Calcul de latences dans un programme Prelude

Rémy Wyss[1], Frédéric Boniol[1], Julien Forget[2], Claire Pagetti[1,3]
[1] ONERA - Toulouse, France ; [2] LIFL - Lille, France ; [3] IRIT/ENSEEIHT - Toulouse, France

6 mars 2012

Résumé

L'approche synchrone a été récemment étendue pour permettre la spécification de systèmes multipériodiques rythmés par des horloges temps réel explicites et non plus seulement des horloges booléennes. Cette extension est au cœur du langage Prelude et permet la compilation de la spécification en un système multi-tâches gérées par un ordonnanceur temps réel. Le gain est une implémentabilité accrue des spécifications synchrones. Mais cette extension offre également un autre intérêt : la vérification formelle de propriété temporelle quantitative avant la phase d'implantation. Cet article se focalise sur cette question et sur une propriété temporelle particulière : la latence entre une entrée et son effet produit en sortie du système.

Table des matières

1 Introduction 1

2 Description du système dans un langage d'assemblage synchrone temps réel 4
- 2.1 Description informelle 4
- 2.2 Sémantique 5
- 2.3 Mots de dépendance de données 6

3 Notion de latence 8
- 3.1 Notion de chaîne fonctionnelle 8
- 3.2 Calcul de latences 9

4 Conclusion et perspectives 11

1 Introduction

Le contexte. Les logiciels embarqués critiques sont soumis à des contraintes fortes englobant le déterminisme, la correction logique (i.e., l'absence d'erreur), mais aussi la correction temporelle (l'assurance que les actions sont effectuées au bon moment). La preuve de la satisfaction de toutes ces contraintes par l'implémentation réelle est une activité complexe et majeure. Dans le contexte aéronautique par exemple, les avionneurs doivent convaincre les autorités de certification que les logiciels embarqués sont sûrs. Pour répondre à ce besoin, les concepteurs spécifient leurs applications embarquées depuis de nombreuses années à l'aide de langages formels de haut niveau. Les langages synchrones [2], et en particulier ceux dits flots de données (comme Lustre [13] implanté par l'outil industriel Scade [8]), sont de plus en plus largement utilisés car ils fournissent un bon niveau d'abstraction et s'appuient sur des générateurs de code exécutable éprouvés.

La complexification des systèmes embarqués a conduit ensuite les avionneurs à développer une *sur-couche* au dessus de ces langages permettant de décrire un système complet comme l'assemblage de sous-systèmes synchrones. Des formalismes maison ont vu le jour, tel que le langage ad-hoc TEMPO utilisé par Airbus. Les langages de description d'architecture tels que Aadl [9] répondent également à ce besoin. Leur inconvénient est de sortir du modèle synchrone au niveau global, et ainsi d'en perdre les bonnes propriétés comme la modularité et le déterminisme. Une autre solution a été apportée récemment par le langage d'architecture synchrone Prelude [10]. Prelude est une extension temps réel de Lustre permettant la modélisation de systèmes multi-périodiques composés de nœuds (écrits par exemple en Lustre) communiquant par échange de flots de données et s'exécutant à des périodes potentiellement différentes. L'objectif recherché par Prelude est d'offrir un niveau sémantique synchrone enrichi pas des contraintes temporelles tout en permettant une génération de code sous la forme d'un ensemble de tâches exécutées par politiques d'ordonnancement temps

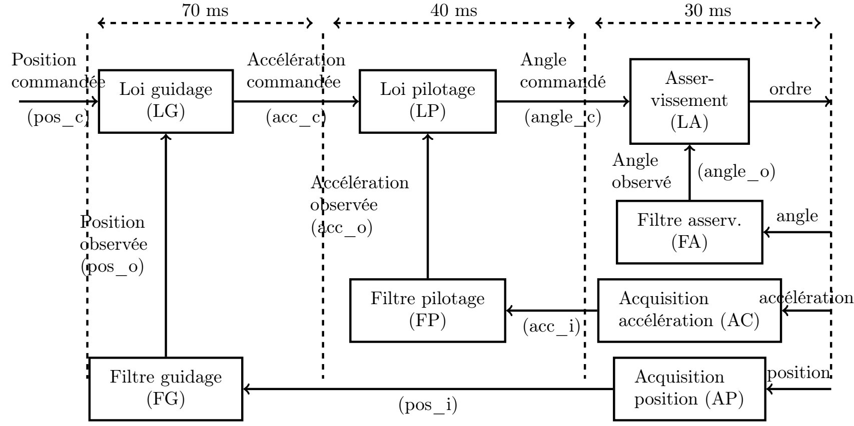
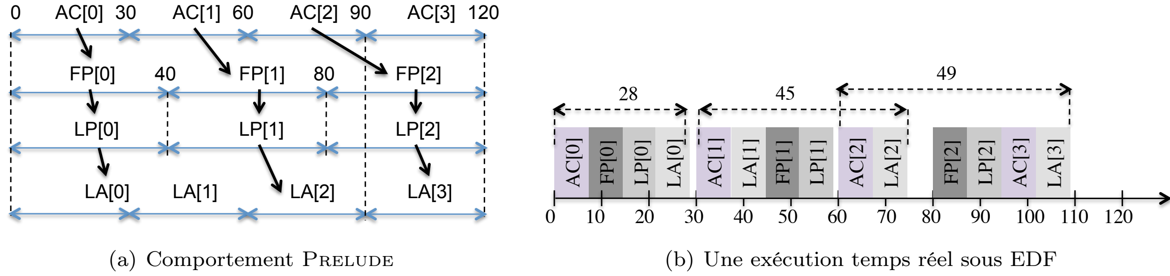


Figure 1 – Système de commande de vol simplifié



(a) Comportement Prelude

(b) Une exécution temps réel sous EDF

Figure 2 – Comportement de la boucle d'asservissement

réel (préemptives ou non) ; l'exécution réelle étant fonctionnellement équivalente au niveau sémantique. Un des intérêts est ainsi de pouvoir vérifier des propriétés temporelles, telles que des latences de bout en bout, dès le niveau spécification en faisant abstraction de l'implantation temps réel finale.

Exemple. Afin d'illustrer notre propos, considérons un exemple simplifié de commande de vol d'un avion. Ce système contrôle l'attitude et la trajectoire de l'avion. Il comprend les organes de pilotage (ex. manche, palonnier), les organes de transmission et de traitement des ordres de l'équipage (calculateurs, bus et câblages), et les servocommandes positionnant les gouvernes. La figure 1 représente l'architecture fonctionnelle de l'application : une boucle rapide à 30ms réalise les asservissements des gouvernes, la boucle intermédiaire à 40ms assure le pilotage automatique et la boucle la moins rapide gère le guidage de l'avion. L'assemblage des nœuds de l'exemple est assez complexe : les tâches communiquent entre elles (quelle que soit leur période), l'ordre des traitements peut avoir un impact sur la correction du système, et le temps de réponse par rapport à une acquisition peut être contraint. Prelude permet de décrire ce système et de préciser les motifs de communication entre tâches. Il existe plusieurs spécifications possibles de l'exemple simplifié des commandes de vol. Imaginons qu'une spécification particulière explicite des motifs de communication qui imposent des précédences comme illustré dans la figure 2(a) : la première valeur produite par AC est consommée par la première instance de FP, qui produit en chaîne une valeur pour la première instance de LP, qui à son tour transmet une valeur à la première instance de LA, ce qui se résume globalement par les précédences $AC[0] \rightarrow FP[0] \rightarrow LP[0] \rightarrow LA[0]$. Ce schéma se reproduit aux itérations suivantes à l'exception de $LP[1] \rightarrow LA[2]$ ($LA[1]$ ne consomme pas de données venant de LP), et $AC[3]$ dont la donnée n'est pas consommée. Le comportement est ensuite répétitif au delà de l'hyperpériode (i.e., le ppcm de toutes les périodes). Quelle que soit l'exécution, les précédences et les communications doivent toujours être respectées. Pour ce faire, le compilateur Prelude génère un ensemble de tâches temps réel avec des contraintes de précédence et un protocole de communication à base de buffers. Ce protocole précise les moments où chaque tâche doit écrire et lire dans les buffers. La figure 2(b) montre une exécution correcte sur un processeur avec une politique d'ordonnancement de type EDF (Earliest Deadline First). L'objectif de l'article est de calculer les latences de chaînes fonctionnelles au niveau de la spécification Prelude et qui seront valides pour toutes les implantations possibles (e.g. indépendantes de la politique d'ordonnancement). Dans l'exemple de la figure 2, la question est de déterminer la latence de la chaîne fonctionnelle $AC \rightarrow FP \rightarrow LP \rightarrow LA$. On montre dans la section 3 que, uniquement à partir de la description Prelude du système et sans présupposer d'implantation

