Aide à la conception multi points de vue de systèmes embarqués

François Revest[1], Frédéric Boniol[2], Claire Pagetti[1]
[1] ONERA-CERT - 2 av. E. Belin 31055 Toulouse
[2] ENSEEIHT - 2 rue C. Camichel 31071 Toulouse
{revest,pagetti}@cert.fr, Frederic.Boniol@enseeiht.fr

11 avril 2007

Résumé

La conception de systèmes complexes et critiques, tels que les systèmes avioniques, est un travail fédérateur et collaboratif au sein d'une organisation industrielle nécessitant de nombreuses compétences. La plupart des fonctionnalités sont aujourd'hui supportées par des calculateurs et des réseaux de type Ethernet.

L'objectif de ce papier est d'aider l'intégrateur à avoir une vision globale du système développé selon des points de vue essentiels pour la validation et la certification. Les premières briques d'une méthodologie générale sont posées et quelques réflexions sur une vision exécutive temps réel sont présentées.

Concrètement, le point de vue fonctionnel est décrit par un ensemble de nœuds LUSTRE multi périodiques. Une réflexion est menée autour de la sémantique de cet ensemble et de l'impact sur l'exécution réelle. La partie matérielle est donnée sous forme d'une description AADL. A partir des ces données, plusieurs ordonnancements de type EDF respectant les contraintes fonctionnelles sont générés puis testés par l'outil Cheddar afin d'en vérifier la correction temps réel. Cette méthode sera illustrée sur un exemple simplifié des commandes de vol.

1 Introduction

Contexte Les systèmes embarqués sont des systèmes de plus en plus complexes, critiques, désormais à logiciels prépondérants, et souvent plongés dans un milieu physique perturbateur. La conception de tels systèmes nécessite de ce fait le concours de plusieurs équipes spécialisées dans des domaines différents : la sûreté de fonctionnement, la conception de plateforme d'exécution et de communication temps réel…. Compte-tenu de la complexité globale induite, toutes ces équipes ne peuvent avoir chacune qu'un "point de vue" partiel du système qu'elles doivent pourtant concourir à spécifier, réaliser, et tester. La conséquence directe en est une plus grande difficulté du processus d'intégration, de conception et de validation.

L'objectif de cet article est de définir les premières briques d'une méthodologie permettant d'avoir les informations globales nécessaires à la validation et à la certification d'un système de type contrôle-commande dans un contexte multi vues.

Cadre conceptuel multi vues La formalisation d'un cadre conceptuel multi points de vue clair et formel a été développé dans [The06]. L'idée développée par les auteurs est de structurer l'organisation de conception entre équipes différentes tout en permettant à l'intégrateur de conserver des propriétés globales de performances temps réel, de sûreté de fonctionnement ... Nous reprenons dans le papier les notions formelles suivantes : une *facette* correspond au point de vue des équipes spécialisées et un *layer* décrit une préoccupation transverse permettant la vérification de propriétés globales.

Vision exécutive globale Notre but est de mettre en œuvre la théorie de [The06] sur la vue d'intégration globale et d'illustrer son avantage sur l'aspect exécutif.

Les points de vue considérés dans l'article sont d'une part la facette fonctionnelle, c'est-à-dire les fonctions attendues du système et d'autre part la facette matérielle, c'est-à-dire le support physique réalisant les fonctions. Concrètement, la partie fonctionnelle est représentée par un ensemble de programmes LUSTRE mutli-horloges contraints par des précédences, et la partie matérielle est donnée sous forme d'une description AADL.

A partir de ces données, plusieurs ordonnancements de type EDF sans préemption respectant les contraintes fonctionnelles sont générées puis testées par l'outil Cheddar afin d'en vérifier la correction temps réel. Ensuite, un modèle du système complet peut être généré automatiquement en AADL permettant ainsi d'avoir une vision globale du système. Cette méthode sera illustrée sur un exemple simplifié des commandes de vol.

Plan Dans la section 2, nous présentons un exemple de commande de vol simplifié. Dans la section 3, nous formalisons les facettes fonctionnelle et matérielle. Dans la section 4, nous présentons une manière de générer des ordonnancements, respectant la sémantique synchrone de la facette fonctionnelle. Ces ordonnancements sont ensuite validés par l'utilisation d'un outil. Puis, pour un ordonnancement correct, un modèle AADL est généré, permettant ainsi à l'intégrateur d'avoir une vision globale du système encodé réellement.

2 Présentation de l'exemple

Nous nous situons dans le contexte de systèmes périodiques de type contrôle / commande. Ces systèmes sont en général composés de trois types d'action :

- acquisition de données, (par exemple dans le cas d'un aéronef, position de gouvernes, position de l'aéronef, vitesse) ;
- traitement, étant donné l'état observé, quelles actions doivent être réalisées pour atteindre l'état souhaité ;

– contrôle, envoi des ordres aux actionneurs.

Ces différentes actions se font en général à des rythmes différents. La recherche de la stabilité d'une loi de contrôle/commande est primordiale et donc réalisée à un rythme rapide. Dans le cas d'un aéronef, il s'agit de ne pas exposer la structure à des déformations catastrophiques. D'autres surveillances, comme le suivi d'un plan de vol, suivent un rythme plus lent.

A titre d'exemple, nous considérons un cas très simple de commandes de vol d'un avion. Ce système est destiné à contrôler l'attitude, la trajectoire et la vitesse de l'avion en mode de pilotage manuel. Plus précisément, cela comprend les organes de pilotage (manche, palonnier, …), les organes de transmission et de traitement des ordres de l'équipage (timoneries et les câbles dans le cas de commandes mécaniques, calculateurs et des câblage dans le cas de commandes électriques), les actionneurs ou servocommandes permettant de positionner les gouvernes.

Architecture fonctionnelle

Nous considérons la partie logicielle des commandes de vol. Un calculateur de commandes de vol assure le calcul des ordres de pilotage effectué par les lois de pilotage, et l'asservissement des gouvernes sur ces ordres de pilotage (lois d'asservissement).

