Implantation multitâche de programmes synchrones multipériodiques

Frédéric Boniol[*] — Mikel Cordovilla[*] — Julien Forget[*]
David Lesens[**] — Claire Pagetti[*,*]**

[*] *ONERA, 2, avenue Edouard Belin, 31055 Toulouse*

[**] *EADS Astrium Space Transportation, route de Verneuil, 78133 Les Mureaux*

[***] *ENSEEIHT, 2, rue Charles Camichel, 31000 Toulouse*

RÉSUMÉ. *La programmation sûre des systèmes embarqués critiques est un enjeu industriel de plus en plus important. Des langages formels de haut niveau ont été définis afin de permettre la spécification de tels systèmes. Les langages synchrones, notamment, sont utilisés avec succès car ils fournissent un bon niveau d'abstraction et s'appuient sur des générateurs de code exécutable éprouvés. La capacité à générer un code multitâche préemptible, qui permettrait d'obtenir un code embarqué plus efficace, reste cependant pour l'instant limitée. Nous montrons qu'il est possible de spécifier un système multipériodique à un niveau abstrait synchrone, puis de générer automatiquement un système multitâche exécutable par un ordonnanceur dynamique tel que EDF. Ceci passe par la définition d'un langage de haut niveau, puis par sa transformation en un ensemble de tâches temps réel, et enfin par la définition d'un protocole de communication garantissant à l'exécution le respect de la sémantique du langage synchrone. On supprime ainsi toute rupture entre la spécification et l'implantation multitâche.*

ABSTRACT. *This article presents a complete scheme for the development of multi-periodic critical embedded systems. The system is programmed with a language that enables to assemble in a modular and hierarchical manner several locally mono-periodic synchronous systems into a globally multi-periodic synchronous system. A program is translated into a set of real-time tasks that can be executed on a real-time platform with a dynamic-priority scheduler such as EDF. The compilation is formally proved correct, meaning that the generated code respects the real-time semantics of the original program (respect of periods, deadlines, release dates and precedences) as well as its functional semantics (respect of variable consumption).*

MOTS-CLÉS : *temps réel, langages synchrones, multitâche préemptif, systèmes embarqués.*

KEYWORDS: *real-time, synchronous languages, preemptive multitasking, embedded systems.*

1. Introduction

Les systèmes embarqués sont des systèmes de plus en plus complexes, critiques et souvent plongés dans un milieu physique perturbateur. La conception de tels systèmes nécessite de ce fait le concours de langages de haut niveau, permettant d'automatiser certaines étapes de développement. Nous nous situons dans le contexte de systèmes multipériodiques de type contrôle-commande. Ces systèmes sont généralement composés de trois sortes d'action : l'*acquisition de données* (par exemple dans le cas d'un aéronef, les lois de commande utilisent les informations relatives à la position des gouvernes, la position et la vitesse de l'aéronef), le *traitement* (étant donné l'état observé, calcul des actions à réaliser pour atteindre l'état souhaité) et le *contrôle* (envoi des ordres aux actionneurs).

L'approche de *conception conjointe commande / ordonnancement* (Cervin, 2003; Simon *et al.*, 2006) consiste à ajuster certains paramètres en temps réel en fonction des performances courantes de la commande par le contrôleur. La prise en compte de la qualité de la commande dans les choix à l'exécution permet une meilleure réactivité et une optimisation des ressources. Dans notre cadre, nous ne remontons pas jusqu'aux spécifications de la commande mais à des spécifications dérivées de moins haut niveau. La qualité de commande a été dérivée en des contraintes sur l'échantillonnage et le comportement temps réel. Dès lors, nous partons d'un environnement simplifié et d'un ensemble d'hypothèses à satisfaire.

Les langages synchrones (Benveniste *et al.*, 2003) sont depuis plusieurs années utilisés avec succès pour concevoir ces systèmes. Néanmoins, ces langages génèrent la plupart du temps un code séquentiel, ce qui donne lieu à des implantations monotâches. Des travaux portant sur la parallélisation de programmes synchrones (Aubry *et al.*, 1996; Caspi *et al.*, 1999; Girault *et al.*, 2006) permettent de générer une implantation multitâche concurrente multiprocesseur. Cette solution peut s'appliquer pour réaliser du multitâche monoprocesseur sous réserve de définir des priorités entre tâche. La gestion des priorités n'est pas traitée dans ces papiers car elle n'est pas pertinente pour le contexte distribué. Du fait de l'apparition de nouveaux systèmes d'exploitation embarqués critiques comme OSEK (OSEK, 2003) pour l'automobile ou l'ARINC 653 (ARINC, 2005) pour l'aéronautique, il est intéressant de proposer des générations de code multitâches et de laisser au système d'exploitation la gestion de l'ordonnancement des tâches tout en assurant un fonctionnement déterministe. Malheureusement, le domaine du temps réel propose généralement des implantations multitâches efficaces tout en laissant de côté la question des communications fonctionnelles. Il est dès lors difficile, voire impossible, de vérifier le système implanté et d'assurer le *déterminisme fonctionnel*. La description de processus temps réel requiert de ce fait l'utilisation d'un langage permettant la description des communications comme les langages de description d'architecture AADL (Feiler *et al.*, 2006) ou Giotto (Henzinger *et al.*, 2003).

L'objectif de cet article est, à partir d'un langage de description d'assemblage multipériodique de fonctions reposant sur une sémantique synchrone, de générer automatiquement un ensemble de tâches temps réel et des mécanismes de buffers associés

permettant de conserver la sémantique du langage initial tout en exécutant le code avec un ordonnanceur dynamique. Le langage choisi permet de décrire des schémas de communication plus flexibles et plus variés.

