Conception synchrone d'applications avioniques par raffinement de modèles

Abdoulaye Gamatié, Thierry Gauthier, Paul Le Guernic

Conception synchrone d'applications avioniques par raffinement de modèles *

Abdoulaye GAMATIÉ, Thierry GAUTIER, Paul LE GUERNIC
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cédex, France
Tel : +33 2 99 84 71 00, Fax : +33 2 99 84 71 71
e-mail : {abdoulaye.gamatie, thierry.gautier, paul.leguernic}@irisa.fr

Résumé

Dans ce papier, nous abordons la conception d'applications avioniques suivant une démarche basée sur le raffinement de modèles (*model refinement*). L'étude est réalisée dans le cadre synchrone dont une particularité forte est son fondement mathématique propice aux raisonnements formels (transformations, validation...). Dans l'approche proposée, on considère d'abord une description fonctionnelle d'une application à l'aide du langage SIGNAL (modèle *polychrone*). Cette description est indépendante de toute architecture de mise en œuvre. Ensuite, des transformations préservant entièrement la sémantique des programmes manipulés sont appliquées à cette description pour en déduire une représentation reflétant une architecture dite modulaire intégrée, plus connue sous l'acronyme IMA (*Integrated Modular Avionics*).

Plan

Mots-clés

Avionique modulaire intégrée, ARINC, conception synchrone, SIGNAL, approche formelle, raffinement de modèles

*Ce travail a été financé en partie par le projet européen IST SAFEAIR (*Advanced Design Tools for Aircraft Systems and Airborne Software* - `http://www.safeair.org`).

1 Introduction

L'avionique occupe une place non négligeable dans le coût des avions modernes (par exemple, dans l'aéronautique civile, on estime que ce coût atteint jusqu'à 35% du coût total). On entend par ce terme l'ensemble des logiciels et matériels embarqués à bord de l'avion, qui assurent des fonctions telles que le traitement des informations provenant des capteurs, le pilotage automatique, la gestion du niveau de carburant, les échanges de messages avec l'opérateur au sol pendant le vol, etc.

La complexité croissante et la haute criticité des systèmes enfouis temps réel dans le domaine de l'avionique posent un certain nombre de défis quant à leur développement. Parmi ces défis on peut citer la correction des systèmes conçus vis-à-vis des exigences, l'effort de développement, la correction et la fiabilité de l'implantation, le temps de mise sur le marché du "produit". Il y a donc nécessité de méthodologies de conception adéquates, prenant en compte les défis mentionnés. D'après Pnueli [20], de telles méthodologies doivent au minimum inclure la possibilité d'avoir des spécifications formelles, de faire de la vérification de propriétés et de l'analyse. D'autre part, la génération automatique de code (éventuellement distribué) doit être possible.

Les systèmes avioniques modernes [19] [22] [4] se distinguent des systèmes traditionnels par un certain nombre de caractéristiques liées notamment à leur complexité et donc à la façon de les concevoir.

- Les fonctionnalités ne cessent d'augmenter dans les systèmes avioniques : la maintenance et les diagnostics à bord, la simulation de missions, le besoin d'autonomie, etc. D'autre part, le niveau d'intégration des fonctions pour une coopération efficace est de plus en plus élevé.
- Contrairement à l'approche traditionnelle où les fonctions sont très faiblement couplées (par exemple, le pilotage automatique et la navigation fonctionnent sur des dispositifs différents qui sont indépendants), les systèmes avioniques adoptent à l'heure actuelle une approche plutôt intégrée, où des fonctions de niveaux critiques différents peuvent s'exécuter sur un même dispositif.
- Par ailleurs, l'industrie de l'avionique utilise de plus en plus des composants matériels commerciaux qui *a priori* n'ont pas été conçus pour de tels systèmes. Si cela peut réduire les coûts de développement et augmenter les fonctionnalités, cela présente néanmoins l'inconvénient du manque de garantie suffisante de tels produits (ces derniers peuvent devenir vite obsolètes ou indisponibles).
- Les exigences de correction et de fiabilité imposent l'utilisation de méthodes formelles pour répondre à ces attentes.

Nous présentons ci-dessous les deux approches majeures reconnues dans le domaine de l'avionique pour la conception d'architectures des systèmes : d'une part, l'approche fédérée et d'autre part, l'approche modulaire intégrée.