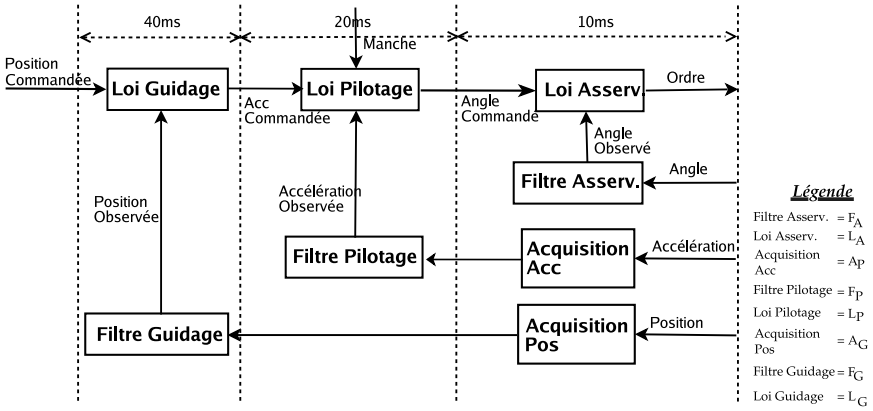


FIG. 1 – Architecture fonctionnelle des commandes de vol

La figure 2 représente l'architecture fonctionnelle de l'application : une boucle rapide à 10ms réalise les asservissements des gouvernes, la boucle intermédiaire à 20ms assure le guidage de l'avion et la boucle la moins rapide gère le pilotage automatique. On constate que les tâches communiquent entre elles et que l'ordre des traitements peut être important. Il existe donc des précédences entre elles sont dérivées d'exigences de fraîcheur des variables.

Architecture matérielle

Parallèlement au travail du bureau d'étude, les équipementiers choisissent un support physique réalisant la fonction. Dans l'exemple des commandes de vol, une architecture matérielle solution est décrite dans la figure 2. Elle

est constituée d'un calculateur réalisant le traitement logiciel (calcul des asservissements, loi de guidage et pilotage), de différents capteurs permettant l'acquisition de l'état de l'avion (le GPS donne la position, l'accéléromètre donne la vitesse, les capteurs ailerons et rudder[1] donnent les angles d'inclinaison des gouvernes) et des actionneurs réalisant les modifications des gouvernes (envoi de courant électrique afin de déplacer les ailerons et le rudder).

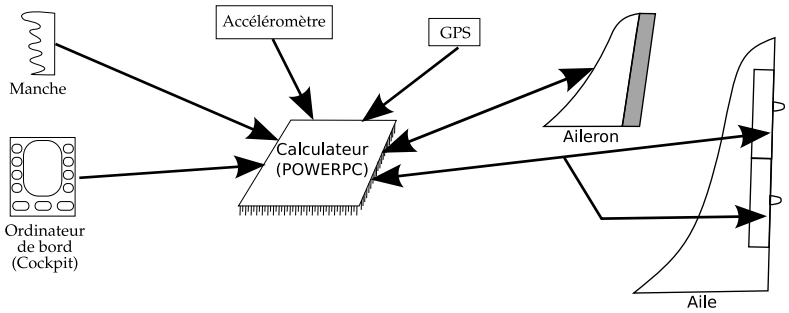


FIG. 2 – Architecture matérielle des commandes de vol

3 Formalisation des facettes

Nous nous intéressons dans cette section à deux facettes particulières : les spécifications fonctionnelles fournies par le bureau d'étude et le support physique choisi par les équipementiers. L'objectif de cette section est de formaliser ces deux facettes afin de faciliter leur manipulation et l'extraction d'information pour l'intégrateur système.

3.1 Facette fonctionnelle

Notre choix pour formaliser cette facette s'est naturellement tourné vers un ensemble de programmes *synchrones* multi-périodiques avec des contraintes de précédence. En effet, les langages synchrones sont souvent utilisés par les équipes du bureau d'études pour spécifier les fonctions et sont particulièrement adaptés à la spécification de systèmes réactifs, notamment ceux de type contrôle / commande.

Rappels sur Lustre

LUSTRE [CPHP87] est un langage synchrone flot de données développé à Verimag. Un programme *réactif* réagit aux stimuli de l'environnement. Un programme est *synchrone* si les réactions à un stimulus sont toujours suffisamment rapides pour terminer avant le prochain stimulus. On dit alors que les réactions se font en un *temps logique*. Cette approche permet de s'abstraire des contraintes temps réel associées à l'architecture physique.

[1] Le rudder est une gouverne de direction. http ://en.wikipedia.org/wiki/Portal :Aviation

Nous verrons section 4 comment relier la facette fonctionnelle à l'architecture physique : chaque opération prend en réalité un certain temps qui sera déterminant dans les performances de la fonction et donc le choix des ordonnancements.

Pour trouver une description complète et détaillée du langage, le lecteur pourra se référer à [Hal92]. 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. L'objet de base est donc la constante, ainsi 1 vaudra un à tous les instants. On peut construire de nouveaux objets à partir des opérateurs arithmétiques de base (+, -, ...), des opérateurs de comparaison ($<$, =, ...), des opérateurs d'horloge (*pre* retarde un flot de un instant, *fby* retarde également un flot tout en initialisant l'objet, *current* sur-échantillonne un flot sur l'horloge de l'horloge du flot, *when* sous-échantillonne un flot sur une horloge).

Voici quelques exemples de construction et la valeur des flots associés :

1	1	1	1	1	1	...
x	x_0	x_1	x_2	x_3	x_4	...
x + 1	x_0 + 1	x_1 + 1	x_2 + 1	x_3 + 1	x_4 + 1	...
$1 \rightarrow x$	1	x_0	x_1	x_2	x_3	...
c	t	t	f	f	f	...
if c then x else 1	x_0	x_1	1	1	1	...

Définition de la notion de facette fonctionnelle

Une facette fonctionnelle est un ensemble de nœuds LUSTRE cadencés à des périodes différentes communiquant entre eux et avec l'extérieur, assujettis à des contraintes de précédences. On suppose également qu'il n'y a pas d'opération d'horloge (*current*, *when*) dans les nœuds (i.e., chaque nœud est localement mono horloge). Soit une fonction f, on note $Dom(f)$ son domaine de définition et $Im(f)$ son image.

Définition 1 *Une facette fonctionnelle $\mathcal{F}onc = \langle (T_i, C_i)_{i \leq n}, \mathcal{F}_d, \mathcal{F}_p \rangle$ est composée d'un n-uplet et de deux fonctions :*

- *le n-uplet $(T_i, C_i)_{i \leq n}$ décrit pour chaque période $P_i = 2^{C_i - 1} b$ dérivée de la période de base b, l'ensemble des nœuds* LUSTRE *$T_i = \{n_1^i, \cdots, n_{j_i}^i\}$ rythmés à cette période. L'environnement est vu comme un nœud particulier noté indifféremment* env *ou n_0^1 et cadencé à l'horloge de base ;*
- *la fonction partielle $\mathcal{F}_d : in \rightarrow out$ décrit les communications synchrones entre nœuds (environnement inclus). On s'abstrait des noms de variables en numérotant de façon systématique les entrées/sorties de chaque nœud, en effet* in *(resp.*out*) correspond à toutes les entrées des nœuds (resp. les sorties des nœuds) :*

$$\begin{array}{rcl} in & = & \{n_j^i.in_k \mid n_j^i \in T_i,\ k = 1, \ldots, k_j^i,\ n_j^i \text{ a } k_j^i \text{ entrées}\} \\ out & = & \{n_j^i.out_k \mid n_j^i \in T_i,\ k = 1, \ldots, l_j^i,\ n_j^i \text{ a } l_j^i \text{ sorties}\} \end{array}$$

- *la fonction partielle $\mathcal{F}_p : in \rightarrow out$ décrit les communications entre nœuds retardées (le flot consommé est l'avant dernier flot produit).*

Les exigences de l'automatique portent sur la fraîcheur des variables, par exemple un flot ne peut être *consommé* que s'il a été produit depuis

moins de 10 μs. Nous les avons dérivées sous forme du découpage $\mathcal{F}_d$ et $\mathcal{F}_p$. L'intégrateur système devra vérifier a posteriori le respect de ces exigences.