Motivation Afin d'illustrer notre propos, nous considérons un exemple très simple de commande de vol d'un avion. Ce système est destiné à contrôler l'attitude, la trajectoire et la vitesse d'un avion en mode de pilotage automatique ou manuel. Ce système 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 actionneurs ou servocommandes (permettant de positionner 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. On constate que les tâches communiquent entre elles et que l'ordre des traitements peut être important. Il existe donc des précédences entre les tâches qui sont dérivées notamment d'exigences de rafraîchissement des variables.

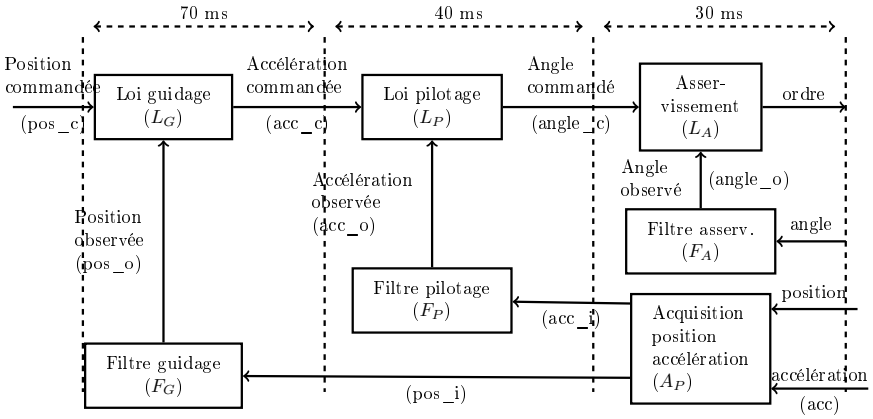

Figure 1. *Exemple de système multipériodique*

Pour un système réel de commandes de vol, il peut y avoir plus de 5000 fonctions à assembler (et non 7 comme dans l'exemple simplifié) qui ont été codées par des personnes différentes, voire des équipes différentes, dans des sociétés différentes. L'intégrateur système a alors en charge l'assemblage des fonctions : il doit respecter des contraintes globales et pouvoir vérifier le système réellement implanté. Le langage d'assemblage défini dans (Forget *et al.*, 2008) permet de décrire les interactions synchrones entre des fonctions importées écrites dans un autre langage comme du C ou du LUSTRE (Halbwachs *et al.*, 1991) par exemple. La force du langage est de décrire précisément la communication de données entre tâches s'exécutant à des périodes différentes. La sémantique formelle synchrone permet alors de vérifier des propriétés fonctionnelles et temporelles qui seront conservées à l'exécution. Afin de profiter des

avantages d'un ordonnanceur temps réel, le code généré par le compilateur n'est pas séquentiel - il aurait alors suffi de traduire un assemblage en un programme LUSTRE- mais est destiné à une implantation multitâche.

Compilation pour une implantation multitâche Pour un système donné (i.e., un assemblage décrit dans le langage), les nœuds importés du système sont assimilés à des tâches temps réel. Les échanges sont décrits par un graphe de communication entre les tâches et leur environnement : ces communications impliquent d'une part une relation de précédence entre tâches et d'autre part des schémas de consommation de données bien précis. Ces schémas sont représentés par des *mots de dépendance périodiques*. Ces mots sont une structure de données compacte contenant toutes les informations de dépendance de données et par conséquent de précédence. Pour résoudre la question des précédences dans le cas de la politique EDF, nous appliquons l'algorithme de (Chetto *et al.*, 1990). Pour respecter la sémantique des consommations de données, nous proposons l'utilisation d'un mécanisme de buffers permettant de faire communiquer les tâches qui assure les mêmes résultats fonctionnels à l'exécution pour une suite d'entrées donnée quelques soient les choix dynamiques de l'ordonnanceur.

Travaux connexes (Sofronis *et al.*, 2006) se sont déjà intéressés à une question similaire et ont défini un protocole de communication appelé *Dynamic Buffering Protocol* (DBP) permettant d'ordonnancer des tâches exprimées en LUSTRE par des ordonnanceurs de type RM ou EDF. Notre contexte est moins large que celui défini par Sofronis et al. puisque nous ne considérons que des systèmes multipériodiques. Pour ces systèmes les auteurs proposent un algorithme de calcul du nombre optimal de buffers de communication sans détailler ensuite les algorithmes d'écriture et de lecture. Nous proposons d'adapter leur algorithme aux mots de dépendance périodiques de façon à trouver le nombre optimal de buffers et à générer le code de gestion des buffers des écrivains et lecteurs. Le protocole DBP défini par Sofronis et al. est un protocole générique, applicable notamment à des tâches apériodiques, qui décrit les accès des tâches aux buffers par l'intermédiaire des pointeurs. Certains de ces pointeurs doivent être modifiés à la date de réveil d'une instance de tâche. Notre contexte multipériodique étant plus restreint, nous connaissons statiquement les schémas d'accès aux buffers. Notre approche consiste à empaqueter le code des tâches de façon à calculer le numéro des cases du buffer à accéder.

Notons que Sofronis et al., même s'ils posent une question similaire à la nôtre consistant à porter un code LUSTRE sur un ordonnanceur EDF ou RM en préservant la sémantique synchrone, passent sous silence le calcul effectif, à partir d'un programme de haut niveau, des tâches à ordonnancer, des caractéristiques temps réel de ces tâches, en particulier des priorités, et des dépendances entre les tâches générées. Ils considèrent que ce problème est traité en amont. (Alras *et al.*, 2009) se basent sur ce protocole pour compiler une extension de LUSTRE avec des primitives temps réel en une implantation multitâche. Dans notre contexte, le système est spécifié à l'aide du langage d'assemblage qui décrit les relations de fréquences entre nœuds importés

qui sont assimilés à des tâches temps réel. Le passage de cette description synchrone (le niveau fonctionnel) en un ensemble multitâches multipériodiques avec précédence (le modèle de tâches concret exécuté par l'ordonnanceur) est alors automatique.

