivantes* sont les données qui doivent être stockées car on a encore besoin d'elles. On parlera de *distance* avec la donnée la plus fraîche. Afin de pouvoir la calculer, on marquera les sommets que l'on visitera. La plus grande des distances calculées pour un nœud i indiquera la taille de mémoire nécessaire. Ainsi, la mémoire pourra être implémentée sous la forme de tampons circulaires.

1: $\mathcal{O}^{init} = empty\ list$
2: $\mathcal{O}^{cycl} = empty\ list$
3: $\mathcal{S}^{init} = topological\ sort\ of\ \Gamma^{init}$
4: $\mathcal{S}^{cycl} = topological\ sort\ of\ \Gamma^{cycl}$
5: **for all** node i in n **do**
6: $\quad \mathcal{M}_i = 1$
7: **end for**
8: **for all** (i,t) according to $\mathcal{S}^{init}$ **do**
9: $\quad$ mark this node (i,t) as *visited*
10: $\quad$ **for all** (i',t') such that it exists an edge $(i',t') \rightarrow (i,t)$ **do**
11: $\quad\quad$ find the most recent node $(i',\bar{t})$ marked as *visited*
12: $\quad\quad$ $dist$ = number of edges between (i',t') and $(i',\bar{t})$
13: $\quad\quad$ $\mathcal{M}_i = max(\mathcal{M}_i, dist + 1)$
14: $\quad\quad$ $vect_{(i,t)}[i'] = dist$
15: $\quad$ **end for**
16: $\quad$ *add* $(i, vect_{(i,t)})$ *at the end of* $\mathcal{O}^{init}$
17: **end for**
18: **for all** (i,t) according to $\mathcal{S}^{cycl}$ **do**
19: $\quad$ mark this node (i,t) as *visited*
20: $\quad$ **for all** (i',t') such that it exists an edge $(i',t') \rightarrow (i,t)$ **do**
21: $\quad\quad$ find the most recent node $(i',\bar{t})$ marked as *visited*
22: $\quad\quad$ $dist$ = number of edges between (i',t) and $(i',\bar{t})$
23: $\quad\quad$ $\mathcal{M}_i = max(\mathcal{M}_i, dist + 1)$
24: $\quad\quad$ $vect_{(i,t)}[i'] = dist$
25: $\quad$ **end for**
26: $\quad$ *add* $(i, vect_{(i,t)})$ *at the end of*$\mathcal{O}^{cycl}$
27: **end for**

Remarque : l'initialisation de la quantité de mémoire à la valeur 1 nous garantit de pouvoir passer à la réaction suivante.

3.4 Génération de code

Le code est directement généré depuis les informations contenues dans l'ordonnancement. Nous attirons l'attention sur deux opérateurs particuliers : `merge` et `fby`. L'opérateur `merge` permet de recombiner deux flots de données. L'utilisation d'un booléen nous permettra de choisir entre ces deux flots. On supposera que l'ordonnancement nous donne cette information. De même, dans le cas de `fby` lors de la phase d'initialisation, il faut pouvoir traiter le premier instant.

À l'entrée de chaque nœud, il sera nécessaire de décaler le pointeur dans les tampons de sauvegardes des données vivantes.

Les tampons pour la mémoire seront alloués au début du programme. Ensuite, pour chaque phase, on génèrera le code des nœuds et les ordonnancements.

Le compilateur pour ce mini-langage n'a pas été écrit faute de temps. Nous donnons dans la prochaine partie les diverses étapes de calcul sur un exemple.

3.5 Exemple

Considérons le programme suivant :

```
c = 0 fby s
s = c + 1
o1 = s when (01)
o2 = merge (01) o1 (s when (10))
o3 = o2 when (0001)
```

Le calcul du *ppcm* des périodes nous donne 4. Nous travaillons donc sur un cycle de longueur 4. Le graphe de dépendance de données de la phase cyclique (qui est le même que celui de la phase d'initialisation) est donné dans la figure 3.2.