Exemple 1 *La période de base des commandes de vol est $b = 10$ ms. La description formelle des spécifications est donnée par :*

- $\big((\{L_A, F_A, A_P, A_G\}, 1), (\{F_P, L_P\}, 2), (\{F_G, L_G\}, 3)\big)$;
- *la fonction $\mathcal{F}_d$ est totale et définie par*
  $\mathcal{F}_d(L_A.in_1) = F_A.out_1, \mathcal{F}_d(L_A.in_2) = L_P.out_1,$
  $\mathcal{F}_d(F_P.in_1) = A_P.out_1, \mathcal{F}_d(L_P.in_1) = F_P.out_1,$
  $\mathcal{F}_d(L_P.in_2) = L_G.out_1, \mathcal{F}_d(F_G.in_1) = A_G.out_1,$
  $\mathcal{F}_d(L_G.in_1) = F_G.out_1, \mathcal{F}_d(F_A.in_1) = env.out_1,$
  $\mathcal{F}_d(L_G.in_2) = env.out_4, \mathcal{F}_d(env.in_1) = L_A.out_1,$
  $\mathcal{F}_d(A_P.in_1) = env.out_2, \mathcal{F}_d(A_G.in_1) = env.out_3,$
  $\mathcal{F}_d(L_P.in_3) = env.out_5$
- $Dom(\mathcal{F}_p) = \emptyset$

On retrouve l'architecture fonctionnelle de la figure 2. Les variables sont transcrites dans les fonctions, par exemple la variable Angle Commandé *(resp.* Manche*) est représentée par la relation $\mathcal{F}_d(L_P.in_2) = L_G.out_1$ (resp. $\mathcal{F}_d(L_P.in_3) = env.out_5$).*

Causalité dans une facette fonctionnelle Les problèmes de causalité ont été étudiés pour LUSTRE [HRR91] pour des raisons de compilation. Cette recherche se fait au niveau des variables. Dans notre contexte, nous regardons la causalité au niveau nœud, ce qui est à peu près équivalent. Il est possible de déterminer s'il y a un cycle dans une facette fonctionnelle. En effet, on note $\mathcal{T} = \bigcup_i T_i$ l'ensemble des tâches de la facette. Soit $\mathcal{P} : \mathcal{T} \to 2^{\mathcal{T}}$ la fonction qui à chaque nœud n associe l'ensemble des nœuds dont une des sorties au moins est prise en entrée (non retardée) par n. On peut définir $\mathcal{P}$ par $\mathcal{P}(n) = \bigcup_{l=1}\{node_{out}(\mathcal{F}_d(n.in_l))\}$ avec $node_{out}(n.out) = n$.

Afin de calculer la fermeture transitive de $\mathcal{P}$, on définit la fonction intermédiaire $\mathcal{P}_{2^{\mathcal{T}}}(\mathcal{N}) : 2^{\mathcal{T}} \to 2^{\mathcal{T}}$ avec $\mathcal{P}_{2^{\mathcal{T}}}(\mathcal{N}) = \bigcup_{n\in\mathcal{N}} \mathcal{P}(n)$ où $\mathcal{N}$ un ensemble quelconque de nœuds. Alors, $\mathcal{P}^{\infty}(n) : \mathcal{T} \to 2^{\mathcal{T}}$ donne l'ensemble des nœuds qui sont en relation avec n, i.e. si $n' \in \mathcal{P}^{\infty}(n)$, pour calculer n il faut les résultats de n'. A noter que comme $\mathcal{T}$ est fini, on est assuré de la terminaison du calcul.

$$\mathcal{P}^{\infty}(n) = \bigcup_{i=1}^{\infty} \mathcal{P}^i_{2^{\mathcal{T}}}(\{n\}) \text{ avec } \mathcal{P}^i_{2^{\mathcal{T}}} = \underbrace{\mathcal{P}_{2^{\mathcal{T}}} \circ \ldots \circ \mathcal{P}_{2^{\mathcal{T}}}}_{i}$$

Définition 2 (Facette fonctionnelle bien construite) *Une facette est bien construite si $Dom(\mathcal{F}_d) \cap Dom(\mathcal{F}_p) = \emptyset$ (pas de relation conflictuelle) ; si $Dom(\mathcal{F}_d) \cup Dom(\mathcal{F}_p) = in$ et $Im(\mathcal{F}_d) \cup Im(\mathcal{F}_p) = out$ (le système est fermé, aucune entrée/sortie n'est indéfinie) ; et enfin si $\forall n \in \mathcal{T},\ n \notin \mathcal{P}^{\infty}(n)$ (pas de cycle et donc pas de problème de causalité). On note alors $\mathcal{F} : in \to out$ la fonction totale définie par $\mathcal{F} = \mathcal{F}_d \cup \mathcal{F}_p$.*

Dans la suite on s'intéresse uniquement aux facettes bien construites.

Sémantique d'une facette fonctionnelle

Notre approche est de laisser le plus de liberté possible à l'intégrateur système, lui-même soumis à d'autres obligations (contraintes de sûreté de fonctionnement, de certification ...), tout en restant dans des exécutions synchrones. Plutôt que d'imposer un choix, la sémantique d'une facette fonctionnelle décrit les assemblages possibles des nœuds LUSTRE respectant les contraintes introduites par $\mathcal{F}_d$ et $\mathcal{F}_p$.

Horloges périodiques Afin de définir les assemblages conformes, nous utiliserons les horloges pour différencier les ordonnancements possibles des nœuds. Une *horloge* [HCRP91] associée à une variable LUSTRE est un flot booléen qui indique si la variable est présente ou non par rapport à l'horloge de base.

Nous reprenons les notations introduites dans [CDE$^+$06]. Un mot binaire infini appartient à $(0+1)^\omega$. Les tâches considérées étant périodiques, nous nous intéressons au sous ensemble des *mots binaires infinis périodiques sans préfixe*, c'est-à-dire aux mots de la forme (v) où $v \in (0+1)^*$ est un mot fini qui se répète à l'infini.