temps réel particulière, cette latence est bornée par 60ms. La latence maximale sur l'exécution de la figure 2(b) est de 49 ms ce qui est bien inférieur à la valeur pire cas prédite par PRELUDE.

Motivation et contribution. Plus généralement, comprendre et prédire des latences induites par l'architecture d'un système (au sens assemblage de fonctions) peut être un enjeu important. D'ordinaire, comme montré sur l'exemple ci-dessus, l'estimation des latences se fait au niveau le plus bas, après l'implantation du système sous une forme mutli-tâches sur un OS temps réel, et donc tard dans le cycle de développement. Si, à ce niveau les latences calculées ou mesurées se révèlent être trop grandes, la seule latitude restante est souvent d'essayer d'optimiser l'implantation du système, voire d'opter pour des composants (processeur, mémoire) plus rapides. Cette solution tardive est souvent limitée, en particulier dans le contexte des systèmes à ressources contraintes (avionique, automobile, etc.). Dans ce contexte, il est souhaitable de pouvoir étudier la dynamique, et en particulier, la ou les latences du système, dès les phases de conception amont, avant l'implantation temps réel. Le modèle de spécification temps réel synchrone offert par le langage PRELUDE permet ce calcul, et donc la vérification d'exigences de latence, directement au niveau spécification. La génération de code multi-tâches préservant la sémantique, la vérification des exigences de latence sera donc également garantie au niveau du code final.

Travaux connexes. Comme indiqué ci-dessus, les latences et les propriétés de performance sont généralement étudiées au niveau de l'implantation. Le temps étant fréquemment vu comme une performance consécutive des choix d'implantation. A ce niveau, trois types de latences ont été étudiés : les temps d'exécution pire cas (WCET) d'une tâche ou d'un thread sur un processeur donné [5, 1], le temps de réponse pire cas (WCRT) d'une transaction (i.e., un ensemble de tâches formant une chaîne) sur une architecture processeur et pour une stratégie d'ordonnancement données [23, 15, 22], et enfin les temps de traversée pire cas (WCTT) de réseaux temps réel (réseaux commutés, bus CAN, etc.) [17, 19]. Ce travaux considèrent cependant tous une architecture d'implantation (ordonnancement et processeur) donnée, et ne peuvent donc intervenir que tard dans le cycle de développement.

Les travaux de O'Neil [20] et Leiserson [18] visent à optimiser le temps d'exécution d'une itération d'un programme synchrone implanté sur des architectures multi-processeurs. Le temps de calcul d'une itération correspond à la durée maximale pour que l'ensemble des tâches du programme soient exécutées. Minimiser le nombre de précédences entre les tâches permet d'augmenter le parallélisme et ainsi diminuer le temps de calcul d'une itération. Ces méthodes se basent sur des graphes de flots de données où les sommets représentent des tâches et les arcs des communications entre ces tâches. Les délais (`fby`) sont des étiquettes associées à un arc. Le graphe d'origine est transformé en un nouveau graphe dont la sémantique est équivalente et dans lequel les (`fby`) ont été déplacés de manière à minimiser le nombre de précédences entre les différentes tâches. Cependant ces travaux sont restreints à des systèmes mono-périodiques.

Plus récemment, plusieurs travaux ont tenté de capturer la notion de latence d'un système global composé de composants logiciels, exécutés sur des processeurs et communiquant via un ou des réseaux temps réel. On peut citer les travaux de F. Carcenac [3] et de M. Lauer [16] qui proposent une méthode d'analyse de latence sur des systèmes avioniques. Cependant, comme dit précédemment, l'ensemble de ces travaux n'adressent eux aussi que le niveau implantation, soit donc un niveau tardif dans le processus de conception.

Dans [24] nous développons une approche similaire à celle présentée dans cet article. Nous définissons formellement la notion de latence sur une spécification fonctionnelle multi-périodique et nous présentons une méthode permettant de calculer une latence à partir de cette spécification. Le calcul présenté repose sur un ensemble de règles d'inférence et est de complexité linéaire. Toutefois les tests effectués ont montré qu'une latence obtenue par calcul était nettement sur-approximative lorsqu'on la compare à celle constatée empiriquement.

Il n'existe à notre connaissance aucune autre formalisation sur la notion de latence au niveau d'une spécification d'architecture fonctionnelle. Cela est dû principalement, nous semble-t-il, aux formalismes et aux sémantiques usuellement utilisés pour de telles spécifications. Deux grandes approches coexistent : l'utilisation de cadres à la AADL [9] ou UML-MARTE [12] qui repoussent la question du temps au niveau des choix d'implantation (ordonnancement temps réel, allocation des priorités, etc.) ; ou des approches synchrones à la LUSTRE qui font abstraction du temps quantitatif et ne considèrent que des horloges booléennes. Dans les deux cas, le calcul de propriétés temporelles sur la spécification sans connaissance du niveau implantation est impossible. Le présent article montre que l'enrichissement temps réel du cadre synchrone apporté par PRELUDE permet de remonter au niveau spécification la vérification de ces propriétés.

Plan. Le langage PRELUDE est présenté succinctement en section 2. La section suivante s'attarde sur la notion de latence et de dépendance de données. Nous définissons formellement la notion de latence en fonction

des dépendances de données. L'exemple du système de commande de vol présenté figure 1 est repris dans chaque section pour illustrer le langage PRELUDE, la notion de latence, et enfin le calcul de latence.

2 Description du système dans un langage d'assemblage synchrone temps réel

2.1 Description informelle

La syntaxe et la sémantique du langage PRELUDE [10] sont proches de LUSTRE [13] mais restreintes à un langage d'assemblage combinant des nœuds importés (i.e., définis dans un autre langage). LUSTRE est un langage synchrone flot de données. De manière informelle, un programme LUSTRE exprime des relations entre les variables d'entrée et de sortie, chaque variable étant une suite infinie de valeurs appelée *flot*. Selon l'hypothèse *synchrone*, un système est décrit sur une échelle de temps discret global et toutes les transformations de flots sont supposées se faire durant une unité de temps appelée *instant*.

PRELUDE reprend en grande partie les hypothèses et la sémantique de LUSTRE en l'étendant avec la notion d'horloge temps réel strictement périodique. Un système PRELUDE est un assemblage de nœuds communicant par consommations et productions synchrones de flots de données. Chaque flot d'entrée de l'assemblage (reçu de l'environnement) est typé par une horloge temps réel *strictement périodique*. C'est ce typage qui introduit la dimension temps réel dans le système : conformément à la sémantique flots de données, chaque nœud de l'assemblage est activé sur l'horloge temps réel de ses flots d'entrée, et produit ses flots de sortie de façon synchrone sur cette même horloge.