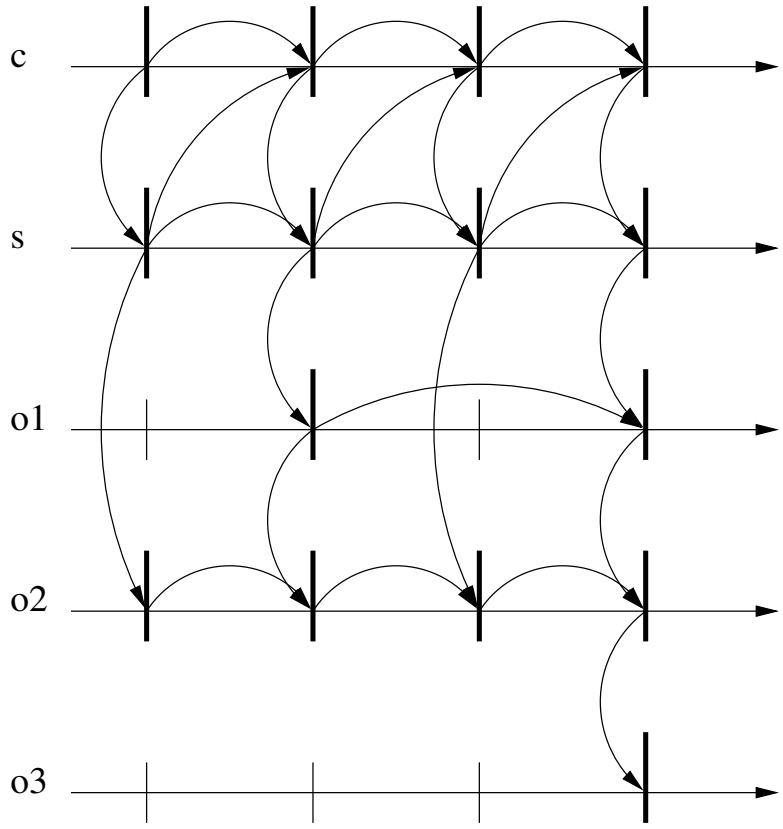

FIG. 3.2 – Graphe des dépendances

Ordonnancement Nous donnons ci-après le résultat de l'algorithme d'ordonnancement :

$$
\begin{array}{lcl}
c & , & \left(s \rightarrow 1 \right) \\
s & , & \left(c \rightarrow 0 \right) \\
c & , & \left(s \rightarrow 1 \right) \\
s & , & \left(c \rightarrow 0 \right) \\
c & , & \left(s \rightarrow 1 \right) \\
s & , & \left(c \rightarrow 0 \right) \\
c & , & \left(s \rightarrow 1 \right) \\
s & , & \left(c \rightarrow 0 \right) \\
o1 & , & \left(s \rightarrow 2 \right) \\
o1 & , & \left(s \rightarrow 0 \right) \\
o2 & , & \begin{pmatrix} s & \rightarrow & 3 \\ o1 & \rightarrow & 0 \end{pmatrix} \\
o2 & , & \begin{pmatrix} s & \rightarrow & 0 \\ o1 & \rightarrow & 1 \end{pmatrix} \\
o2 & , & \begin{pmatrix} s & \rightarrow & 1 \\ o1 & \rightarrow & 0 \end{pmatrix} \\
o2 & , & \begin{pmatrix} s & \rightarrow & 0 \\ o1 & \rightarrow & 0 \end{pmatrix} \\
o3 & , & \left(o2 \rightarrow 0 \right)
\end{array}
$$

Mémoire Voici les tailles des tampons calculées :

$$
\mathcal{M} = \begin{pmatrix} c & \rightarrow & 1 \\ s & \rightarrow & 4 \\ o1 & \rightarrow & 2 \\ o2 & \rightarrow & 1 \\ o3 & \rightarrow & 1 \end{pmatrix}
$$