On note par $[w]_p$ la position du $p-$ième 1 dans w. On définit alors la relation d'ordre sur les mots binaires : $w_1 \succeq w_2 \iff \forall p \geq 1,\ [w_1]_p \geq [w_2]_p$. Comparer deux mots binaires périodiques est calculable.

Notation 1 *On note pour tout $i, j \in \mathbb{N}$, (i/j) l'ensemble des horloges périodiques (v) où v est de longueur j avec exactement i 1, i.e.*

$$(i/j) = \{(v) \mid |v| = j \wedge |v|_1 = i\}$$

Horloge associée à un nœud La fonction *horloge* associe à chaque variable son horloge dans $(0+1)^\omega$. On associe à chaque nœud une *signature d'horloges*, c'est à dire la transformation des horloges des variables d'entrée vers les horloges des variables de sorties. Ainsi tout nœud n a pour signature : *horloge*$(n.in_1) \times \cdots \times$ *horloge*$(n.in_{k_n}) \rightarrow$ *horloge*$(n.out_1) \times \cdots \times$ *horloge*$(n.out_{l_n})$. Comme les nœuds considérés n'utilisent pas d'opérateurs d'horloges, toutes les variables d'entrée et de sortie ont la même horloge. Ainsi, *horloge*$(n.in_{k_i}) =$ *horloge*$(n.out_{l_j}) = a_n$. Donc la signature d'horloge de n est $a_n^{k_n} \rightarrow a_n^{l_n}$ que nous noterons par abus de notation a_n. On étend alors la fonction *horloge* sur les nœuds de sorte que *horloge*$(n) = a_n$.

Horloges associées aux nœuds de la facette fonctionnelle Si un nœud n appartient au cycle C_i, on en déduit que *horloge*$(n) \in (1/C_i)$. D'autres contraintes portent sur *horloge*(n) issues de $\mathcal{F}_d$ et $\mathcal{F}_p$:

- contraintes sur les nœuds de même période : si deux nœuds n_j^i et $n_{j'}^i$ sont en relation $n_j^i \in \mathcal{P}^\infty(n_{j'}^i)$, alors l'écriture devra se réaliser avant ou sur la même horloge *horloge*$(n_{j'}^i) \succeq$ *horloge*(n_j^i) ;
- contraintes des nœuds rapides vers des plus lents : si deux nœuds n_j^i et $n_{j'}^{i'}$ sont en relation $n_j^i \in \mathcal{P}^\infty(n_{j'}^{i'})$, $i' > i$, le lecteur prendra la valeur la plus récente de son rythme *horloge*$(n_{j'}^{i'}) \succeq$ *horloge*(n_j^i).

Exemple 2 *A titre d'exemple, l'horloge de L_P est de la forme $(1/2)$, c'est-à-dire (01) ou (10). D'après la fonction $\mathcal{F}_d$, l'horloge de L_P est plus grande que celle de F_P, laquelle n'est pas contrainte. Ainsi, si horloge$(F_P) = (10)$, on aura horloge$(L_P) = (10)$ ou (01). Si en revanche, horloge$(F_P) = (01)$, on aura nécessairement horloge$(L_P) = (01)$.*

On généralise cette idée et on en déduit alors toutes les horloges possibles associées aux nœuds d'une facette fonctionnelle :

Définition 3 *Soit $\mathcal{F}onc = \langle (T_i, C_i)_{i \leq n}, \mathcal{F}_d, \mathcal{F}_p \rangle$ une facette fonctionnelle. Tous les nœuds à C_1 ont pour horloge (1) et les combinaisons possibles des horloges des autres nœuds sont données par $(\text{horloge}(n_1^2), \ldots, \text{horloge}(n^n_{\#T_n})) \in \mathcal{C}$ où $\mathcal{C}$ est défini par :*

$$\mathcal{C} = \bigcup \left\{ (h_1^2, \ldots, h^2_{\#T_2}, \ldots, h^n_{\#T_n}) \;\middle|\; \begin{array}{l} \forall i \in [1,n], \forall l \in [1, \#T_i], \\ h_l^i \in (1/C_i) \;\wedge \forall i, i' \in [1,n], \\ \forall l \in [1, \#T_i], \forall l' \in [1, \#T_{i'}], \\ n_l^i \in \mathcal{P}^\infty \left(n_{l'}^{i'} \right) \wedge i \geq i' \Longrightarrow \\ h_j^i \succeq h_{j'}^{i'} \end{array} \right\}$$

L'ensemble $\mathcal{C}$ donne tous les ordonnancements conformes à la facette fonctionnelle.

Sémantique d'une facette fonctionnelle Pour faire communiquer un flot et un nœud non synchrones, nous utiliserons la fonction *rsync* qui prend un nœud et un flot et resynchronize le flot sur le nœud.

$$\textit{rsync}(x, n) = \left\{ \begin{array}{ll} x \;\; \textit{when}\; clock(n) & \textit{si}\; x \in out_{|T_1} \\ (\textit{current}(x)) \;\; \textit{when}\; clock(n) & \textit{sinon} \end{array} \right.$$

L'idée est de sur-échantillonner x sur l'horloge de base (1) puis de l'échantillonner sur l'horloge de n. A titre d'exemple,

x	x_1		x_2		x_3		x_4	
current(x)	x_1	x_1	x_2	x_2	x_3	x_3	x_4	x_4
current(x) when (0100)		x_1				x_3		

On définit la fonction intermédiaire f qui implante la communication synchrone ou retardée :

$$f(\mathcal{F}(n_j^i.in_l)) = \left\{ \begin{array}{ll} \mathcal{F}(n_j^i.in_l) & \text{si} \quad n_j^i \in Dom(\mathcal{F}_d) \\ \text{pre}(\mathcal{F}(n_j^i.in_l)) & sinon \end{array} \right.$$

Nous pouvons maintenant définir la sémantique d'une facette fonctionnelle $\mathcal{F}onc = \langle (T_i, C_i)_{i \leq n}, \mathcal{F}_d, \mathcal{F}_p \rangle$. Il s'agit de l'ensemble de nœuds LUSTRE obtenus à partir des horloges possibles $\mathcal{C}$ de la forme :

node main ($env.out_1$,...,$env.out_k$)
returns ($env.in_1, \ldots, env.in_k$)

var

$clock_{n_j^i}$ pour $i = 2, \cdots, p$; $j = 1, \cdots, \#T_i$

$n_j^i.out_1, \ldots, n_j^i.out_{l_j^i}$ when $clock(n_j^i)$ pour $i = 1, \cdots, p$; $j = 1, \cdots, \#T_i$

let

$env.in_i = n_0^1.out_i$; $env.out_i = n_0^1.in_i$;

$(clock_{n_1^2}, \ldots, clock_{n^p_{\#T_p}}) \in \mathcal{C}$

...