Une horloge est strictement périodique si l'intervalle entre deux tops successifs est constant. Ainsi, si $h[i]$ représente la date du ième top de l'horloge h, alors pour tout i, $h[i+1] - h[i]$ vaut toujours la même constante entière (la période de l'horloge), et $h[0]$ est la date du premier top (la phase de l'horloge). Une horloge strictement périodique h est donc définissable par un couple (p, q), où $p \in \mathbb{N}$ est la valeur de sa période ($h[i+1] - h[i] = p$), et $q \in \mathbb{Q}^+$ est la valeur de sa phase en fraction de la période ($h[0] = pq$). On supposera dans la suite que toutes les horloges sont entières. L'horloge la plus rapide est $(1, 0)$ car elle est présente à tout instant à partir du premier instant. De même $(3, 2/3)$ est l'horloge activée toutes les 3 unités de temps et qui commence à 2.

Conformément à la sémantique synchrone, un nœud importé dans un assemblage PRELUDE ne peut combiner des flots d'entrée rythmés par des horloges différentes. PRELUDE introduit trois opérateurs temps réel, agissant sur les flots comme des transformateurs d'horloge :

1. $x *\hat{} k$ est un opérateur d'accélération de rythme, qui sur-échantillonne x sur une horloge k fois plus rapide ;
2. $x /\hat{} k$ est un opérateur de décélération de rythme, qui sous-échantillonne x sur une horloge k fois plus lente ;
3. $x \sim> q$ est un opérateur de déphasage, qui décale x de q fois sa période ;

où $k \in \mathbb{N}$ et $q \in \mathbb{Q}$. Le comportement de ces trois opérations est illustré par le tableau figure 3 (où `fby` est l'opérateur de retard avec initialisation de SCADE).

	1	2	3	4	5	6	7	...	horloge
x	x_0		x_1		x_2		x_3	...	(2,0)
0 `fby` x	0		x_0		x_1		x_2	...	(2,0)
(0 `fby` x) $*\hat{}$ 2	0	0	x_0	x_0	x_1	x_1	x_2	...	(1,0)
(0 `fby` x) $/\hat{}$ 4	0				x_1			...	(4,0)
x $\sim>$ 1/2		x_0		x_1		x_2		...	(2,1/2)

FIGURE 3 – Comportement des opérateurs de PRELUDE

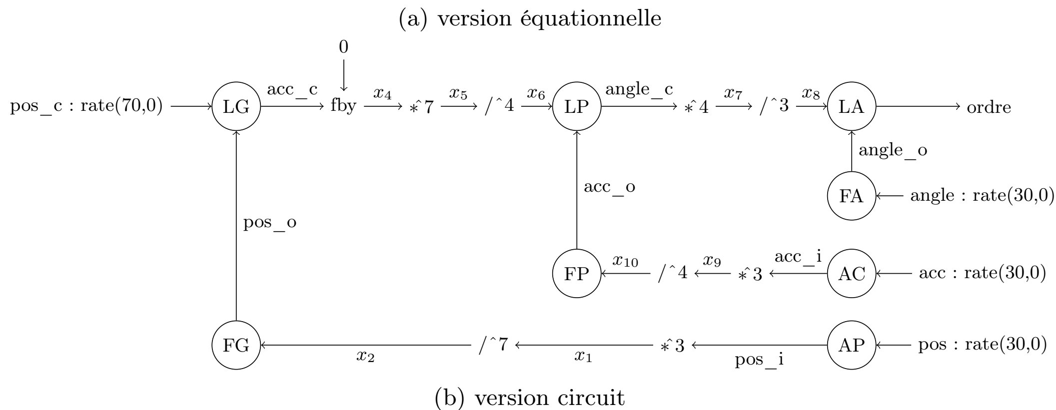
D'un point de vue concret, l'exécution d'un système PRELUDE suit l'hypothèse « synchrone relâchée » [7] : les expressions ne sont pas nécessairement évaluées strictement à chaque top de leur horloge, mais dans l'intervalle défini par leur période avant leur échéance réelle, c'est-à-dire avant leur utilisation par d'autres nœuds du système ; ce qui permet une implantation multi-tâches (un nœud importé par tâche) gérée par un ordonnancer temps réel.

C'est l'explicitation des périodes et des phases des horloges (contrairement à LUSTRE qui n'autorise que des horloges booléennes) et des opérations de changement de rythme qui permet l'analyse de propriétés temporelles, et en particulier des latences, sur un assemblage PRELUDE quels que soient les choix d'implantation respectant l'hypothèse « synchrone relâchée ».

Exemple. Considérons l'exemple des commandes de vol simplifiées (figure 1). Ce système est construit comme l'assemblage de 8 nœuds importés programmés hors de PRELUDE (et non détaillés ici). La figure 4 propose une modélisation PRELUDE de l'exemple des commandes de vol simplifiées sous la forme d'un assemblage "à plat". Deux versions équivalentes de cet assemblage sont proposées : une version textuelle équationnelle, et une version sous forme de circuit graphique. Le système est décrit sous la forme d'un nœud (appelé ici `cdv`), admettant des flots d'entrée (ici, il y en a 4, les trois premiers étant sur l'horloge (30,0), et le dernier `pos_c` sur l'horloge (70,0)), et des flots de sortie (ici `ordre`). Nous supposerons dans la suite que tous les flots internes de l'assemblage sont identifiés par des noms explicites (ici x_1, x_2..., `acc_i`, `pos_i`...). L'assemblage du système est ensuite défini par un ensemble d'équations. Par exemple `pos_i = AP(pos);` définit le flot `pos_i` comme le résultat du nœud `AP` sur le flot `pos` et sur la même horloge que ce flot (i.e., `pos_i` a (30,0) comme horloge). Les deux équations suivantes `x1 = pos_i *^3;` et `x2 = x1 /^7;` synchronisent `pos_i` sur l'horloge (70,0) de manière à permettre l'activation de `FG` sur cette même horloge. Comme en LUSTRE, l'ordre syntaxique des équations n'est pas significatif. Les traitements sont exécutés dans l'ordre induit par les dépendances de données.

```
node cdv (angle : rate(30,0); acc : rate(30,0); pos : rate(30,0); pos_c : rate (70,0))
returns (ordre)
var x1, x2, x4, x5, x6, x7, x8, x9, x10, pos_i, acc_i, pos_o, acc_o, acc_c, angle_c, angle_o;
let
  pos_i = AP(pos);   x1 = pos_i *^3; x2 = x1 /^7 ; pos_o = FG(x2);
  acc_i = AC(acc); x9 = acc_i *^3; x10 = x9 /^4; acc_o = FP(x10);
  acc_c = LG(pos_c, pos_o); x4 = 0 fby acc_c; x5 = x4 *^7; x6 = x5 /^4;
  angle_c = LP(x6, acc_o);  x7 = angle_c *^4; x8 = x7 /^3;
  ordre = LA(x8, angle_o); angle_o = FA(angle);
tel
```

(a) version équationnelle



(b) version circuit

FIGURE 4 – Modélisation PRELUDE de l'exemple des commandes de vol.

Calcul d'horloges. A partir d'une spécification telle que définie ci-dessus, le compilateur PRELUDE infère les horloges de l'ensemble des flots du système. Ce calcul d'horloge se fait simplement par propagation des horloges à travers les équations. Dans l'exemple traité, il est aisé de calculer que `acc_i` et `pos_i` sont sur (30,0) (comme `pos` et `acc`), alors que `x1` est sur (10,0) (multiplication de la fréquence par 3), et `x2` sur (70,0) (division de la fréquence par 7). On calcule ainsi la période $\pi(x)$ et la phase $\varphi(x)$ de l'ensemble des flots x du système. Le lecteur intéressé par le calcul d'horloges de PRELUDE pourra se reporter à [10].