L'outil Simulink (Mat, 2009) propose des opérateurs de changement de période relativement proches de ceux du langage d'assemblage. Il est possible de générer, à partir de l'outil Real-time Workshop, du code pour des ordonnanceurs de type rate monotonic mais les systèmes doivent être harmoniques, c'est-à-dire que les périodes sont multiples les unes des autres (ce qui est plus restrictif que multipériodique), les échéances doivent être égales aux périodes (ce qui peut être trop contraignant) et les communications se font par sémaphore (ce qui alourdit la vérification et la validation de l'implantation).

Plan Dans la partie 2, nous présentons succinctement le langage d'assemblage choisi comme langage de spécification. Nous montrons ce qui est obtenu automatiquement à partir d'une telle description, à savoir un ensemble de tâches temps réel contraintes par des relations de dépendance de données. Dans la partie 3, nous détaillons l'approche choisie pour implanter ces tâches dans un ordonnanceur EDF. Cela se fait en trois étapes : le codage des précédences dans les échéances, le calcul du nombre de buffers nécessaires aux communications inter-tâches et le code associé à chaque tâche pour communiquer via ces buffers.

2. Spécification du système

On suppose que le système est spécifié à l'aide du langage d'assemblage défini dans (Forget *et al.*, 2008). La syntaxe et la sémantique du langage sont proches de LUSTRE mais sont réduites à la possibilité de combiner des nœuds importés. Deux nouveaux opérateurs $*\hat{}$ et $/\hat{}$ ont été introduits de manière à définir des accélérations et des ralentissements de flots[1]. Dans cette section, nous présentons succinctement le langage en montrant une manière de programmer l'exemple défini dans la figure 1. Nous montrons ensuite les tâches temps réel obtenues et la relation de précédence induite entre ces tâches.

2.1. *Présentation du langage d'assemblage*

LUSTRE (Halbwachs *et al.*, 1991) est un langage synchrone flot de données développé à Verimag. Pour trouver une description complète et détaillée du langage, le lecteur pourra se référer à (Halbwachs, 1993). De manière informelle, un programme LUSTRE exprime des relations entre les variables d'entrée et de sortie, chaque variable

1. Les termes "accélération" et "ralentissement" sont en réalité des abus de langage. Il s'agit plus formellement de "sur-échantillonnage" et "sous-échantillonnage". Par accélération il faut entendre "rendre plus fréquent" et par ralentissement "rendre moins fréquent".

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*. L'*horloge* d'un flot indique si le flot est présent ou non sur l'horloge de base, celle qui rythme tous les instants. Un flot a pour horloge l'horloge de base du système s'il est toujours présent.

Le langage d'assemblage défini dans (Forget *et al.*, 2008) reprend les hypothèses de LUSTRE. Un programme manipule des flots qui sont construits à partir de constantes, de variables, de tuples de valeurs, d'appel de nœuds importés sur des flots et de l'application de trois opérateurs principaux :

1) `fby` : est l'opérateur défini en SCADE (version industrielle du langage LUSTRE). Cet opérateur permet d'initialiser un flot par une constante et de prendre une valeur retardée d'un instant d'un flot ;

2) $*\hat{}$: est un opérateur d'accélération de rythme. Appliqué à un flot, cet opérateur permet de construire un flot qui va k fois plus rapidement (i.e. k fois plus fréquent) ;

3) $/\hat{}$: est un opérateur de décélération de rythme. Appliqué à un flot, cet opérateur permet de construire un flot qui va k fois plus lentement (i.e., k fois moins fréquent).

L'horloge de base est exprimée sur une échelle de temps donnée sur les entiers naturels. Les variables manipulées par le langage sont des flots dont les horloges sont *strictement périodiques*. Une horloge est strictement périodique si l'intervalle entre deux activations successives est constant. Ainsi, si $h[i]$ représente la date de la ième valeur de l'horloge h, alors pour tout i, $h[i+1]-h[i]$ vaut toujours la même constante entière (la période de l'horloge). Dans la suite, on supposera que toutes les *horloges strictement périodiques* commencent en même temps à l'instant 0. On peut donc représenter une *horloge strictement périodique* par sa période n. On notera (n) avec $n \in \mathbb{N}$ une telle horloge. Par exemple, l'horloge la plus rapide est (1) car elle est présente à tout instant. De même (2) représente l'horloge activée tous les 2 unités de temps, et qui a donc 2 comme période.

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

On constate que les opérateurs ont des actions directes sur les horloges : `fby` ne modifie pas l'horloge, $*\hat{}\ k$ multiplie la fréquence de l'horloge par k (et donc divise d'autant sa période) et $/\hat{}\ k$ divise sa fréquence par k (et donc multiplie d'autant sa période). Ces bonnes propriétés permettent de calculer statiquement les horloges des flots et par suite les périodes des tâches.

Codage de l'exemple dans le langage d'assemblage Dans l'exemple des commandes de vol simplifiées de la figure 1, les nœuds importés sont les fonctions décrites

dans les boîtes comme la loi de guidage ou la loi de pilotage. Les noms des nœuds sont ceux apparaissant entre parenthèses. Ainsi, la spécification de l'assemblage commence par la liste des nœuds importés.

```
imported node FA(i: int) returns (o: int); wcet FA=5;
imported node LA(i1,i2: int) returns (o: int); wcet LA=5;
...
```

On spécifie pour chaque nœud le type des entrées et des sorties. Ainsi, le nœud F_A prend une entrée entière et renvoie une sortie entière. On suppose que les nœuds importés sont synchrones et polymorphes du point de vue des horloges. Ainsi, F_A prend une entrée dont l'horloge est de type polymorphe α (c'est-à-dire qu'elle sera de la forme (k) avec $k \in \mathbb{N}$) et renvoie une sortie dont l'horloge est de même type α. On précise également le wcet (worst case execution time ou pire temps d'exécution) de chaque nœud.