Code généré Voici le code qui serait généré :

```
open Array

(*** buffer size allocation ***)
let c_buf_len = 1
let s_buf_len = 4
let o1_buf_len = 2
let o2_buf_len = 1
let o3_buf_len = 1

(*** buffer allocation ***)
let c_buf = make c_buf_len 0
let s_buf = make s_buf_len 0
let o1_buf = make o1_buf_len 0
let o2_buf = make o2_buf_len 0
let o3_buf = make o3_buf_len 0

(*** global index ***)
let c_global_index = ref 0
let s_global_index = ref 0
```

```
let o1_global_index = ref 0
let o2_global_index = ref 0
let o3_global_index = ref 0

(*** init phase ***)

let c_code_init b s_index =
    c_global_index := (!c_global_index + 1) mod c_buf_len;
    c_buf.(!c_global_index) <-
        if b then
            0
        else
            s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let s_code_init c_index =
    s_global_index := (!s_global_index + 1) mod s_buf_len;
    s_buf.(!s_global_index) <-
        c_buf.(abs (!c_global_index - c_index) mod c_buf_len) + 1

let o1_code_init s_index =
    o1_global_index := (!o1_global_index + 1) mod o1_buf_len;
    o1_buf.(!o1_global_index) <-
        s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let o2_code_init b o1_index s_index =
    o2_global_index := (!o2_global_index + 1) mod o2_buf_len;
    o2_buf.(!o2_global_index) <-
        if b then
            o1_buf.(abs (!o1_global_index - o1_index) mod o1_buf_len)
        else
            s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let o3_code_init o2_index =
    o3_global_index := (!o3_global_index + 1) mod o3_buf_len;
    o3_buf.(!o3_global_index) <-
        o2_buf.(abs (!o2_global_index - o2_index) mod o2_buf_len)

let scheduling_init () =
    (
        c_code_init true 0;
        s_code_init 0;
        c_code_init false 0;
        s_code_init 0;
        c_code_init false 0;
        s_code_init 0;
        c_code_init false 0;
        s_code_init 0;
        o1_code_init 2;
        o1_code_init 0;
```

```
        o2_code_init true 3 0;
        o2_code_init false 0 1;
        o2_code_init true 1 0;
        o2_code_init false 0 0;
        o3_code_init 0
    )

(*** cyclic phase ***)

let c_code_cycl s_index =
    c_global_index := (!c_global_index + 1) mod c_buf_len;
    c_buf.(!c_global_index) <-
        s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let s_code_cycl c_index =
    s_global_index := (!s_global_index + 1) mod s_buf_len;
    s_buf.(!s_global_index) <-
        c_buf.(abs (!c_global_index - c_index) mod c_buf_len) + 1

let o1_code_cycl s_index =
    o1_global_index := (!o1_global_index + 1) mod o1_buf_len;
    o1_buf.(!o1_global_index) <-
        s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let o2_code_cycl b o1_index s_index =
    o2_global_index := (!o2_global_index + 1) mod o2_buf_len;
    o2_buf.(!o2_global_index) <-
        if b then
            o1_buf.(abs (!o1_global_index - o1_index) mod o1_buf_len)
        else
            s_buf.(abs (!s_global_index - s_index) mod s_buf_len)

let o3_code_cycl o2_index =
    o3_global_index := (!o3_global_index + 1) mod o3_buf_len;
    o3_buf.(!o3_global_index) <-
        o2_buf.(abs (!o2_global_index - o2_index) mod o2_buf_len)

let scheduling_cycl () =
    (
        c_code_cycl 0;
        s_code_cycl 0;
        c_code_cycl 0;
        s_code_cycl 0;
        c_code_cycl 0;
        s_code_cycl 0;
        c_code_cycl 0;
        s_code_cycl 0;
        o1_code_cycl 2;
        o1_code_cycl 0;
```

```
        o2_code_cycl true 3 0;
        o2_code_cycl false 0 1;
        o2_code_cycl true 1 0;
        o2_code_cycl false 0 0;
        o3_code_cycl 0
    )

(*** main entry ***)

let main () =
    scheduling_init ();
    while true do
        scheduling_cycl ()
    done
```

3.6 Discussion et comparaison

Étant donné que notre idée d'étudier les langages périodiques est issue de l'étude des travaux réalisés sur StreamIt, il peut être intéressant de comparer les deux approches et de discuter des limites de notre solution.

3.6.1 Limites

Modularité Le système proposé ici n'est pas modulaire et ne permet pas la définition de nœuds réutilisables.

Boucles Notre toute première motivation était de générer du code sous forme de boucles afin de privilégier la localité du code et ainsi tirer parti au mieux des architectures actuelles. De ce point de vue c'est un échec. Il suffit de regarder les ordonnancement produits pour le comprendre : on ne peut pas beaucoup factoriser à cause des décalages passés en argument.

Explosion combinatoire La méthode actuelle déplie les périodes selon le *ppcm*. Mais cette valeur peut devenir très grande pour des périodes a priori petites. Plusieurs voies ont été explorées, certaines répondent au problème dans certains cas, aucune n'y répond de manière générale.