2.2 Sémantique

Dans la suite de l'article, nous considérons que la spécification d'un système est modélisé par $S =< F^i, F, F^o, Eqs, \pi, \varphi >$ où F^i est l'ensemble de flots d'entrée (i.e., les flots définis par aucune équation), F les flots internes, et F^o les flots de sortie, $\pi : F^i \cup F \cup F^o \rightarrow \mathbb{N}$ est la fonction qui associe à chaque flot sa période, $\varphi : F^i \cup F \cup F^o \rightarrow \mathbb{Q}$ associe à chaque flot sa phase (rappelons que π et φ sont calculées par le compilateur du langage), et Eqs est un ensemble d'équations définies par la syntaxe suivante :

$$eqs ::= eq; eqs$$
$$eq ::= (x_1, \dots, x_n) = N(y_1, \dots, y_p) \mid cte \texttt{ fby } x \mid x *\hat{} k \mid x /\hat{} k \mid x \sim> q$$

où $x_i \in F \cup F^o$, $y_i \in F^i \cup F \cup F^o$, N un nœud importé, *cte* une constante, $k \in \mathbb{N}$, et $q \in \mathbb{Q}$. Les équations définissent la structure interne du système. C'est sur cette structure que nous calculerons, par induction, la latence de bout en bout induite par le système. Dans la suite, on ne considère que des systèmes bien formés, c'est-à-dire qui ne contiennent aucun cycle instantané (tous les cycles contiennent au moins un `fby` ou un décalage de phase), et tous les appels de nœuds importés $N(y_1, \ldots, y_p)$ portent sur des flots de même horloge. On supposera donnée dans la suite la fonction sémantique $(\widehat{o_1}, \ldots, \widehat{o_n}) = [\![S]\!](\widehat{i_1}, \ldots, \widehat{i_p})$ qui, à un système S et une valuation des flots d'entrée, associe une valuation des flots de sortie, avec $\{i_1, \ldots, i_p\} = F^i$, $\{o_1, \ldots, o_n\} = F^o$ et $\hat{x}$ est une valuation du flot x.

Sémantique de Kahn. Nous définissons une sémantique de Kahn sur les flots [14]. C'est une adaptation de la sémantique synchrone présentée dans [4] au *Tagged Signal Model*. Pour chaque opération $\diamond$, $\diamond^{\#}(s_1, \ldots, s_n) = s'$ signifie que l'opération $\diamond$ appliquée à une séquence $(s_1, \ldots, s_n)$ produit la séquence s'. Le terme $(v, t).s$ représente le flot dont la tête a la valeur v et le *tag* t et dont la queue est la séquence s. La sémantique des opérations est définie de manière inductive sur la structure des arguments. La sémantique des opérations PRELUDE est donnée en figure 5.

$$
\begin{array}{lcll}
\texttt{fby}^{\#}(v, (v', t).s) & = & (v, t).\texttt{fby}^{\#}(v', s) & \\
\tau^{\#}((v, t).s) & = & (\tau(v), t).\tau^{\#}(s) & \text{où } \tau \text{ est un nœud importé} \\
\hat{*}^{\#}((v, t).s, k) & = & \prod_{i=1}^{k}(v, t'_i). *\hat{}^{\#}(s, k) & \text{où } t'_1 = t \text{ et } t'_{i+1} - t'_i = \pi(s)/k \\
/\hat{}^{\#}((v, t).s, k) & = & \begin{cases} (v, t)./\hat{}^{\#}(s, k) & \text{si } k * \pi(s) | t \\ /\hat{}^{\#}(s, k) & \text{sinon} \end{cases} & \\
\sim>^{\#} ((v, t).s, q) & = & (v, t'). \sim>^{\#} (s, q) & \text{avec } t' = t + q * \pi(s)
\end{array}
$$

FIGURE 5 – Sémantique de Kahn

Exemple 1. *Soit le programme* PRELUDE *suivant :*

```
node ex (i1 : rate(2,0); i2 : rate(5,0); i3 : rate(7,0))
returns (o1, o2, o3)
let
  o1 = tau_1((0 fby (0 fby i1))/^3);
  o2 = tau_2(i2*^5, i3*^7);
  o3= tau_3(o1 ~>1/2 , (o2/^6) ~> 1/2);
tel
```

La sémantique de ce programme est donnée ci-dessous :

	0	1	2	3	4	5	6
$i1$	$i_1[1]$		$i_1[2]$		$i_1[3]$		$i_1[4]$
i_2	$i_2[1]$					$i_2[2]$	
i_3	$i_3[1]$						
0 `fby` i_1	0		$i_1[1]$		$i_1[2]$		$i_1[3]$
0 `fby` (0 `fby` i_1)	0		0		$i_1[1]$		$i_1[2]$
$o1$	$\tau_1(0)$						$\tau_1(i_1[2]$
o_2	$\tau_2(i_2[1], i_3[1])$	$\tau_2(i_2[1], i_3[1])$	$\tau_2(i_2[1], i_3[1])$	$\tau_2(i_1[1], i_2[1])$	$\tau_2(i_2[1], i_3[1])$	$\tau_2(i_2[2], i_3[1])$	$\tau_2(i_2[2], i_3[1])$
o_3				$\tau_3(o_1[1], o_2[1])$			

	7	8	9	10	11	12	...
i_1		$i_1[5]$		$i_1[6]$		$i_1[7]$	...
i_2				$i_2[3]$			...
i_3	$i_3[2]$						...
0 `fby` i_1		$i_1[4]$		$i_1[5]$		$i_1[6]$	...
0 `fby` (0 `fby` i_1)		$i_1[3]$		$i_1[4]$		$i_1[5]$	...
o_1						$\tau_1(i_1[5])$	...
o_2	$\tau_2(i_2[2], i_3[2])$	$\tau_2(i_2[2], i_3[2])$	$\tau_2(i_2[2], i_3[2])$	$\tau_2(i_2[3], i_3[2])$	$\tau_2(i_2[3], i_3[2])$	$\tau_2(i_2[3], i_3[2])$	...
o_3			$\tau_3(o_1[2], o_2[6])$				...

2.3 Mots de dépendance de données

La sémantique du langage implique un ordre partiel dans l'exécution des tâches ainsi qu'un schéma précis des consommations. Si un nœud importé A est appliqué sur un flot résultat d'un nœud importé B, le nœud B doit s'exécuter avant le nœud A. Il y a deux types de consommations : *directes* lorsqu'il s'agit d'un

enchaînement d'opérateurs $/\hat{}\ k$ et $*\hat{}\ k$ et *indirecte* lorsqu'il y a des `fby`. Dans la suite $x[i]$ représentera la i^{ieme} valeur du flot x.

Un mot de dépendance de donnée permet d'exprimer la construction d'une séquence sortie o à partir d'une séquence d'entrée i lorsque o est définie par une expression du type $o = oprs(i)$ où $oprs$ représente une combinaison des opérations $*\hat{}$, $/\hat{}$, `fby`, $\sim>$. Dans la mesure où l'on ne considère que des systèmes multi-périodiques un mot de dépendance de données est un mot fini. Dans la suite, on note $o \to i$ lorsque o dépend de i, c'est-à-dire que o est défini par une expression du type $o = oprs(i)$.