On décrit ensuite les lois régissant l'assemblage. Plusieurs découpages modulaires sont possibles et dans l'exemple ci-dessous, on a choisi d'écrire un nœud unique pour programmer le système.

```
node cdv (angle, acc, pos; pos_c: rate (70))
returns (ordre)
var acc_c, angle_c, pos_i, acc_i;
let
        (pos_i, acc_i) = AP(pos, acc);
        ordre = LA(FA(angle),  (angle_c*^4)/^3);
        angle_c = LP (FP(acc_i*^3/^4),((0 fby acc_c)*^7)/^4);
        acc_c = LG(FG(pos_i*^3/^7),pos_c);
tel
```

La syntaxe est semblable à du LUSTRE. Un nœud est décrit par son nom, les variables d'entrée - ici, il y en a 4 et en particulier `pos_c` dont le rythme est imposé à 70 - et les variables de sortie - ici `ordre` est l'unique commande. Une variable interne peut être utilisée pour un calcul intermédiaire - c'est notamment le cas pour `pos_i`. Après le mot clé `let`, l'ensemble des équations est déclaré. La première équation calcule des variables au rythme le plus rapide et ne décrit que des connexions simples. L'ordre imposé par les lois d'asservissement utilise la donnée générée par *FA* et la donnée `angle_c` produite à 40ms. Pour être utilisable par un nœud à 30, il faut resynchroniser cette donnée. Le choix fait ici est d'accélérer par 4 puis de ralentir par 3. De la même manière, la variable `acc_c` est produite toutes les 70 et est consommée à rythme de 40. Le choix est de prendre les valeurs avec délai `0 fby acc_c` puis d'accélérer par 7 et de ralentir par 4. Cette donnée est moins critique qu'`angle_c` et peut être utilisée avec une valeur plus ancienne.

2.2. *Premières sorties du compilateur*

Le compilateur analyse les programmes pour assurer la correction d'un point de vue typage de données et des horloges. L'idée est de vérifier statiquement qu'aucune

équation n'essaie de combiner deux flots qui ne sont pas présents aux mêmes instants. Si les analyses statiques assurent la correction du programme, le compilateur extrait l'ensemble de tâches temps réel dont les périodes sont calculées automatiquement par le calcul d'horloge décrit dans (Forget *et al.*, 2008). Il faut également faire le lien entre l'horloge globale sur $\mathbb{N}$ définie dans le langage et le temps physique du système. Dans notre cas, l'unité de temps globale correspond à 1 ms. On constate qu'aucune tâche ne s'exécute à ce rythme.

Le programme - un système d'équations - est transformé en un *graphe de précédences étendues* qui contient toutes les informations de manière condensée. Le graphe généré (V, E) est un ensemble de nœuds V correspondant aux fonctions importées ainsi qu'aux variables d'entrée et de sortie du système global. Les transitions $E \subseteq V \times V$ sont étiquetées par des listes d'opérateurs appliqués entre deux tâches. Le graphe de précédences étendues pour les commandes de vol est donné dans la figure 2. On reconnaît la structure de l'exemple et le choix de communication du concepteur pour des fonctions n'ayant pas la même période.

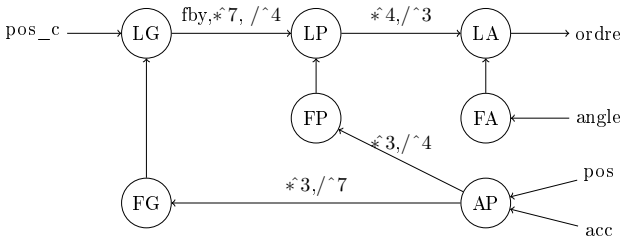

Figure 2. *Graphe de précédences étendues des commandes de vol*

2.3. *Mots de dépendance de données*

Nous considérons un modèle de tâches très simple issu des travaux de (Liu *et al.*, 1973). Une tâche τ est définie par sa période (T), sa durée d'exécution maximale (C) et son échéance relative (D), i.e. τ=(T,C,D). Comme le langage ne fait qu'accélérer ou réduire les rythmes, on n'agit pas sur les dates de réveil initiales et on est assuré qu'elles sont toutes à 0. On note $\tau[j]$ la j-ième instance de τ. L'échéance absolue, notée d, de l'instance $\tau[j]$, avec $j \in \mathbb{N}^+$, est alors $(j-1)T + D$.

La sémantique du langage d'assemblage implique un ordre partiel dans l'exécution des tâches ainsi qu'un schéma précis des consommations de données. En effet, si un nœud importé A est appliqué sur un flot résultat d'un autre nœud importé B alors le nœud B doit s'exécuter avant le nœud A. Il y a deux types de consommation : *directes* lorsqu'il s'agit d'un enchaînement d'opérateurs $*\hat{}\,k$ et $/\hat{}\,k$; *indirectes* lorsqu'il y a des `fby` . Le langage permet n'importe quelle combinaison, mais nous supposerons dans la suite que les délais ne peuvent être appliqués qu'au début, c'est-à-dire la

combinaison ne peut être que de la forme $\texttt{fby}^{\,k}$, combinaison de $*\hat{}\,k$ et $/\hat{}\,k$. Nous utilisons les *mots de dépendance de données* pour représenter les informations.