1.1 L'approche fédérée

Usuellement dans les systèmes avioniques, chaque fonction de contrôle dispose de ses propres ressources matérielles (représentées par une machine ou un calculateur). Ces dispositifs, qui sont très souvent répliqués pour la tolérance aux fautes, peuvent varier d'une fonction à l'autre. Il en résulte ainsi une architecture hétérogène et faiblement couplée,

où chaque fonction peut opérer de manière quasi indépendante vis-à-vis des autres fonctions. Ce qui n'est pas le cas localement, où les éléments constituant une fonction doivent beaucoup coopérer pour accomplir la tâche affectée à celle-ci. Une telle architecture est qualifiée de *fédérée* [1]. Par exemple, la construction des *Airbus* A330 et A340 repose sur ce type d'architecture. Les équipements numériques mettant en œuvre certaines des fonctions à bord sont reliés par un bus mono-émetteur.

Un avantage majeur des architectures fédérées, en plus de leur simplicité, est la minimisation des risques de propagation d'erreurs qui peuvent survenir lors de l'exécution d'une fonction au sein du système. Un autre avantage des architectures fédérées est leur hétérogénéité. En effet, les types de machines utilisées peuvent varier d'une fonction à l'autre au sein du même système. Cela permet l'utilisation de machines aux puissances variables.

Malheureusement, le gros inconvénient des architectures fédérées est le risque d'utilisation abusive de ressources matérielles. Le fait que chaque fonction nécessite sa propre machine (qui plus est, répliquée pour la tolérance aux fautes) peut conduire à un coût trop élevé : d'une part, il faut suffisamment d'espace à bord de l'avion pour stocker tous les calculateurs et leur poids doit être supportable ; d'autre part, il faut en assurer l'installation et la maintenance.

1.2 L'approche modulaire intégrée

Plus récemment, une autre vision a émergé dans la conception des systèmes avioniques. Celle-ci a pour but de pallier les insuffisances des architectures fédérées, en proposant une organisation du système dans laquelle plusieurs fonctions peuvent désormais partager des ressources de calcul et de communication communes (offertes par une machine tolérante aux fautes). Ce type d'architecture est qualifié de *modulaire intégré*, plus communément appelé *IMA* (*Integrated Modular Avionics*) [1]. Des exemples d'avions adoptant cette solution sont le futur *Airbus* A380 ou le *Boeing* B777.

Cette organisation du système permet d'économiser les ressources, et par la même occasion de limiter le coût de réalisation de façon raisonnable. Cependant, elle est susceptible d'introduire une forte probabilité de propagation d'erreur qui n'existe pas dans les architectures fédérées (par exemple, une fonction en dysfonctionnement peut monopoliser le système de communication ou envoyer des commandes inappropriées, et pour chacune des fonctions il est difficile de se mettre à l'abri d'un tel comportement). La solution proposée à ce problème consiste à en un découpage fonctionnel du système qui prend en compte les ressources disponibles en temps et en mémoire. L'unité d'allocation résultant de ce découpage est appelée *partition* [1]. En pratique, le partitionnement spatial des applications repose souvent sur l'utilisation de composants matériels chargés de gérer la mémoire, pour empêcher toute corruption de zones mémoires adjacentes à une zone en cours de modification. Quant au partitionnement temporel, il dépend des fonctionnalités attendues de l'ensemble (par exemple, on aura besoin d'exécuter plus souvent des fonctions qui s'occupent du rafraîchissement de paramètres critiques).

Organisation des applications. Les applications (mettant en œuvre différentes fonctions) sont organisées de telle sorte qu'une ou plusieurs applications peuvent être regroupées sous forme de partitions, au sein d'un *module-noyau*, qui offre les ressources de

calcul nécessaires. Les partitions disposent chacune d'un espace mémoire propre et d'un budget temps pour s'exécuter. Elles sont gérées par un système d'exploitation (*module-level OS*) conforme à la norme ARINC 653 [2]. Les partitions sont quant à elles composées de *processus* représentant les entités élémentaires d'exécution (qui accomplissent la mission affectée à la partition qui les contient). Leur gestion est placée sous la responsabilité du système d'exploitation au niveau partition (*partition-level OS*). La politique d'ordonnancement adoptée à ce niveau peut varier d'une partition à une autre. Par contre, l'ordonnancement doit être préemptif et basé sur les priorités.