Definition 1 (Mot de dépendance de données)**.** *Un mot de dépendance de données est défini par la grammaire suivante :*

$$\begin{array}{lll} w & ::= & (-1, d_0).(k, d).u \\ u & ::= & (k, d) \mid u.u \end{array}$$

avec $d_0 \in \mathbb{N}$, $k, d \in \mathbb{N}^*$*. Soit* i *une séquence de valeurs et* $x = (-1, d_0).(k_1, d_1).(k_2, d_2). \ldots .(k_n, d_n)$*. un mot de dépendance de données. L'application de* w *à* i *produit une séquence de valeurs* $o = w(i)$ *définie de la manière suivante :*

$$o[p] \to \begin{cases} init & si\ p \in [1, d_0] \\ i[k_1] & si\ p \in [d_0+1, d_0+d_1] \\ i[k_1+k_2] & si\ p \in [d_0+d_1+1, d_0+d_1+d_2] \\ \ldots & \\ i[q.(\sum_{j\in[2,n]} k_j) + \sum_{j\leq l} k_j] & si\ [q.(\sum_{j\in[2,n]} d_j) + \sum_{j\leq l-1} d_j + 1, q.(\sum_{j\in[2,n]} d_j) + \sum_{j\leq l} d_j]\ avec\ q \in \mathbb{N}, l \in [1, n] \end{cases}$$

Les deux premières lettres $(-1, d_0).(k_1, d_1)$ *correspondent au préfixe du mot. Les* d_0 *premières valeurs de* o *sont égales à la valeur initiale (init).* k_1 *est l'index de la première valeur de* i *à être utilisée pour produire* o*.et* d_1 *indique combien de valeurs de* o *sont calculées à partir de* $i[k_1]$*. La séquence* $(k_2, d_2). \ldots .(k_n, d_n)$ *est le motif répétitif du mot : les* d_2 *prochaines valeurs de* o *sont calculées en tenant compte de la valeur* $i[k_1 + k_2]$*, puis les* d_3 *prochaines valeurs de* o *sont calculées à partir de la valeur* $i[k_1 + k_2 + k_3]$*. Ce motif est répétitif ainsi lorsque l'on atteint la dernière paire du motif* (k_n, d_n) *on boucle sur la première valeur du motif* (k_2, d_2)*. Après avoir utilisé la valeur* $i[k_1 + k_2 + \ldots + k_n]$ d_n *fois, les* d_2 *prochaines valeurs de* o *seront calculées à partir de la valeur* $i[k_1 + k_2 + \ldots + k_n + k_2]$ *et ainsi de suite.*

Exemple 2. *Soit une dépendance* $o \to i$ *définie par le mot* $w_1 = (-1, 0).(1, 1).(1, 1)$*. Il s'agit du mot le plus simple. En appliquant la définition ci-dessus on obtient* $o[1] \to i[1]$ *puisque* $d_0 = 0$*,* $k_1 = 1$ *et* $d_1 = 1$*,* $o[2] \to i[2]$ *puisque* $d_2 = 1$ *et* $k_2 = 1$ *ce qui peut être généralisé à* $o[p] \to i[p]$ *avec* $p \in \mathbb{N}$*.*

Si l'on considère l'équation $acc_i = AC(acc)$*. On obtient* $\forall p \in \mathbb{N}^*$

$$acc_i[p] \to AC(acc_i[p])$$

Exemple 3. *Soit une dépendance* $o \to i$ *définie par le mot* $w_2 = (-1, 1), (1, 1), (1, 1)$*, on a :*

$$\begin{cases} o[1] \to init & puisque\ d_0 = 1 \\ o[p+1] \to i[p] & avec\ p \in \mathbb{N}^* \end{cases}$$

Ce mot w_2 *est obtenu à partir de l'équation* `o = 0` **fby** `i`

Exemple 4. *Soit une dépendance* $o \to i$ *définie par le mot* $w_3 = (-1, 0), (1, 4), (1, 4)$*, on a :* $o[1] \to i[1]$*,* $o[2] \to i[1]$*,* $o[3] \to i[1]$*,* $o[4] \to i[1]$ $o[5] \to i[2]$*. Ceci peut être résumé de la manière suivante :*

$$o[p] \to i\left[\left\lceil \frac{p}{4} \right\rceil\right]\ avec\ p \in \mathbb{N}^*$$

Le mot w_3 *est le mot construit à partir de l'équation* `o= i *^ 4`*.*

Exemple 5. *Soit une dépendance* $o \to i$ *définie par le mot* $w_4 = (-1, 0), (1, 1), (3, 1)$*, on a :* $o[1] \to i[1]$*,* $o[2] \to i[4]$ *... qui peut être résumé de la manière suivante :*

$$o[p] \to i[3 * (p-1) + 1]\ avec\ p \in \mathbb{N}^*$$

Le mot w_4 *est le mot construit à partir de l'équation* `o= i /^ 3`*.*

La sémantique d'un programme Prelude peut être exprimée par l'ensemble des mots de dépendance induit par les équations du programme. Ce résultat est présenté dans [21] et les règles de construction sont rappelées dans la propriété 1 page 8.

3 Notion de latence

Comme nous venons de le voir, l'introduction d'horloges temps réel dans un assemblage synchrone permet le calcul des caractéristiques temps réel de chaque flot, et par suite des nœuds les consommant et les produisant, et au final une implantation temps réel multi-tâche. Dans sa thèse J. Forget [11] a défini un schéma de génération de code de programmes Prelude en un code multithreadé ordonnançable par des politiques EDF ou FP sur une architecture monoprocesseur. Dans sa thèse M. Cordovilla a étendu ce schéma à des cibles multiprocesseur [6].

Le langage permet également un second type d'analyse qui n'a pas encore été étudié jusque là, plus proche du domaine de la vérification de propriétés temporelles : la vérification d'exigences de latence de bout en bout.

3.1 Notion de chaîne fonctionnelle

Une latence de bout en bout est toujours relative à une chaîne fonctionnelle reliant un flot d'entrée à un flot de sortie. Soit un système $S =< F^i, F, F^o, Eqs, \pi, \varphi >$. Une chaîne de S est une liste ordonnée $L = (x_1, \dots, x_n)$ de flots telle que chaque x_i $(i > 1)$ est défini par une équation de Eqs en fonction de x_{i-1}. Graphiquement, une chaîne correspond à un chemin continu reliant une entrée à une sortie.

Exemple 6. *Considérons à nouveau le programme* `ex` *de l'exemple 1. Les chaînes fonctionnelles sont :* $L_1 = (i1, o1, o3)$, $l_2 = (i2, o2, o3)$, *et* $L_3 = (i3, o2, o3)$.

Exemple 7. *Si on considère à nouveau le système figure 4, ce système contient quatre chaînes, par exemple* $L_1 = (angle, angle_o, ordre)$ *et* $L_2 = (acc, acc_i, x_9, x_{10}, acc_o, angle_c, x_7, x_8, ordre)$.

Soit $(x_1, \dots, x_n)$ une chaîne fonctionnelle et $oprs$ la liste des opérations que traverse cette chaîne ($x_n = oprs(x_1)$. Pour toutes valeurs de x_1, la séquence des valeurs peut être représentée par un mot de dépendance w.