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

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

avec $d_0 \in \mathbb{N}$, $i,d \in \mathbb{N}^+$. Un mot $w = (-1,d_0)(i_1,d_1)(i_2,d_2)\dots(i_n,d_n)$ représente la communication entre deux tâches $\tau_1 \rightarrow \tau_2$ extraite du graphe de précédences étendues. Les deux premières lettres $(-1,d_0)(i_1,d_1)$ correspondent au préfixe du mot à savoir le nombre d_0 de valeurs d'initialisation consommées à cause des délais et la première instance i_1 à être réellement consommée. Le reste du mot est le motif périodique de consommation se répétant à l'infini. La signification est la suivante :

init	*est consommé par*	$\tau_2[1],\ \tau_2[2]\dots\tau_2[d_0]$
$\tau_1[i_1]$	*est consommé par*	$\tau_2[d_0+1],\ \tau_2[d_0+2]\dots\tau_2[d_0+d_1]$
$\tau_1[i_2+i_1]$	*est consommé par*	$\tau_2[d_0+d_1+1],\dots,\tau_2[d_1+d_2]$
$\dots$		
$\tau_1[\Sigma_{j<=n}i_j]$	*est consommé par*	$\tau_2[\Sigma_{i<=n-1}d_i+1]...\tau_2[\Sigma_{i<=n}d_i]$
$\tau_1[\Sigma_{j<=n}i_j+i_1]$	*est consommé par*	$\tau_2[\Sigma_{i<=n}d_i+1]...\tau_2[\Sigma_{i<=n}d_i+d1]$
$\dots$		

Dans le cas d'une communication simple où il n'y a pas de changement de rythme, comme par exemple FA → LA, le mot de dépendance est $(-1,0)(1,1)(1,1)$. En effet, $\tau_1[i]$ est consommé par $\tau_2[i]$ pour chaque instance de tâche. Les mots sont utiles pour les combinaisons des trois opérateurs du langage.

	0	1	2	3	4	5	6	7	8	9	mot
τ_1	x_1				x_2				x_3		(-1,0)(1,1)(1,1)
$\rightarrow^{\texttt{fby}}$	i				x_1				x_2		(-1,1)(1,1)(1,1)
$\rightarrow^{\texttt{fby},*\hat{}\,4}$	i	i	i	i	x_1	x_1	x_1	x_1	x_2	x_2	(-1,4)(1,4)(1,4)
$\rightarrow^{\texttt{fby},*\hat{}\,4,/\hat{}\,3}$	i			i			x_1			x_2	(-1,2)(1, 1)(1, 1)(1, 2)(1, 1)

Proposition 1 *Quand on applique une suite d'opérateurs oprs $= \left(\texttt{fby}^{\,p_0}, \text{combinaison } (*\hat{}\,k_i, /\hat{}\,k_i)\right)$ à $(-1,0)(1,1)(1,1)$, on obtient un mot de dépendance dont la longueur est inférieure à $\prod_{/\hat{}\,k_i} k_i + 2$.*

Preuve 1 *Soit un mot $w = (-1,d_0)(i_1,d_1)(i_2,d_2)...(i_n,d_n)$, On note la longueur $l(w) = n-1$ (on ne prend pas en compte le préfixe de longueur constante 2) et $nd(w) = \Sigma_{2\leq i\leq n} d_i$. On fait la preuve par induction sur la liste d'opérateurs oprs. Initialement, oprs= [], alors le mot est $(-1,0)(1,1)(1,1)$ avec $l(w) = 1 \leq \prod_{\emptyset} k_i = 1$ et $nd(w) = 1$. Supposons que le mot associé à oprs est $w = (-1,d_0)(i_1,d_1)(i_2,d_2)...(i_n,d_n)$. On cherche le mot w' associé oprs,op.*

1) si on applique `fby` *, le mot est nécessairement de la forme* $w = (-1, d_0)(1,1)(1,1)$ *puisqu'on a fait l'hypothèse de n'appliquer* `fby` *qu'au début. Alors* $w' =$ `fby` $(w) = (-1, d_0{+}1)(1,1)(1,1)$ *avec* $l(w') = l(w) = 1$ *et* $nd(w') = 1$*. Donc si oprs est de la forme* `fby` $^{p_0}, f(*\hat{}\ k_i, /\ \hat{}\ k_i)$*, après application de tous les* `fby` *, on obtient* $w = (-1, p_0)(1,1)(1,1)$ *;*

2) si on applique $*\hat{}\ k$*, alors* $w' = *\hat{}\ k(w) = (-1, k \times d_0)(i_1, k \times d_1)(i_2, k \times d_2)...(i_n, k \times d_n)$*. Donc* $l(w') = l(w) \leq \prod_i k_i$ *par hypothèse d'induction. On a de plus* $nd(w') = nd(w) \times k$ *;*

3) si on applique $/\hat{}\ k$*, le calcul de* w' *est un peu plus compliqué. Il faut parcourir* w *en enlevant* $k-1$ *consommateurs tous les* k*. On diminue les* d_j *mais il se peut qu'on fasse disparaître des instances consommées. Il faut d'abord calculer le nouveau préfixe* $(-1, \lceil d_0/k \rceil)(i'_1, d'_1)$*. Si* d_0 *mod* $k = 0$ *alors* $i'_1 = i_1$ *et* $d'_1 = \lceil d_1 \rceil$*. Sinon* $i'_1 = \Sigma_{j \leq p} i_j$ *avec* $p =$ *min*$_j\{d_0$ *mod* $k + \Sigma d_j \geq k\}$ *et* $d'_1 = \lceil (d_0 + \Sigma_{j \leq p} i_j - k)/k \rceil$*.*

Il faut ensuite dupliquer le sous-mot $(i_{p+1}, d_{p+1}) \ldots (i_{p+l(w)-1}, d_{p+l(w)-1})$ *r fois en le mot* w_i *de sorte que* $nd(w_i)$ *soit divisible par* k*. Il faut en effet prendre une instance du consommateur tous les* k*. On obtient alors le mot* w' *dont la longueur* $l(w') \leq l(w_i)$ *et* $nd(w') = nd(w_i)/k$*. Or la longueur* $l(w_i) =$ *ppcm*$(nd(w), k)/nd(w) \times l(w) \leq k \times l(w) \leq k \times \prod_i k_i$*. Pour calculer le mot* w*, il faut faire un raisonnement similaire au calcul du préfixe : on avance dans le mot* w_i *tant que* $\Sigma d_j \leq k$ *et on ne garde que les instances où cette somme est* $> k$*. On a alors* $i' = i_{prec} + \Sigma_j i_j$ *et* $d' = \lceil (d + \Sigma_j d_j + (d_{prec}$ *mod* $k) - k) \rceil$ *et on reparcourt la suite du mot en remettant* Σd_j *et* Σi_j *à 0. Il faut également remarquer que* $d_n = d_1$*, répétition de la première instance. Ce qui assure la périodicité du calcul.*

Cette représentation par mots de dépendance est équivalente au graphe de précédences étendues et permet d'exprimer de façon factorisée les dépendances de données. Cela nous permettra de mettre en commun des emplacements mémoires pour des tâches consommatrices d'une même donnée.