3.6.2 Comparaison des ordonnancements

Nous avons retrouvé certains aspects de StreamIt lors de notre étude des langages périodiques. Ainsi, le même compromis localité de l'ordonnancement - taille de la mémoire apparaît. Prenons notre exemple développé plus avant. Nous avons vu qu'il existait au moins deux ordonnancements parfaitement distincts :

- $(cs)^4(o_1)^2(o_2)^4o_3$: chaque nœud n'apparait qu'une fois, dans une boucle. Par contre, la mémoire à allouer pour stocker les données intermédiaires est la plus grande possible. C'est l'ordonnancement privilégié dans notre algorithme.
- $(o_2)(cs)^2o_1(o_2)^2(cs)^2o_1o_2o_3$: ici, la mémoire à allouer est la plus petite possible. On cherche à utiliser les données qui viennent d'être calculées le plus rapidement possible.

Ce deuxième ordonnancement peut être généré en se servant des compilateurs traditionnels. En effet, chaque nœud est traduit vers une fonction [1] dont l'exécution est gardée par une horloge.

[1] dans le langage hôte

Si on déplie cette fontion autant de fois que la longueur de la période — on instancie les guardes par les valeurs `true` ou `false` selon les horloges — et qu'on élimine le code mort — une fonction guardée par un `false` — alors on peut générer ce code.

Remarque : Le premier ordonnancement est à comparer au *Single Appareance Schedule* alors que le deuxième est à comparer au *Push Schedule* (*cf* 2.1.2 à la page 9).

Chapitre 4

Conclusion

Ce stage a permis de comparer différents modèles de calculs sur les flots de données : les *Synchronous DataFlow* (et StreamIt) et les systèmes synchrones. Le travail de bibliographie – en particulier à propos les travaux réalisés sur StreamIt – m'a donné l'idée d'étudier les méthodes de compilation des *langages synchrones* avec des *horloges périodiques*.

Nous avons alors étudié un algorithme d'ordonnancement pour un petit langage sur des *équations de flots* (partie 3.3 page 16). Nous avons identifié que le *compromis* entre la *qualité de l'ordonnancement* et la *quantité de mémoire utilisée* reposait sur la stratégie de parcours des graphes des dépendances de données (partie 3.3.5 page 21). Ces choix d'ordonnancements ont alors été comparés à ceux faits dans StreamIt (partie 3.6.2 page 27).

Cependant, nous essayions de produire du code sous la forme de boucles imbriquées. La remarque faite en 3.6.1 page 27 au sujet des arguments de fonctions empêchant la factorisation de code nous indique que cette méthode ne répond pas à la question. De manière générale, on pense que l'approche choisie — générer l'ordonnancement puis le factoriser seulement ensuite — est une erreur. Enfin, elle souffre du défaut de l'explosion combinatoire.

Annexe : downscaler

util.ml

```
open Random
open Graphics
open Unix
open Images
open OImages

(* Note :
      - camlimage place le (0,0) en haut à gauche
      - graphics place le (0,0) en bah à gauche
   Il vaut donc mieux utiliser les fontions redefinies ici
*)

let image_in_width = 1920
let image_in_height = 1080
let image_out_width = 720
let image_out_height = 480

type rgb = {r : int; g : int; b : int}

let from_file_to_rgb file : rgb array array (*: OImages.rgb24_class*) =
  Obj.magic (
    let oimage = OImages.load file [] in
    match OImages.tag oimage with
      | Index8 img ->
          let rgb = img#to_rgb24 in
            img#destroy;
            rgb
      | Index16 img ->
          let rgb = img#to_rgb24 in
            img#destroy;
            rgb
      | Rgb24 img -> img
      | _ -> raise (Invalid_argument "not supported")
  )

let get_red (pix : rgb) (*: int*) = (Obj.magic pix).r
let get_green (pix : rgb) (*: int*) = (Obj.magic pix).g
```

```
let get_blue (pix : rgb) (*: int*) = (Obj.magic pix).b

let my_get (img : rgb array array) (x : int) (y : int) =
  let gget img = (Obj.magic img)#get in
  let c = gget img x y in
  {r = c.r; g = c.g; b = c.b}

let negatif {r = r; g = g; b = b} =
  {r = 255 - r; g = 255 - g; b = 255 - b}

let bw {r = r; g = g; b = b} =
  let y = int_of_float (0.299 *. float_of_int r
                        +. 0.587 *. float_of_int g
                        +. 0.114 *. float_of_int b) in
  {r = y; g = y; b = y}

let rgb {r = r; g = g; b = b} =
  Graphics.rgb r g b

let plot x y = plot x (image_out_height - 1 - y)

let load_img file = dump_image (Graphic_image.of_image (Images.load file []))

let show_image img x y =
  let gr_img = Graphics.make_image img in
  Graphics.draw_image gr_img x y;;


Graphics.open_graph " 720x480";;
let img = from_file_to_rgb "nbc.jpg";;
auto_synchronize false;;
```