$n_j^i.out_1, \ldots, n_j^i.out_{l_j^i} = n_j^i\big(\mathit{rsync}(f(\mathcal{F}(n_j^i.in_1)), n_j^i), \ldots, \mathit{rsync}(f(\mathcal{F}(n_j^i.in_{k_j^i})), n_j^i)\big)$;

...

tel ;

Code 3.1: Patron d'un nœud d'assemblage

Exemple 3 *L'ensemble des 30 assemblages possibles pour les commandes de vol est donné en annexe 5 page 15.*

3.2 Facette matérielle

La facette matérielle décrit le point de vue des équipementiers, à savoir l'architecture physique. Pour décrire cette facette, un bon candidat est un langage d'architecture, comme par exemple AADL [FLV03]. AADL (Achitecture & Analysis Description Language) est un langage dédié à la description d'architectures allant des composants physiques à la couche exécutive.

Pour trouver une description complète et détaillée du langage, le lecteur pourra se référer à [FGH06]. De manière informelle, un système est décrit en AADL sous forme de composants de nature matérielle, logicielle ou système. Chaque composant est décrit en deux temps : son type (partie publique) et son implémentation (description plus détaillée héritant des propriétés du type).

Pour décrire l'architecture matérielle, les composants matériels nous suffisent. Dans le langage AADL plusieurs types de composants matériels sont pré-définis comme les processeurs, les mémoires, les réseaux, les périphériques ... Dans l'architecture de l'exemple, le processeur est un PowerPC alimenté par une source *Power*, connecté à un réseau Ethernet *HS* et à un réseau avionique Arinc429 *HA*, n'acceptant que des programmes C. Le type AADL est alors :

processor PowerPC
features
HS : **requires bus access** EtherSwitch
HA : **requires bus access** Arinc
Power : **requires bus access** PowerSupply.V9_0 ;
properties
Supported_Source_Language = C ;
end PowerPC ;

Il est ainsi possible de décrire l'architecture de la figure 2 : chaque connexion est un réseau, les capteurs et actionneurs sont vus comme des périphériques. La description AADL est fournie en Annexe 5 page 16.

Dans le modèle proposé, la vue matérielle ne se confine pas uniquement aux processeurs et autres éléments matériels mais elle contient également les services offerts par la plateforme. On entend par service toute fonction du domaine purement exécutif comme par exemple la communication à travers un réseau, la gestion des entrées/sorties du logiciel embarqué, ou bien la communication interne entre les nœuds. Dans la pratique, le processeur est pourvu d'un OS minimal et les services sont fournis par un ensemble de librairie écrites en C ou en Assembleur.

Dans cette section, nous avons formalisé les données issues des équipes pour l'intégrateur système. La prochaine section va se focaliser sur le travail de ce dernier.

4 Génération du point de vue de l'intégration globale

Nous nous intéressons maintenant à l'intégration des deux facettes pour la conception du système complet. L'intégrateur doit faire tourner le code C généré à partir de la facette fonctionnelle sur le PowerPC. La sémantique de la facette fonctionnelle fournit plusieurs choix d'implantation : l'idée est de traduire chaque nœud LUSTRE en C et de produire à partir des informations des facettes, une implantation valide par rapport à cette sémantique. Dans cette section, nous donnons un moyen formel d'intégrer le code sur un support matériel sous des hypothèses assez fortes : génération automatique d'ordonnancements EDF à partir de la facette fonctionnelle et validation avec un outil. Ensuite, pour un ordonnancement valide, on génère un modèle global AADL du système réellement encodé.

4.1 Ordonnançabilité

Un ordonnancement consiste à organiser dans le temps l'exécution d'un ensemble tâches compte tenu de contraintes temporelles et de contraintes portant sur la disponibilité des ressources requises.

Le modèle de Liu-Layland [LL73] permet de décrire les caractéristiques temps réel des tâches en s'abstrayant de la partie comportementale. Une tâche est alors représentée par un 4-uplet $\mathcal{T} = (T, c, r, d)$ avec T la période du processus, c le temps maximal d'exécution (WCET), r la date de réveil et d la deadline (date impérative à laquelle le processus doit être achevé).

Le calcul du WCET [FH04] d'une tâche dépend de plusieurs éléments : le processeur, le compilateur C, les différentes mémoires ... C'est un domaine de recherche à part entière et dans notre contexte, [Fre05] propose des techniques pour le PowerPC.

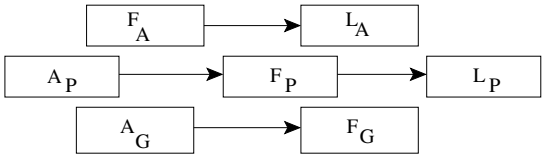
On s'intéresse aux ordonnancements de type EDF (Earliest Deadline First) qui permettent d'encoder les précédences par modification des deadline. En effet, le principe d'un tel ordonnancement est de donner la priorité à une tâche dont l'échéance est la plus proche. A noter qu'on se restreint à un EDF non préemptif, donc chaque tâche s'exécute dans son intégralité.

Génération automatique des ordonnancements Pour chaque affectation d'horloge possible de la facette fonctionnelle, c'est-à-dire un élément de l'ensemble $\mathcal{C}$, un modèle de Liu-Layland est automatiquement généré pour chacun des nœuds. La période de chaque nœud est connue. On suppose qu'un outil permet à l'intégrateur de connaître le WCET pour chaque nœud. La date de réveil se calcule en fonction de l'horloge *horloge*(n) et l'échéance s'obtiendra à partir des contraintes de précédence de $\mathcal{F}_d$.

Soient une facette fonctionnelle $\mathcal{F}onc = \langle (T_i, C_i)_{i \leq n}, \mathcal{F}_d, \mathcal{F}_p \rangle$, une affectation d'horloge valide $\in \mathcal{C}$ définie dans la facette fonctionnelle et un nœud n, nous calculons son modèle de tâche $\mathcal{T} = (T, c, r, d)$.

La période T est celle définie dans la facette fonctionnelle, à savoir si $n \in T_i$, $T = 2^{C_i - 1}b$. Le WCET est supposé connu. La date de réveil est imposée par l'affectation d'horloge, $r = ([\textit{horloge}(n)]_1 - 1)b$. L'échéance est contrainte par l'hypothèse synchrone, donc $d \leq b$. En effet, on force la tâche à se réveiller et s'exécuter à son horloge logique. Pour coder les précédences imposées par $\mathcal{F}_d$, il faut contraindre les échéances. Ce travail a déjà été réalisé dans [RC99] dans le cas multi-période, ce qui étend le cas mono-période de [CSB90]. L'idée est de résoudre un système d'inéquations : $n \in \mathcal{P}^{\infty}(n')$ alors $d_n \leq d_{n'} - c_{n'}$.