3. Implantation multitâche du système

Nous présentons dans cette partie l'implantation multitâche d'un programme réalisé avec le langage d'assemblage. Les tâches temps réel sont simplement les fonctions importées dans l'assemblage. Concernant les caractéristiques temps réel : le wcet est spécifié dans le programme lors de la déclaration d'importation du nœud et la période est automatiquement calculée par le compilateur du langage à partir des périodes des entrées sorties. La date de réveil est toujours nulle.

L'objectif est alors de proposer une implantation de ces tâches avec un ordonnanceur dynamique, comme EDF, qui respecte la sémantique de l'assemblage. Ceci requiert deux étapes :

1) assurer le respect des précédences : soit l'ordonnanceur dynamique est directement capable de les traiter, soit il faut utiliser un encodage. Pour un ordonnanceur

de type EDF par exemple, (Chetto *et al.*, 1990) ont défini un algorithme qui transforme des tâches dépendantes avec précédence en tâches indépendantes en modifiant les dates de réveil et les échéances ;

2) optimiser le nombre de buffers pour les communications : une exécution multitâche ne respectera pas nécessairement la sémantique car des entrelacements variant d'une hyperpériode à l'autre sont possibles. A partir des mots de dépendance, on peut néanmoins en déduire un protocole de communication statique périodique qui assure le déterminisme fonctionnel de l'implantation multitâche. Nous montrons comment calculer un nombre optimal de buffers et comment en déduire un code statique d'empaquetage des tâches décrivant les instances et les buffers où chaque tâche doit lire et écrire ses données.

3.1. *Encodage des précédences*

(Chetto *et al.*, 1990) ont proposé un algorithme pour transformer un ensemble de tâches apériodiques avec précédence en un ensemble de tâches indépendantes en modifiant les dates de réveil et les échéances. Notre contexte est légèrement plus simple puisque nous considérons des tâches multipériodiques de dates de réveil nulles. L'algorithme initial modifie les dates de réveil et échéances absolues. Comme toutes les dates de réveil sont à zéro, il n'y a pas lieu de les modifier. L'algorithme agit pour toute tâche τ sur son échéance absolue de façon à calculer $d^\star = \mathit{min}_{j \mid \tau \prec \tau_j}(d, d_j^\star - C_j)$. Dans notre modèle nous raisonnons sur des échéances relatives, on adapte donc la formule en modifiant simplement les échéances relatives. Soit $\tau_1 \rightarrow^{oprs} \tau_2$, on calcule alors le mot de dépendance associé à cette communication $w = (-1, d_0)(i_1, d_1)\ldots(i_n, d_n)$. On en déduit toutes les contraintes sur les échéances de τ_1. L'instance $\tau_1[i_k]$ doit se terminer avant le début de la première instance de τ_2 consommatrice des données produites par $\tau_1[i_k]$ à savoir $\tau_2[\Sigma_{\leq k-1} d_j + 1]$ (par définition du mot de dépendance). Cela donne l'ensemble des contraintes absolues pour $k = 1, \ldots, n$, $d^\star(\tau_1[i_k]) \leq d(\tau_2^\star[\Sigma_{j \leq k-1} d_j + 1]) - C(\tau_2)$. D'où en contrainte relative,

$$\begin{aligned} D^\star(\tau_1) \quad \leq \mathit{min}_{k \leq n} \big\{ \quad & D^\star(\tau_2) + (\Sigma_{j \leq k-1} d_j + 1) T(\tau_2) \\ & - \mathit{max}(0, (\Sigma_{j \leq k} i_j - 1)) T(\tau_1) - C(\tau_2) \big\} \end{aligned}$$

On peut ainsi en déduire l'échéance relative de τ_1 comme la plus grande échéance respectant toutes les contraintes imposées par ses consommateurs. On est assuré d'après l'hypothèse sur les délais (`fby` en début de la liste des opérateurs) qu'il n'y a pas de boucle dans les contraintes.

Le modèle de tâches de l'exemple simplifié des commandes de vol devient donc $L_A = (30, 5, 30)$, $F_A = (30, 5, 25)$, $A_P = (30, 5, 15)$, $F_P = (40, 5, 20)$, $L_P = (40, 5, 25)$, $F_G = (70, 7, 63)$ et $L_G = (70, 7, 70)$. Par une analyse classique d'ordonnançabilité, on peut vérifier que ce système est ordonnançable.

3.2. *Calcul du nombre optimal de buffers*

Afin d'adapter le modèle synchrone au contexte des systèmes multipériodiques, nous remplaçons l'hypothèse synchrone, selon laquelle les traitements doivent tous terminer leur exécution avant la fin de l'instant, par une *hypothèse synchrone relâchée* (introduite dans (Curic, 2005)) selon laquelle un traitement doit se terminer *avant sa prochaine activation*. L'échéance d'une instance de tâche est donc le minimum entre l'échéance imposée par cette hypothèse de synchronisme et l'échéance calculée.

Problème des entrelacements Pour conserver la sémantique synchrone, il faut que les instances consomment les données produites par les bonnes instances productrices. Du fait des différences de rythme, il se peut qu'il faille introduire des mémoires de communication supplémentaires.