Propriété 1. *Le principe est de partir du mot initial* $w_{init} = (-1, 0)(1, 1)(1, 1)$ *et de raisonner par induction sur la structure de la liste d'opérations traversées par cette chaîne* $oprs = op_1, \dots, op_n$*. Supposons que le mot courant soit* $w = (-1, d_0)(k_1, d_1) \dots (k_n, d_n)$ *et que le prochain opérateur qui sera appliqué est op, on veut calculer le mot* w' *obtenu à partir de* w *en appliquant op. On distingue alors différents cas :*

1. *Si* $op = \texttt{fby}$*, on obtient* $w' = (-1, d_0 + 1)(k_1, d_1) \dots (k_n, d_n)$.
2. *Si* $op = *\hat{} k$*, le mot* $w' = (-1, k_0 * k)(k_1, d_1 * k), \dots, (k_n, d_n * k)$.
3. *Si* $op = /\hat{}\, k$*, le calcul de* w' *est un peu plus compliqué. Il faut parcourir le mot courant* w *en enlevant* $k - 1$ *consommateurs tous les* k*. On diminue les* d_i *mais il se peut qu'on fasse disparaître des instances consommées. Il faut dans un premier temps calculer le nouveau préfixe* $(-1, \lceil \frac{d_0}{k} \rceil)(i'_1, d'_1)$*. Si* $d_0 \mod k = 0$ *alors* $i'_1 = i_1$ *et* $d'_1 = \lceil \frac{d_1}{k} \rceil$*, sinon* $i'_1 = \sum_{j \leq p} i_j$ *avec* $p = min_j\{d_0 \mod k + \sum_j d_j \geq k\}$ *et* $d'_1 = \left\lceil \frac{d_0 \mod k + \sum_{j \leq p} d_j - k}{k} \right\rceil$*. Il faut ensuite dupliquer r fois le sous-mot* $(i_{p+1}, d_{p+1}), \dots, (i_{l(w)-1}, d_{l(w)-1})$ *ce qui donne le mot* w_i*. Pour terminer et enfin obtenir le mot* w' *il faut faire un raisonnement similaire à celui que nous avons utilisé pour le calcul du préfixe et l'appliquer au mot* w_i*. En pratique on avance dans le mot* w_i *tant que* $\sum d_j \leq k$ *et on ne garde que les instances où cette somme est supérieure à* k*. On a alors* $i' = i_{prec} + \sum_j i_j$ *et* $d' = \left\lceil \frac{d + \sum_j d_j + (d_{prec} \mod k) - k}{k} \right\rceil$.
4. *Si* $op =\sim>$ *ou op = l'application d'un nœud importé alors le mot* w' *est égal au mot courant* w.

La preuve de cette propriété est donnée dans [21].

Exemple 8. *Considérons le nœud* Prelude *suivant :*

```
node ex (x: rate (40,0)) returns (o: rate (30,0))
var x1,x2;
let
  x1 = 0 fby x;
  x2 = x1 *^ 4;
  o = x2 /^ 3;
tel
```

et la chaîne fonctionnelle $L = (x, x_1, x_2, o)$*. La liste des opérations sur la chaîne L est la suivante :* $oprs = (\texttt{fby}, *\hat{}4, /\hat{}3)$*. Le calcul du mot de dépendance entre x et o est donné dans le tableau ci-dessous :*

	0	10	20	30	40	50	60	70	80	90	mot
x	x_1				x_2				x_3		(-1,0,)(1,1)(1,1)
$\rightarrow^{\texttt{fby}}$	0				x_1				x_2		(-1,1)(1,1)(1,1)
$\rightarrow^{\texttt{fby},*\hat{}4}$	0	0	0	0	x_1	x_1	x_1	x_1	x_2	x_2	(-1,4)(1,4)(1,4)
$\rightarrow^{\texttt{fby},*\hat{}4,/\hat{}3}$	0			0			x_1			x_2	(-1,2)(1, 1)(1, 1)(1, 2)(1, 1)

Exemple 9. *Considérons à nouveau le programme* `ex` *de l'exemple 1.*

On calcule mot de dépendance de données associé à l'équation `o1 = tau_1(0 fby (0 fby i1))/^2`. *On construit la liste oprs d'opérateurs en parcourant de manière inductive la structure de l'expression (partie droite de l'équation). On obtient alors la liste oprs* $= [\texttt{fby}; \texttt{fby}; /\hat{}\ 2]$. *On part du mot initial* $w = (-1,0)(1,1)(1,1)$ *et on construit les nouveaux mots en parcourant récursivement la liste oprs. On calcule d'abord le nouveau mot* w_1 *construit à partir de* w *et de l'opération* $op = \texttt{fby}$. *En appliquant la règle sur le* `fby` *de la propriété 1, on obtient* $w_1 = (-1,1)(1,1)(1,1)$. w_1 *devient alors le nouveau mot courant. On s'intéresse ensuite au calcul du mot* w_2 *calculé à partir du mot* w_1 *et de l'opérateur* $op = \texttt{fby}$. *Comme précédemment on applique la règle* `fby` *de la propriété 1 sur le mot* w_1 *et l'on obtient le mot* $w_2 = (-1,2)(1,1)(1,1)$ *qui devient notre nouveau mot courant. Pour terminer on calcule le mot* w_3 *à partir de* w_2 *avec* $op = /\hat{}2$. *En accord avec la définition décrite dans la propriété 1, on commence par calculer le préfixe de* w_3. *Puisque* $2 \mod 2 = 0$ *le préfixe de* w_3 *est :* $(-1, \lceil\frac{2}{2}\rceil)(1, \lceil\frac{1}{2}\rceil) = (-1,1)(1,1)$. *Le motif répétitif du mot* w_2 *est* $(1,1)$. *La somme des* d_i *de ce sous-mot est égale à 1. Il faut donc dupliquer ce motif 2 fois pour obtenir une somme des* d_i *divisible par 2. Nous obtenons un mot* $w_i = (1,1)(1,1)$ *et on calcule la suite du mot* w_3 *en agissant de la même manière que pour le calcul du préfixe sur le mot* w_i. *On parcourt récursivement le mot* w_i *: on fait la somme des* d_i *jusqu'à ce que cette somme soit supérieure ou égale à 2 et on fait la même somme sur les* k_i. *Le nouveau motif calculé est défini de la manière suivante :* $\left(\sum_i k_i, \left\lceil\frac{\sum_i d_i}{2}\right\rceil\right) = \left(1+1, \lceil\frac{1+1}{2}\rceil\right) = (2,1)$ *En le raccrochant au préfixe calculé précédemment on obtient alors le mot* $w_3 = (-1,1)(1,1)(2,1)$.

Les mots obtenus pour les autres équations du programme `ex` *sont les suivants :*

1. *L'équation* `o2 = tau_2(i2*^5, i3*^7)` *donne deux mots de dépendance puisque* o_2 *dépend de deux variables (*i_2 *et* i_3*) :* $\begin{cases} o_2 \rightarrow i_2 : & w_1 = (-1,0)(1,5)(1,5) \\ o_2 \rightarrow i_3 : & w_2 = (-1,0)(1,7)(1,7) \end{cases}$
2. *L'équation* `o3=tau_3(o1 ~> 1/2, (o2/^6)~> 1/2)` *produit également deux mots de dépendance de données puisque* o_3 *dépend à la fois de* o_1 *et* o_2 *:* $\begin{cases} o_3 \rightarrow o_1 : & w_1 = (-1,0)(1,1)(1,1) \\ o_3 \rightarrow o_2 : & w_2 = (-1,0)(1,1)(6,1) \end{cases}$

Exemple 10. *Considérons maintenant l'exemple des commandes de vol et la chaîne allant de acc jusqu'à ordre. Les dépendances de données sont résumées par la figure 6. On voit que :* $ordre[1] \rightarrow acc[1]$, $ordre[2] \rightarrow acc[1]$, $ordre[3] \rightarrow acc[2]$, $ordre[4] \rightarrow acc[3]$ *et* $ordre[5] \rightarrow acc[5]$.