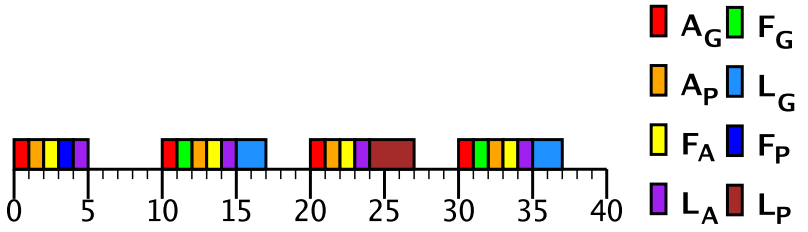
Exemple 4 *Soit une configuration de la sémantique des commandes de vol, horloge (F_G)= horloge (L_G)=(01), horloge (F_P) =(1000), horloge (L_P) = (0010). Le graphe de précédence est décrit dans le schéma suivant :*



Les contraintes générées sont alors :

$$\begin{cases} d_{L_A} - c_{L_A} \geq d_{F_A} \\ d_{L_G} - c_{L_G} \geq d_{F_G} \\ d_{A_G} - c_{A_G} \geq d_{L_G} \\ d_{A_P} - c_{A_P} \geq d_{F_P} \end{cases}$$

Ce qui donne les modèles de tâche $\mathcal{T}_{F_A} = (10, 1, 0, 9)$, $\mathcal{T}_{L_A} = (10, 1, 0, 7)$, $\mathcal{T}_{A_G} = (10, 1, 0, 7)$, $\mathcal{T}_{A_P} = (10, 1, 0, 9)$, $\mathcal{T}_{F_G} = (20, 1, 0, 8)$, $\mathcal{T}_{L_G} = (20, 2, 0, 10)$, $\mathcal{T}_{F_P} = (40, 1, 0, 10)$, $\mathcal{T}_{L_G} = (40, 3, 10, 10)$. *Ce qui nous donne le chronogramme suivant :*



Hypothèses Dans ce papier, nous faisons des hypothèses fortes sur les ordonnancements générés : ils sont de type EDF sans préemption. On assure donc que la somme des WCET des nœuds placés sur un cycle de 10 ms

est inférieur à 10 ms et les contraintes de réveil, WCET et deadline sont construits de façon à obtenir un ordonnancement non préemptif.

Les exécutions possibles, même dans le cas non préemptif, ne sont pas déterministes. En effet, si deux nœuds dépendants fonctionnellement par $\mathcal{F}_p$ ont la même deadline, ils pourront s'exécuter dans n'importe quel ordre. Une première solution pour éviter cela pourrait être de modifier les deadlines pour prendre en compte ce phénomène. Malheureusement, cela pourrait entraîner des problèmes de causalité. Une deuxième solution consiste à ajouter un nœud intermédiaire, une sorte de buffer, qui prendrait les sorties du nœud et les recopierait en fin de période. Cela forcerait les exécutions de l'ordonnancement à être déterministes.

Choix d'un ordonnancement Pour chaque configuration d'horloge, on obtient un ordonnancement. Tous ces ordonnancements ne sont pas nécessairement corrects en terme de temps réel, il se peut qu'il y ait un dépassement d'échéance. Il existe des outils pour vérifier l'ordonnançabilité d'un ensemble de tâches. On se propose dans le présent article d'utiliser l'outil Cheddar [SLNM04] qui prend en entrée des modèles de tâches multi périodes, afin d'aider au choix d'une configuration d'ordonnancement.

L'outil Cheddar prend en entrée une description AADL ou un modèle de Liu-Layland. Pour une description AADL il prend en compte que la borne maximale de temps d'exécution. L'hypothèse synchrone ne nécessite pas de connaître les meilleurs temps de calculs pour l'étude d'ordonnaçabilité.

4.2 Génération du modèle d'intégration global

Une fois l'ordonnancement global obtenu, il est aisé de traduire automatiquement la facette fonctionnelle en AADL et de la relier au support matériel pour un ordonnancement valide. La partie logicielle AADL permet de décrire la facette fonctionnelle et la partie système exprimera les relations avec le matériel, ce qui donnera à l'intégrateur une vision globale du code embarqué.

La transformation est la suivante :

- chaque nœud LUSTRE est décrit par un thread. En effet, un thread AADL contient tous les éléments temps réel d'un nœud : ports, type de communication et de propriétés temporelles (période, deadline, temps d'exécution, ...) ;
- l'architecture fonctionnelle, c'est-à-dire la relation flot de données $\mathcal{F}$ est décrite par un processus AADL regroupant les threads ;
- le composant système décrira les relations entre le logiciel et le matériel.

Génération des threads Pour chaque période T_i, un type thread de période T_i est généré. Par exemple,

– *thread pour* T_i
thread threadi
properties
Dispatch_Protocol $\Rightarrow$ Periodic ;
Period $\Rightarrow 2^{C_i-1}$ Ms ;
end threadi ;

Pour chaque nœud n, un type thread n est généré décrivant les entrées/sorties du composant. Par exemple, le nœud F_A s'écrit en AADL :

thread FAtype
extends thread1
features
in1 : **in data port** BaseType : :Integer ;
out1 :**out data port** BaseType : :Integer ;
end FAtype ;

L'implémentation des nœuds est équivalente à la génération du modèle de Liu-Layland avec en sus l'intégration des deadline et dates de réveil calculées (en AADL les dates de reveils ne sont pas gérées, nous utilisons l'extension proposée dans[SLNM05]) dans la section précédente.

thread implementation FAtype.FAImpl
properties
Compute_Execution_Time $\Rightarrow$ 2ms..5ms ;
Deadline $\Rightarrow$ 9 ;
Cheddar_Properties : :Dispatch_Absolute_Time $\Rightarrow$ 0 ;
end FAtype.FAImpl ;

Génération du processus Le processus déclare les entrées/sorties avec l'environnement, tous les sous-composants, c'est-à-dire tous les nœuds LUSTRE et les relations entre les ports de tous les threads, c'est-à-dire la description de la fonction $\mathcal{F}$.

Le processus des commandes de vol est décrit en annexe 5 page 19.

Assemblage du composant système L'intégrateur doit relier la facette fonctionnelle et la facette matérielle. Pour cela il relie les entrées/sorties de l'environnement de la facette fonctionnelle et sur les capteurs/actionneurs de la facette matérielle. L'intégrateur attribuera ces connexions sur les bus de la facette matérielle.

Le système des commandes de vols est décrit en annexe 5 page 19.