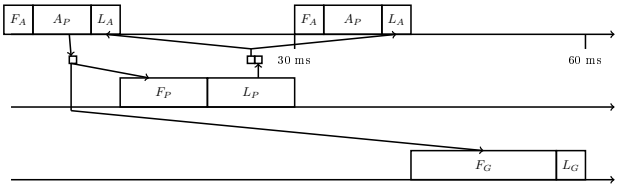

Figure 3. *Exemple d'exécution*

A titre d'illustration, les données produites par A_P, dans l'exemple des commandes de vol, sont consommées par F_P et F_G. Dans la figure 3, on constate que les données produites par la première instance doivent être conservées pendant plusieurs périodes de A_P. Soit on considère que la donnée doit être disponible pendant toute la période de F_G soit 70 ms, soit on veut optimiser et décider que la valeur doit être disponible pendant le pire temps de réponse des tâches consommatrices. Nous faisons le choix de la solution 1, de même que Sofronis et al., car l'optimisation de la solution 2 nous semble coûteuse et non nécessairement efficace. La tâche L_G communique avec un retard avec L_P, il lui faut donc deux cases : une pour stocker la valeur précédente et une pour écrire la valeur courante.

Méthode de Sofronis *et al.* Les auteurs de (Sofronis *et al.*, 2006) partent d'un ensemble de tâches temps réel obtenues à partir d'un code LUSTRE (ils n'expliquent pas comment sont dérivées ces tâches à partir du code LUSTRE). Ils identifient également deux types de précédence : directe et indirecte. Les auteurs étudient d'abord les mécanismes de buffer dans un cadre général et dans un deuxième temps, ils proposent de calculer le nombre optimal de buffers nécessaires pour des tâches multipériodiques.

L'approche est simple : pour l'ensemble des communications $\tau \rightarrow \tau_1, \ldots, \tau \rightarrow \tau_n$, on regarde sur l'hyperpériode des tâches et pour chaque instant d'écriture de τ si :

1) la donnée produite est consommée,

2) si oui, on teste s'il est possible de réutiliser une case existante du buffer de communication pour stocker cette donnée. Si aucune case ne peut être réutilisée, on ajoute 1 au nombre de cases.

Protocole de communication Nous appliquons cet algorithme sur les communications décrites dans le graphe de précédences étendues. Nous calculons le nombre de buffers mais également les moments d'écriture de τ et de lecture des τ_k. Soit $\tau \rightarrow^{oprs} \tau_1$, on calcule alors le mot de dépendance associé à cette communication $w = (-1, d_0)(i_1, d_1)\dots(i_n, d_n)$. On peut déduire d'un mot de dépendance le dernier instant de consommation d'une donnée. En effet, les données produites par l'instance $\tau[i_k]$ sont consommées par plusieurs instances de τ_1 et doivent être disponibles jusqu'à la fin de l'exécution de la dernière à savoir $\tau_1[\Sigma_{\leq k} d_j]$ (par définition du mot de dépendance). En temps absolu, cela donne si $p = max(0, (\Sigma_{j \leq k} i_j - 1))$:

$$\begin{array}{ll} \tau[i_k] & \textit{écrit dans l'intervalle } [p \times T(\tau), p \times T(\tau) + D(\tau)] \\ \tau[i_k] & \textit{disponible jusqu'à } T(\tau_1) \times (\Sigma_{j \leq k} d_j - 1) + D(\tau_1) \end{array}$$

Pour toutes les consommations de τ, $\tau \rightarrow^{oprs} \tau_j$, $j = 1, \dots, m$, on obtient m mots de dépendance $w^j = (-1, d_0^j)(i_1^j, d_1^j)\dots(i_n^j, d_n^j)$. Il faut raisonner sur l'hyperpériode de l'ensemble de ces mots, à savoir $ppcm_j(\Sigma_k i_k^j)$. On déplie alors les mots périodiques pour qu'ils soient tous de même taille (ce *ppcm*). On calcule ensuite un tableau représentant le protocole de communication entre τ et ses consommateurs de longueur *ppcm* dont chaque élément est de la forme : *(numéro de l'instance de τ consommée, numéro du buffer, liste des consommateurs et numéro des instances, date absolue de fin de consommation)*. Pour la première instance consommée, on obtient le tableau [$min_j i_1^j$, 1, liste de consommateurs, $max_{j, i_1^j = min_p i_1^p}((d_0^j + d_1^j - 1) \times T(\tau^j) + D(\tau^j))$]. Pour chaque nouvelle instance consommée i_k, on regarde si l'un des buffers p est disponible dans le tableau, si oui on rajoute dans le tableau [i_k, p, liste de consommateurs, plus grande date de consommation] sinon on ajoute un nouveau buffer [i_k, nb +1, liste de consommateurs, plus grande date de consommation].

Sur l'exemple des commandes de vol, on obtient 6 buffers. Il faut en particulier 2 cases pour les communications $AP \rightarrow^{*3,/4} FP$ et $AP \rightarrow^{*3,/7} FG$. Les mots sont respectivement $(-1, 0)(1, 1)(1, 1)(1, 1)(2, 1)$ et $(-1, 0)(1, 1)(2, 1)(2, 1)\ (3, 1)$. Le premier motif porte sur les instances 2 à 5, et le deuxième des instances 3 à 8. Le motif d'écriture de AP est basé sur les 40 premières instances. La première instance de *AP* écrit dans la première case du buffer et elle sera consommée par les premières instances de *FP* et *FG*. Cela donne à l'initialisation du tableau [(1, 1, [(FP,1) ;(FG,1)], 63)] car $D(\text{FG}) = 63$. La deuxième instance est écrite dans la deuxième case car elle débute à 30 et est consommée par la deuxième instance de *FP*. Le calcul donne alors [(1, 1, [(FP,1) ;(FG,1)], 63) ; (2, 2, [(FP,2)], 60)]. La troisième instance est consommée par les deux tâches ce qui donne [(1, 1, [(FP,1) ;(FG,1)], 63) ; (2, 2, [(FP,2)], 60) ; (3, 1, [(FP,3) ;(FG,2)], 126)] et ainsi de suite jusqu'à 40.