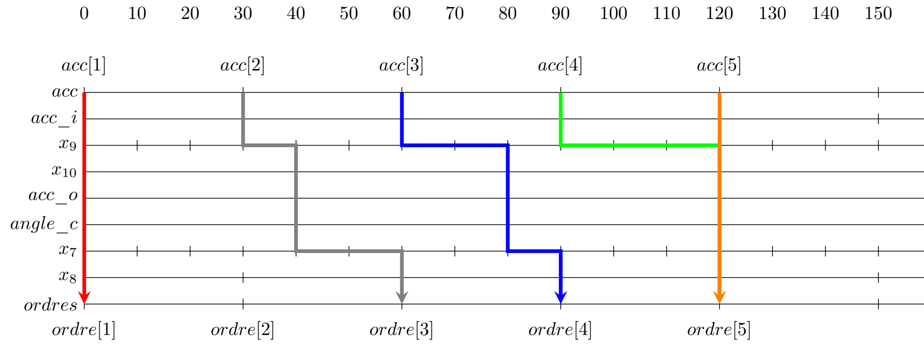


FIGURE 6 – Dépendances de données dans le système des commandes de vol.

3.2 Calcul de latences

Dans cette section on se propose de définir une méthode de calcul de latences à partir d'une spécification fonctionnelle multi-périodique. La méthode que nous exposons est basée sur la notion de mots de dépendances de données.

Définition de la notion de latence d'une chaîne La latence d'une chaîne est le temps nécessaire à la propagation d'un flot le long de cette chaîne. Considérons la chaîne L_1 de l'exemple des commandes de vol. Cette chaîne (voir figure 4) est mono-périodique sur l'horloge (30,0). L'arrivée d'une valeur sur *angle* à un instant t (de l'horloge (30,0)) active dans l'instant le nœud FA qui produit également dans l'instant une valeur sur le flot *angle_ o*, qui active le nœud LA et qui produit une valeur sur *ordre* toujours dans le même instant. On a donc $\forall i \in \mathbb{N}^*.ordre[i] \rightarrow angle[i]$. Du point de vue de la sémantique synchrone, la latence *synchrone* est nulle. En généralisant, on peut se convaincre assez simplement que la latence *synchrone* d'une chaîne mono-périodique est égale au nombre de `fby` qu'elle traverse multiplié par la période de la chaîne.

En réalité les choses sont un peu plus complexes. Nous avons considéré qu'un flot est produit instantanément lorsque sa période est activée : $acc[1]$ est produit à 0 $acc[2]$ est produit à 30 ... Mais un flot est produit dans un intervalle qui est compris entre le début de sa période et la fin de sa période. Plus formellement on peut dire que la valeur $x[i]$ du flot x est produite dans l'intervalle $[\pi(x) * (i-1), \pi(x) * i[$. Ceci a bien entendu une influence sur la latence que nous cherchons à calculer. Cette latence n'est donc pas une valeur mais un intervalle. Si l'on considère à nouveau la chaîne L_1, la latence sur cette chaîne est comprise dans l'intervalle $[0, 30[$.

Le cas des chaînes multi-périodiques est plus complexe car les flots ne sont plus sur les mêmes horloges. Considérons la chaîne L_2, si $ordre[k] \rightarrow acc[i]$ alors la valeur $acc[i]$ est produite au plus tôt à la date $\pi(acc) * (i-1)$ et son impact sur le flot *ordre* est perçu au plus tôt à la date $\pi(ordre) * (k-1)$ et au plus tard à la date $\pi(ordre) * k$. Si l'on considère l'exemple figure 7 les dépendances de données sont décrites par un schéma qui se répète périodiquement. Pour calculer la latence de L_2 il nous faut considérer l'ensemble des dépendances de ce schéma périodique.

Propriété 2 (Latence)**.** *Soit un système $S =< F^i, F, F^o, Eqs, \pi, \varphi >$. Soit une chaîne $L = (i, x_1, \ldots, x_n, o)$ de S. On a donc $i \rightarrow x_1 \rightarrow \ldots \rightarrow o$ et donc par transitivité $i \rightarrow^* o$. Soit $w = (-1, d_0)(k_1, d_1), \ldots, (k_n, d_n)$ le mot de dépendance de données représentant la dépendance $i \rightarrow^* o$. La latence $Lat(L) = [a, b[$ avec a et b définis de la manière suivante :*

$$\begin{cases} a = min_{m\in[1,n]} \left(\pi(o) * \left(\sum_{j=0}^{m} d_j - 1\right) + \phi(o) - \pi(i) * \left(\sum_{j=1}^{m} k_j - 1\right) - \phi(i)\right) \\ b = max_{m\in[1,n]} \left(\pi(o) * \left(\sum_{j=0}^{m} d_j\right) + \phi(o) - \pi(i) * \left(\sum_{j=1}^{m} k_j - 1\right) - \phi(i)\right) \end{cases}$$

Preuve 1. *Soit une chaîne fonctionnelle $L = (i, \ldots, o)$ et un mot $w = (-1, k_0)(k_1, d_1) \ldots (k_n, d_n)$ décrivant les dépendances sur L.*

– *Si on a $o[d_0 + d_1] \rightarrow i[k_1]$ alors la latence entre $i[k_1]$ et $o[d_0 + d_1]$ est :*

$$L_1 = [\pi(o) * (d_0 + d_1 - 1) + \phi(o) - \pi(i) * (k_1 - 1) - \phi(i), \pi(o) * (d_0 + d_1) + \phi(o) - \pi(i) * (k_1 - 1) - \phi(i)[$$

– *Si on a $o[d_0 + d_1 + d_2] \rightarrow i[k_1 + k_2]$ alors la latence entre $i[k_1 + k_2]$ et $o[d_0 + d_1 + d_2]$ est :*

$$L_2 = [\pi(o)*(d_0+d_1+d_2-1)+\phi(o)-\pi(i)*(k_1+k_2-1)-\phi(i), \pi(o)*(d_0+d_1+d_2)+\phi(o)-\pi(i)*(k_1+k_2-1)-\phi(i)[$$

– *En généralisant on a $o[d_0 + \sum_{i=1}^{m} d_i] \rightarrow i[\sum_{i=1}^{m} k_i]$ et :*

$$L_n = [\pi(o)*(d_0+\sum_{i=1}^{m} d_i-1)+\phi(o)-\pi(i)*(\sum_{i=1}^{m} k_i-1)-\phi(i), \pi(o)*(d_0+\sum_{i=1}^{m} d_i)+\phi(o)-\pi(i)*(\sum_{i=1}^{m} k_i-1)-\phi(i)[$$

L'ensemble des latences possibles sur cette chaîne est l'union des intervalles $\cup_{m\leq n} L_m$. Comme on cherche à calculer le plus petit intervalle qui englobe les ensemble L_m, on a bien $Lat(L) =$:

$$\left[min_m \left(\pi(o) * \left(d_0 + \sum_{j=1}^{m} d_j - 1\right) + \phi(o) - \pi(i) * \left(\sum_{j=1}^{m} k_j - 1\right) - \phi(i)\right), max_m \left(\pi(o) * \left(d_0 \sum_{j=1}^{m} d_j\right) + \phi(o) - \pi(i) * \left(\sum_{j=1}^{m} k_j - 1\right) - \phi(i)\right)\right[$$

Exemple 11. *Si l'on considère à nouveau la chaîne L_2, les dépendances de données entre ordre et acc permettent de montrer que la latence de L_2 est comprise dans l'intervalle $[0, 60[$. Ce résultat est illustré par la figure 7.*

La difficulté réside donc dans le fait de calculer ce schéma de dépendance de données. Nous définissons dans la section suivante une méthode permettant de calculer ces dépendances sur une chaîne.