5 Conclusion

Le but de cet article était de présenter une approche d'aide à la conception reposant sur la séparation des points de vue, et sur la génération de points de vue globaux, tels que l'intégration, à partir de points de vue locaux. Nous avons considéré deux points de vue locaux, appelés facettes : la facette fonctionnelle et la facette matérielle. Nous avons alors étudié et défini le contenu de ces deux facettes de manière à pouvoir ensuite générer le point de vue exécutif, nécessairement global et transversal, de l'intégrateur. Plus spécifiquement, nous nous sommes intéressés à la génération d'ordonnancements corrects de sorte que les exécutions soient sémantiquement équivalentes à un nœud synchrone. Dans ce papier, nous avons présenté une formalisation de notre cadre multi vue et une première approche pour générer des ordonnancements.

Ce travail n'est qu'une première ébauche s'appuyant sur des hypothèses de travail très fortes et restrictives : l'utilisation d'un EDF non préemptif imposant le choix de nœuds LUSTRE ayant des WCET adaptés (et donc en général une faible granularité), ainsi que la choix d'une facette matérielle mono processeur. Ces deux hypothèses simplifient considérablement la génération de l'intégration, notamment en évacuant la question de la répartition des nœuds et des communications sur une architecture distribuée. La prochaine étape de ce travail consistera à étendre l'approche en relachant ces hypothèses. Des travaux ont déjà été menés dans ce sens, notamment dans [STC06].

Parallèlement, seules les propriétées temporelles du processeur ont été prise en compte. Dans une prochaine étape de tiendra compte des délais de l'architecture physique comme les bus.

Références

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

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[CSB90] Houssine Chetto, Maryline Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3) :181–194, 1990.

[FGH06] Peter H. Feiler, David P. Gluch, and John Hudak. The architecture & analysis design language (aadl) : An introduction. Technical report, Carnegie Mellon University, feb 2006. CMU/SEI-2006-TN-011.

[FH04] Christian Ferdinand and Reinhold Heckmann. aiT : worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions 2004*, pages 377–384, 2004.

[FLV03] Peter H. Feiler, Bruce Lewis, and Steve Vestal. The SAE Architecture Analysis & Design Language (AADL) standard : A basis for model-based architecture driven embedded systems engineering. In *RTAS 2003 Worshop on Model-Driven Embedded Systems*, Washington, D.C., May 2003. IEEE.

[Fre05] Freescale. Mpc755 risc microprocessor hardware specifications, Mar. 2005.

[Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[RC99] Pascal Richard and Francis Cottet. Ordonnancement temps réel des contrainntes de précédence généralisé. Technical report, LISI/ENSMA, 1999.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV(4) :1–8, 2004.

[SLNM05] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. In *SigAda '05 : Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[STC06] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *EMSOFT '06 : Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 21–33, New York, NY, USA, 2006. ACM Press.

[The06] Wolfgang Theurer. *Une méthodologie de modélisation multi-modèles distribuée par métier pour les sytème embarqués*. PhD thesis, Supaéro, 2006.

Annexes

Assemblage fonctionnel des commandes de vol

Les nœuds LUSTRE appartenant à la sémantique de la facette fonctionnelle associée aux commandes de vol sont de la forme suivante :

node main_CDV($env.out_1, env.out_2, env.out_3, env.out_4, env.out_5$)
returns ($env.in_2$)
var
  $clock_{F_P}$, $clock_{L_P}$, $clock_{F_G}$, $clock_{L_G}$, $L_A.out_1$, $F_A.out_1$,
  $A_P.out_1, F_P.out_1$, $L_P.out_1$, $A_G.out_1, F_G.out_1$, $L_G.out_1$;
let
  $(clock_{F_P}, clock_{L_P}, clock_{F_G}, clock_{L_G}) \in \mathcal{C}$;
  $L_A.out_1 = L_A(F_A.out_1, \texttt{current}(L_p.out_1))$;
  $F_A.out_1 = F_A(env.out_1)$;
  $L_P.out_1 = L_p((\texttt{current}(F_P.out_1))\texttt{when } clock_{L_P},$
  $(\texttt{current}(L_g.out_1))\texttt{when } clock_{L_P},$
  $env.out_5\texttt{when } clock_{L_P})$;
  $F_P.out_1 = F_P((\texttt{current}(A_P.out_1))\texttt{when } clock_{F_P})$;

$A_P.out_1 = A_P(env.out_2)$;
$L_G.out_1 = L_G((\texttt{current}(F_G.out_1))\texttt{when } clock_{L_G}, env.out_4\texttt{when } clock_{L_G})$;
$F_G.out_1 = F_G((\texttt{current}(A_G.out_1))\texttt{when } clock_{F_G})$;
$A_G.out_1 = A_G(env.out_3)$;
tel

Les 30 affectations possibles des horloges peuvent être décrites par le tableau suivant :

	$clock_{L_P}$	$clock_{F_P}$	$clock_{L_G}$	$clock_{F_G}$
1	H_1^2	H_1^2	H_1^3	H_1^3
2	H_2^2	H_1^2	H_1^3	H_1^3
3	H_2^2	H_2^2	H_1^3	H_1^3
4	H_1^2	H_1^2	H_2^3	H_1^3
5	H_2^2	H_1^2	H_2^3	H_1^3
6	H_2^2	H_2^2	H_2^3	H_1^3
7	H_1^2	H_1^2	H_3^3	H_1^3
8	H_2^2	H_1^2	H_3^3	H_1^3
9	H_2^2	H_2^2	H_3^3	H_1^3
10	H_1^2	H_1^2	H_4^3	H_1^3
11	H_2^2	H_1^2	H_4^3	H_1^3
12	H_2^2	H_2^2	H_4^3	H_1^3
13	H_1^2	H_1^2	H_2^3	H_2^3
14	H_2^2	H_1^2	H_2^3	H_2^3
15	H_2^2	H_2^2	H_2^3	H_2^3
16	H_1^2	H_1^2	H_3^3	H_2^3
17	H_2^2	H_1^2	H_3^3	H_2^3
18	H_2^2	H_2^2	H_3^3	H_2^3
19	H_1^2	H_1^2	H_4^3	H_2^3
20	H_2^2	H_1^2	H_4^3	H_2^3
21	H_2^2	H_2^2	H_4^3	H_2^3
22	H_1^2	H_1^2	H_3^3	H_3^3
23	H_2^2	H_1^2	H_3^3	H_3^3
24	H_2^2	H_2^2	H_3^3	H_3^3
25	H_1^2	H_1^2	H_4^3	H_3^3
26	H_2^2	H_1^2	H_4^3	H_3^3
27	H_2^2	H_2^2	H_4^3	H_3^3
28	H_1^2	H_1^2	H_4^3	H_4^3
29	H_2^2	H_1^2	H_4^3	H_4^3
30	H_2^2	H_2^2	H_4^3	H_4^3