it.ls

```
(***********************************************************************)
(*                                                                     *)
(*  Lucid Synchrone                                                    *)
(*                                                                     *)
(*  Author : Marc Pouzet                                               *)
(*  Organization : Demons, LRI, University of Paris-Sud, Orsay         *)
(*                                                                     *)
(***********************************************************************)

(* $Id: *)

(* usefull stuff for streaming apps : sliding window, bounded delay, etc. *)
```

```
open Array

(* lifting of a imported stateful function *)
(* this follow the classical schema (e.g., found in Lutin, Scade, Simulink)*)
(* an imported function f is charactirised by a pair (init, step) *)
(* - init : unit -> 's *)
(* - step : 'a -> 's -> 'b * 's *)

let node lift (init, step) input = o where
  rec (o, s) = step (init () fby s) input

(* counting modulo *)
let node count n = ok where
  rec cpt = n -> if pre cpt <= 0 then 0 else pre cpt - 1
  and ok = false -> pre cpt = 1

let node prefix n = ok where
  rec cpt = n -> if pre cpt <= 0 then 0 else pre cpt - 1
  and ok = (cpt = 0)

(* sliding window (multi-plexing) in O(1) *)
let node multiplexing (static n) (static default) input =
  let static a = Array.make n default in
  let rec index = 0 -> if pre index = n - 1 then 0 else pre index + 1 in
  set a index input;
  a

(* window to stream (de-multi-plexing) in O(1) *)
let node demultiplexing (static n) a =
  let rec index = 0 -> if pre index = n - 1 then 0 else pre index + 1 in
  get a index

(* delay in O(1) *)
let node delay (static n) (static default) input =
  let static a = Array.make n default in
  (* the top and bottom of the array *)
  let rec index = 0 -> if pre index = n - 1 then 0 else pre index + 1 in
  let v = get a index in
  set a index input; v

let node delay_signals (static n) (static default) input =
  let static a = Array.make n default in
  let rec index = 0 -> if ?input
                       then (
                           if pre index = n - 1
                           then 0
                           else pre index + 1
                       )
                       else pre index in
```

```
  output where
  rec output =
      present
        input(i) -> let v = get a index in
                    set a index i; v
      | _ -> last output
      end
  and last output = default
```