On en déduit aisément les mots de lecture et d'écriture qui seront empaquetés dans le code. Nous avons mis en œuvre nos algorithmes et nous les avons appliqués sur

une étude de cas réelle composée de 750 tâches, 4 périodes (10, 20, 40 et 120), 392 précédences directes et 1023 indirectes. Comme ce système était harmonique, nous sommes assurés que la longueur des mots d'écriture et de lecture est plus petite que l'hyperpériode, donc plus petite que 12.

Implantation Le protocole *Dynamic Buffering Protocol* (DBP) défini par Sofronis et al. se base sur l'utilisation de pointeurs qui sont modifiés en deux temps : d'une part au moment où la tâche se réveille et d'autre part au moment où la tâche s'exécute. Notre choix est différent du fait des patterns périodiques des tâches. On empaquette le code de façon à connaître à chaque exécution la case dans laquelle la tâche doit écrire (resp. lire pour les tâches lecteurs) ses données. Le prototype génère un code C utilisant les extensions temps réel de POSIX. L'étude de cas a été testée avec MARTE OS (Rivas *et al.*, 2002) et l'ordonnançabilité vérifiée avec Cheddar (Singhoff *et al.*, 2004).

4. Conclusion

Ce travail présente une génération de code complète d'un langage d'assemblage de fonctions multipériodiques vers un code multitâche embarquable dans un ordonnanceur EDF. Le cadre de développement proposé est plutôt dédié à un travail d'intégration de systèmes manipulant un grand nombre de fonctions développées à part. Cette première phase assure la faisabilité de l'approche et la possibilité d'implanter le système avec un ordonnanceur embarqué sans avoir recours à un séquencement manuel. En faisant l'étude de cas réelle, on constate que le grain de la *fonction importée* pour les tâches temps réel n'est pas forcément le bon. Dans la suite, nous allons étudier des moyens de regroupement de fonctions importées de façon à réduire le nombre de tâches et donc le nombre de préemptions.

Par ailleurs le langage d'assemblage présenté ici a été simplifié car nous n'avons pas utilisé l'opérateur de décalage de phase $\sim>$. Cet opérateur va introduire des dates de réveil non toutes nulles : nous devrons adapter le calcul du nombre de buffers à ce nouveau contexte, ainsi que l'empaquetage des tâches.

Enfin pour une meilleure convivialité nous pourrons nous intéresser au développement d'un éditeur graphique en utilisant un outil comme TopCased.

5. Bibliographie

Alras M., Caspi P., Girault A., Raymond P., « Model-based design of embedded control systems by means of a synchronous intermediate model », *International Conference on Embedded Software and Systems (ICESS'09)*, Hangzhou, China, May, 2009.

ARINC, *ARINC Specification 653 : Avionics Application Software Standard Interface*, Aeronautical Radio INC. 2005.

Aubry P., Le Guernic P., Machard S., « Synchronous distribution of Signal programs », *29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1 : Software Technology and Architecture*, 1996.

Benveniste A., Caspi P., Edwards S. A., Halbwachs N., Le Guernic P., de Simone R., « The synchronous languages 12 years later », *Proc. IEEE*, 2003.

Caspi P., Mazuet C., Salem R., Weber D., « Formal design of distributed control systems with Lustre », *18th International Conference on Computer Computer Safety, Reliability and Security (SAFECOMP'99)*, Toulouse, France, September, 1999.

Cervin A., Integrated control and real-time scheduling, PhD thesis, Department of Automatic Control, Lund University, Sweden, April, 2003.

Chetto H., Silly M., Bouchentouf T., « Dynamic scheduling of real-time tasks under precedence constraints », *Real-Time Systems*, 1990.

Curic A., Implementing Lustre programs on distributed platforms with real-time constraints, PhD thesis, Université Joseph Fourier, Grenoble, 2005.

Feiler P. H., Gluch D. P., Hudak J. J., The Architecture Analysis & Design Language (AADL) : an introduction, Technical report, Carnegie Mellon University, 2006.

Forget J., Boniol F., Lesens D., Pagetti C., « A multi-periodic synchronous data-flow language », *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, December, 2008.

Girault A., Nicollin X., Pouzet M., « Automatic rate desynchronization of embedded reactive programs », *ACM Trans. Embedded Comput. Syst.*, vol. 5, n° 3, p. 687-717, 2006.

Halbwachs N., *Synchronous programming of reactive systems*, Kluwer Academic Publisher, 1993.

Halbwachs N., Caspi P., Raymond P., Pilaud D., « The synchronous data-flow programming language Lustre », *Proc. IEEE*, 1991.

Henzinger T. A., Horowitz B., Kirsch C. M., « Giotto : A time-triggered language for embedded programming », *Proc. IEEE*, 2003.

Liu C. L., Layland J. W., « Scheduling algorithms for multiprogramming in a hard-real-time environment », *Journal of the ACM*, 1973.

Mat, *Simulink : User's Guide*. 2009.

OSEK, *OSEX/VDX Operating System Specification 2.2.1*, OSEK Group. 2003, www.osek-vdx.org.

Rivas M. A., Harbour M. G., « POSIX-compatible application-defined scheduling in MaRTE OS », *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, Washington, USA, 2002.

Simon D., Sename O., Robert D., « Conception conjointe commande/ordonnancement et ordonnancement régulé », *in* N. Navet (ed.), *Systèmes temps-réel 2 : ordonnancement, réseaux, qualité de service*, Traité IC2, Lavoisier - Hermès, p. 81-108, 2006. CO - NECS.

Singhoff F., Legrand J., Nana L., Marcé L., « Cheddar : a flexible real time scheduling framework », *Ada Lett.*, 2004.

Sofronis C., Tripakis S., Caspi P., « A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling », *Sixth International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October, 2006.