Description de l'architecture matérielle en AADL

Dans cette section nous donnons le code AADL décrivant l'architecture matérielle de l'exemple figure 2. L'architecture AADL est représentée dans la figure 3.

processor PowerPc
features

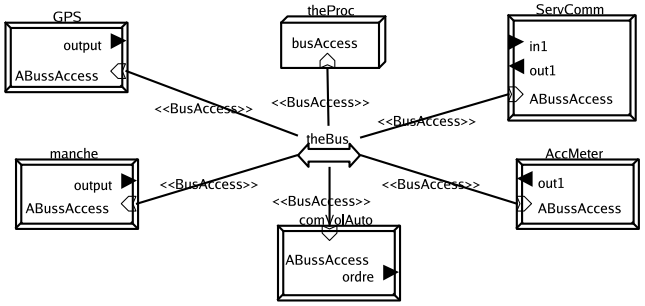


FIG. 3 – Architecture matérielle des commandes de vol en AADL

```
    busAccess : requires bus access Abus.impl ;
end PowerPc ;

device ServoCommandeType
extends DeviceGen
features
    in1 : in data port DataTypes : :integer ;
    out1 : out data port DataTypes : :integer ;
end ServoCommandeType ;

device AccMeterType
extends DeviceGen
features
    out1 : out data port DataTypes : :integer ;
end AccMeterType ;

memory mem
end mem ;

device DeviceGen
features
    ABussAccess : requires bus access Abus.impl ;
properties
    Device_Dispatch_Protocol ⇒ Periodic ;
end DeviceGen ;

device GPSType
extends DeviceGen
features
    output : out data port DataTypes : :integer ;
end GPSType ;

device mancheType
extends DeviceGen
features
    output : out data port DataTypes : :integer ;
```

```
end mancheType ;

bus Abus
end Abus ;

device commPiloteAutoType
extends DeviceGen
features
    ordre : out data port DataTypes : :integer2 ;
end commPiloteAutoType ;

processor implementation PowerPc.mp755
subcomponents
    theMem : memory mem.impl ;
properties
    Scheduling_Protocol ⇒ EDF ;
    Allowed_Dispatch_Protocol ⇒ Periodic ;
end PowerPc.mp755 ;

device implementation ServoCommandeType.impl
properties
    Period ⇒ 10 Ms ;
 end ServoCommandeType.impl ;

device implementation AccMeterType.impl
properties
    Period ⇒ 10 Ms ;
end AccMeterType.impl ;

 memory implementation mem.impl
properties
    Word_Count ⇒ 1024 ;
end mem.impl ;

device implementation GPSType.impl
properties
    Period ⇒ 10 Ms ;
end GPSType.impl ;

device implementation mancheType.impl
properties
    Period ⇒ 10 Ms ;
end mancheType.impl ;

bus implementation Abus.impl
end Abus.impl ;

device implementation commPiloteAutoType.impl
properties
```

Period ⇒ 10 Ms ;
end commPiloteAutoType.impl ;

Description du processus AADL

Dans cette section nous présentons la description de l'assemblage de la facette fonctionnelle.

process main
features
out4 : **in data port** DataTypes : :integer2 ;
out1 : **in data port** DataTypes : :integer ;
in1 : **out data port** DataTypes : :integer ;
out2 : **in data port** DataTypes : :integer ;
out3 : **in data port** DataTypes : :integer ;
out5 : **in data port** DataTypes : :integer ;
end main ;

process implementation main.impl
subcomponents
FA : **thread**FAtype.impl ;
LA : **thread** LAtype.impl ;
FP :**thread** FPType.impl ;
LP : **thread** LPType.impl ;
FG : **thread** FGType.impl ;
LG : **thread** LGType.impl ;
AP : **thread** APType.impl ;
AG : **thread** AGType.impl ;
connections
FA_out1ToLA_In1 : **data port** FA.out1 → LA.in1 ;
LP_out1ToLA_in2 :**data port** LP.out1 → LA.in2 ;
FP_out1ToLP_in1 : **data port** FP.out1 → LP.in1 ;
LG_out1ToLP_in2 : **data por**t LG.out1 → LP.in2 ;
FG_out1ToLG_in1 : **data port** FG.out1 → LG.in1 ;
AAToFP_in1 : **data port** AA.out1 → FP.in1 ;
AA_out1ToFG_in1 :**data port** AA.out1 → FG.in1 ;
envOut1_to_FAIn1 : **data port** out1 → FA.in1 ;
LAOut1_to_envIn1 :**data port** LA.out1 → in1 ;
envOut2_to_APIn1 : **data port** out2 → AP.in1 ;
envOut3_to_AGIn1 :**data port** out3 → AG.in1 ;
envOut4_to_LGIn2 :**data port** out4 → LG.in2 ;
envOut5_to_LPIn2 :**data port** out5 → LP.in3 ;
end main.impl ;

Assemblage du système en AADL

Dans cette section nous présentons le code en AADL permettant la description de l'assemblage de la facette fonctionnelle et la facette matérielle de

nos commande de vol.

```
system main
end main ;

system implementation main.impl
subcomponents
    AccMeter : device Hardware : :AccMeterType.impl ;
    theProc : processor Hardware : :PowerPc.mp755 ;
    ServComm : device Hardware : :ServoCommandeType.impl ;
    GPS : device Hardware : :GPSType.impl ;
    manche : device Hardware : :mancheType.impl ;
    theProcess : process CDVSoftware : :main.impl ;
    theBus : bus Hardware : :Abus.impl ;
    comVolAuto : device Hardware : :commPiloteAutoType.impl ;
connections
    DataConnection1 :data port theProcess.in1 → ServComm.in1 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    DataConnection2 : data port ServComm.out1 → theProcess.out1 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    DataConnection3 :data port AccMeter.out1 → theProcess.out2 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    DataConnection4 : data port GPS.output → theProcess.out3 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    DataConnection5 : data port comVolAuto.ordre → theProcess.out4 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    DataConnection6 : data port manche.output → theProcess.out5 {
        Actual_Connection_Binding ⇒ reference thebus ;
    } ;
    BusAccessConnection1 : bus access theBus → GPS.ABussAccess ;
    BusAccessConnection2 : bus access theBus → AccMeter.ABussAccess ;
    BusAccessConnection3 : bus access theBus → ServComm.ABussAccess ;
    BusAccessConnection4 : bus access theBus → theProc.busAccess ;
    BusAccessConnection5 : bus access theBus → manche.ABussAccess ;
    BusAccessConnection6 : bus access theBus → comVolAuto.ABussAccess ;
properties
    Actual_Processor_Binding ⇒ reference theproc applies to theProcess ;
end main.impl ;
```