video.ls

```
open Graphics
open Array
open Util
open It

(* fonctions d'utilité *)

let node counter () = c where
  rec c = 0 -> pre c + 1

let node counter_to_n n = x where
  rec x = 0 -> if pre x = n - 1 then 0 else pre x + 1

let printer (x,y) =
  print_int x; print_string "\t\t"; print_int y; print_newline ()

let node pixel_counter width height = (x, y, time) where
  rec x = 0 -> if pre x = width - 1 then 0 else pre x + 1
  and y = 0 -> if pre x = width - 1
               then if pre y = height - 1 then 0 else pre y + 1
               else pre y
  and time = true -> if pre x = width - 1 && pre y = height - 1
             then not (pre time)
             else pre time

let node acquire_x_y (vref, href) =
  let clock reset_x = href || vref in
  let clock reset_y = vref in
  let x = reset counter () every reset_x in
  let rec y' = 0 -> if href then pre y' + 1 else pre y' in
  let rec y = 0 ->
    if vref
    then 0
    else (if href then pre y + 1 else pre y) in
  (x, y)
```

```
let convolution (a, z, e, r, t, y, u, i, o) =
  let convolution_c (a, z, e, r, t, y, u, i, o) =
    let a = float_of_int a in
    let z = float_of_int z in
    let e = float_of_int e in
    let r = float_of_int r in
    let t = float_of_int t in
    let y = float_of_int y in
    let u = float_of_int u in
    let i = float_of_int i in
    let o = float_of_int o in
    let c = (a +. z +. e +. r +. 8. *. t +. y +. u +. i +. o) /. 16. in
    int_of_float c in
  let rr = convolution_c (a.r, z.r, e.r, r.r, t.r, y.r, u.r, i.r, o.r) in
  let gg = convolution_c (a.g, z.g, e.g, r.g, t.g, y.g, u.g, i.g, o.g) in
  let bb = convolution_c (a.b, z.b, e.b, r.b, t.b, y.b, u.b, i.b, o.b) in
  {r = rr; g = gg; b = bb}

(* filtres *)

let node filter_convolution (vref, href, pixel) =
  let static image_width = image_in_width in
  let static latence = 2+2*image_width in
  let nb_pixel = let (x,y) = acquire_x_y (vref, href) in
                 x + y * image_width in
  let static pixel_default = {r = 0; g = 0; b = 0} in
  let pix11 = delay (2+2*image_width) pixel_default pixel in
  let pix12 = delay (1+2*image_width) pixel_default pixel in
  let pix13 = delay (2*image_width) pixel_default pixel in
  let pix21 = delay (2+image_width) pixel_default pixel in
  let pix22 = delay (1+image_width) pixel_default pixel in
  let pix23 = delay image_width pixel_default pixel in
  let pix31 = delay 2 pixel_default pixel in
  let pix32 = delay 1 pixel_default pixel in
  let vref' = delay (2+2*image_width) false vref in
  let href' = delay (2+2*image_width) false href in
  (vref', href', convolution (pix11, pix12, pix13,
                              pix21, pix22, pix23,
                              pix31, pix32, pixel))

let clock vf =
  let c = counter_to_n (720*9) in
  if c < 720 then true
  else if c < 720*2 then false
  else if c < 720*3 then true
  else if c < 720*4 then false
  else if c < 720*5 then false
  else if c < 720*6 then true
  else if c < 720*7 then false
```

```
  else if c < 720*8 then false
  else if c < 720*9 then true
  else true

let clock hf =
  let rec hf' = true fby false fby true fby false fby false
      fby true fby false fby false fby hf' in
  hf'

let node downscaler (vref, href, pixel) =
  let vref' = vref when hf when vf in
  let href' = href when hf when vf in
  let pixel' = pixel when hf when vf in
  (vref', href', pixel')

(* le traitement video *)

let node video_in () =
  let (x,y,time) = pixel_counter image_in_width image_in_height in
  let vref = x = 0 && y = 0 in
  let href = x = 0 in
  let pixel = if time
              then my_get img x y
              else negatif (my_get img x y) in
  (vref, href, pixel)

let node video_out (vref, href, pixel) =
  let rec (x,y) = acquire_x_y (vref, href) in
  (if vref then synchronize () else ()); set_color (rgb pixel); plot x y

let node main () =
  let (vref, href, pixel) = video_in () in
  let (vref', href', pixel') = filter_convolution (vref, href, pixel) in
  let (vref'', href'', pixel'') = downscaler (vref', href', pixel') in
  video_out (vref'', href'', pixel'')
```

Bibliographie

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[2] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre : a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.

[3] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[4] Zbigniew Chamski. Sally application example. January 23, 2001.

[5] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. *N*-Synchronous Kahn Networks : a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[6] David G. Messerschmidt Edward A. Lee. Synchronous data flow. September 9, 1987.

[7] G. Berry, P. Couronné et G. Gonthier. Programmation synchrone des systèmes réactifs , le langage Esterel. *Technique et Science Informatique*, 4 :305–316, 1987.

[8] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[9] Edward A. Lee. Technical memorandum no. ucb/erl m03/25. July 2, 2003.

[10] Massachusetts Institute of Technology. *StreamIt Langage Specification, version 2.*, October 2003. Distribution available at : `http ://cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf`.

[11] Saman Amarasinghe Michal Karczmarek, William Thies. Phased scheduling of stream programs. June, 2003.

[12] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006. Distribution available at : `www.lri.fr/~pouzet/lucid-synchrone`.