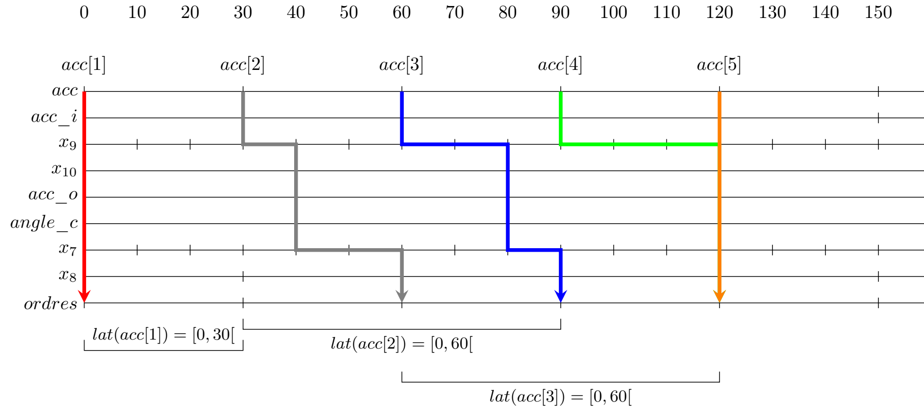


FIGURE 7 – Les différents cas de latences de la chaîne L_2.

Calcul de latence Pour calculer la latence d'une chaîne fonctionnelle :

1. on commence par calculer le mot de dépendance de données de cette chaîne,
2. on calcule ensuite la latence en appliquant la propriété 1.

Exemple 12. *Reprenons l'exemple du système de commande de vol et la chaîne L_2 définie par : $L_2 = (acc, acc_i, x9, x10, acc_o, angle_c, x7, x8, ordre)$. On commence par calculer $w(L_2)$ le mot de dépendance de cette chaîne.*

$w(acc, acc_i)$	$w_1 = (-1,0),(1,1),(1,1)$
$w(acc, acc_i, x9)$	$w_2 = (-1,0),(1,3),(1,3)$
$w(acc, acc_i, x9, x10)$	$w_3 = (-1,0),(1,1),(1,1),(2,1)$
$w(acc, acc_i, x9, x10, acc_o)$	$w_4 = (-1,0),(1,1),(1,1),(2,1)$
$w(acc, acc_i, x9, x10, acc_o, angle_c)$	$w_5 = (-1,0),(1,1),(1,1),(2,1)$
$w(acc, acc_i, x9, x10, acc_o, angle_c, x7)$	$w_6 = (-1,0),(1,4),(1,4),(2,4)$
$w(acc, acc_i, x9, x10, acc_o, angle_c, x7, x8)$	$w_7 = (-1,0),(1,2),(1,1),(1,1),(2,2)$
$w(acc, acc_i, x9, x10, acc_o, angle_c, x7, x8, ordre)$	$w_8 = (-1,0),(1,2),(1,1),(1,1),(2,2)$

Le mot w calculé nous permet de déduire le schéma de dépendances suivant entre acc et ordre $ordre[1] \to acc[1]$, $ordre[2] \to acc[1]$, $ordre[3] \to acc[2]$, $ordre[4] \to acc[3]$ $ordre[5] \to acc[5]$ $ordre[6] \to acc[5]$. Il nous reste maintenant à appliquer la formule de la latence définie en section 3 pour obtenir la latence de la chaîne. On a :

$$Lat(L_2) = [0, 60[$$

4 Conclusion et perspectives

L'objectif du travail présenté dans cet article est double : d'une part proposer une nouvelle formalisation de la notion de latence d'une spécification fonctionnelle (un assemblage de nœuds et de flots étiquetés par des horloges périodiques), et d'autre part proposer une méthode qui calcule la latence d'une chaîne dans un système synchrone indépendamment de la façon dont le système sera implanté et ordonnancé. La méthode proposée repose sur les dépendances de données entre les variables de la chaîne. Le calcul a été implanté en OCAML et intégré à la chaîne de compilation PRELUDE.Ce travail ouvre ainsi plusieurs perspectives.

Notion de fraîcheur. Cet article s'est concentré sur la notion de latence d'une chaîne fonctionnelle. D'autres types de propriétés temporelles peuvent également être étudiée, en particulier pour les systèmes de contrôle-commande. Reprenons la figure 7, et intéressons nous maintenant au rafraichissement du flot `ordre` en sortie de la chaîne. La figure montre que, bien que la période de `ordre` soit de 30ms, la durée maximale possible entre deux mises à jour du flot est 90ms. Cette situation survient lorsque `ordre`[1] est produit très rapidement au début de sa période, lorsque `ordre`[3] est produit presque à la fin de sa période, `ordre`[2] n'ayant pas été rafraîchie par rapport à `ordre`[1]. On voit donc que la notion de fraîcheur est différente de cette de période ou de celle de latence. Une troisième piste d'investigation portera sur la formalisation de ce type de propriété, et la définition d'un calcul analogue.

Prendre en compte des nœuds importés non synchrones. Enfin, nous avons supposé dans cet article que les nœuds importés au sein d'un assemblage sont synchrones au sens où ils n'introduisent pas de latence

(hormis leur temps d'exécution propre). Dans une approche de spécification hiérarchique, cette hypothèse devra être abrogée, un nœud importé pouvant devenir lui-même un assemblage complexe. Une quatrième piste d'amélioration consistera à introduire cette dimension hiérarchique dans le calcul de latence.

Références

[1] Absint. *aiT Worst-Case Execution Time Aanalyzers.*

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[3] François Carcenac. *Un modèle d'abstraction pour la vérification des systèmes embarqués distribués : application à l'avionique.* Thèse de doctorat, SupAéro, décembre 2005.

[4] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, USA, October 2003.

[5] Antoine Colin, Isabelle Puaut, Christine Rochange, and Pascal Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Technique et Science Informatiques*, 22(5) :651–677, 2003.

[6] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures : the Prelude-SchedMCore toolset. In *19th International Conference on Real-Time and Network Systems*, Nantes, France, September 2011. Irccyn.

[7] Adrian Curic. *Implementing Lustre programs on distributed platforms with real-time constraints.* PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[8] Francois-Xavier Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference (ERTS'2008)*, 2008.

[9] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL) : an introduction. Technical report, Carnegie Mellon University, 2006.

[10] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, December 2008.

[11] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. Implementing multi-periodic critical systems : from design to code generation. In Manuela L. Bujorianu and Michael Fisher, editors, *FMA*, volume 20 of *EPTCS*, pages 34–48, 2009.

[12] Object Management Group. A UML profile for MARTE. Technical report, Object Management Group, Inc, 2007.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[14] Gilles Kahn. The semantics of simple language for parallel programming. In *International Federation for Information Processing (IFIP'74) Congress*, New York, USA, 1974.

[15] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata : a hybrid method for analyzing embedded real-time systems. In *EMSOFT '09 : Proceedings of the seventh ACM international conference on Embedded software*, pages 107–116, 2009.

[16] Michael Lauer, Frédéric Boniol, Jérôme Ermont, and Claire Pagetti. Latency and freshness analysis on IMA systems. In *Emerging Technologies and Factory Automation (ETFA), Toulouse, France.* IEEE, septembre 2011.

[17] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus : a theory of deterministic queuing systems for the internet.* Springer-Verlag, 2001.

[18] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 23–36. IEEE, 1981.

[19] S. Martin. *Maîtrise de la dimension temporelle de la Qualité de Service dans les Réseaux.* PhD thesis, Univsersité Paris XII, 2004.

[20] T.W. O'Neil, S. Tongsima, and EHM Sha. Optimal scheduling of data-flow graphs using extended retiming. In *Proceedings of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*, pages 292–297, 1999.

[21] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3) :307–338, 2011.

[22] Ahmed Rahni, Emmanuel Grolleau, and Michaël Richard. An efficient response-time analysis for real-time transactions with fixed priority assignment. *Innovations in Systems and Software Engineering.*

[23] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, pages 101–104, 2000.

[24] Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. Calcul et vérification de propriétés de latences sur une spécification fonctionnelle synchrone multi-périodique. In *Approches Formelles dans l'assistance au développement de logiciels*, pages 10–25, 2012.