La communication entre partitions est essentiellement réalisée à travers des messages. Le support de base pour la transmission de ces messages est le *canal* (lien logique entre une source et une ou plusieurs destinations). L'accès aux canaux se fait via des *ports*. Tous les canaux et ports sont définis à la configuration, ce qui a pour conséquence de fixer définitivement les choix de connexions entre les ports. Le transfert de messages à travers les canaux se fait suivant deux modes : *sampling* et *queuing*. Dans le premier mode, aucune file d'attente de messages n'est autorisée. Pendant le transfert, les messages successifs contiennent des données similaires mais mises à jour. Ainsi, tout message écrit sur le port-source y reste jusqu'à ce qu'il soit transmis ou bien écrasé par une nouvelle occurrence de message. Dans ce mode, ni l'envoi, ni la réception de messages ne sont bloquants. Le mode *queuing*, à l'inverse de l'autre, autorise les files d'attente de messages gérées en FIFO (*First In First Out*), au niveau des ports reliant un canal. Ce mode garantit l'absence de perte de messages durant les transferts. On distingue trois mécanismes de communication entre processus au sein d'une partition. Le *buffer* offre le moyen d'écrire et de lire des messages rangés dans une file bornée, gérée en FIFO. L'*event* permet de notifier l'occurrence d'un événement à des processus en attente de celui-ci. Enfin, le *blackboard* permet d'écrire et de lire des messages dans un tampon à une place : un message reçu dans le tampon y reste jusqu'à ce qu'il soit écrasé par une nouvelle écriture de message ou bien que le *blackboard* soit "nettoyé". La synchronisation entre les processus est effectuée à l'aide de *sémaphores* à compteur.

Le standard APEX-ARINC 653 [2] décrit une interface générique entre la couche applicative et le système d'exploitation dans une architecture IMA. Cette interface, appelée APEX (APplication EXecutive), permet de contrôler l'exécution des partitions et processus, y compris la gestion des communications et synchronisations. Elle offre aussi des services permettant la gestion du temps et des erreurs.

Dans la suite, nous décrivons à l'aide du langage synchrone SIGNAL [18] (section 2) les concepts de base nécessaires à la modélisation d'applications selon l'architecture IMA. Nous proposons dans la section 3 une méthodologie de conception descendante (par raffinements), utilisant les éléments modélisés en section 2. Enfin, une conclusion est donnée en section 4.

2 Description de concepts IMA en SIGNAL

Nous présentons ici la modélisation des notions introduites dans la section 1.2 pour décrire des applications avioniques suivant une architecture IMA [8, 7, 10, 11] (c'est-à-dire, partitions, processus et services APEX). Nous partons du fait qu'un modèle exécutable

d'une partition (représentant une application) peut être représenté comme sur la FIG. 1. Trois aspects y sont principalement distingués :

1. les *entités d'exécution*, représentées par les processus ;
2. les *échanges* entre ces entités (communications et synchronisations entre processus), réalisés au moyen de services APEX ;
3. l'*exécution concurrente* des processus au sein de la partition, placée sous le contrôle de l'exécutif temps réel (représentant le *partition-level OS*).

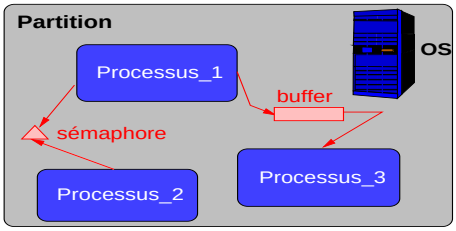


FIGURE 1 – Modèle exécutable d'une partition.

Dans ce qui suit, nous présentons la démarche de description des services APEX (section 2.1). Nous donnons ensuite un modèle de l'exécutif temps réel, qui repose en majeure partie sur l'utilisation de ces services (section 2.2). Enfin, un modèle est proposé pour les processus d'une partition (section 2.3).

2.1 Description des services APEX

Pour illustrer notre démarche, nous considérons le cas particulier d'un service APEX, appelé *read_blackboard*, qui permet de lire un message dans un *blackboard*. La description des modèles des autres services repose sur le même principe.

Nous partons d'abord d'une spécification SIGNAL très abstraite du service (cf. FIG. 2), qui sera détaillée progressivement. En procédant ainsi, nous obtenons des descriptions successives jusqu'à ce que le niveau de détail souhaité soit atteint (par exemple, jusqu'à ce que le fonctionnement interne soit entièrement explicité).

Sur la description SIGNAL de la FIG. 2, deux entrées (repérées par le symbole "`?`") sont distinguées : l'identificateur du *blackboard* représenté par `board_ID` et une quantité de temps dénotée par `timeout`. Cette dernière entrée indique, lors d'une requête utilisant le service, la durée d'attente du processus appelant dans le cas où il n'y aurait aucun message à lire. Parmi les paramètres de sortie (repérés par le symbole "`!`"), on note l'adresse et la taille du message lu, respectivement représentées par `message` et `length`. La sortie `return_code` indique quant à elle le diagnostic d'une requête.

Cette description abstraite exprime un certain nombre de propriétés, par exemple `(s.2)` décrit les instants de présence d'un code retour. En SIGNAL, l'équation `x1 ^= x2` spécifie que les signaux `x1` et `x2` sont présents aux mêmes instants logiques (ils sont dits aussi *synchrones*). Ainsi, le code retour est reçu uniquement lorsqu'un certain signal lo-

```
process READ_BLACKBOARD =
  { ProcessID_type process_ID; }
  ( ? Comm_ComponentID_type board_ID;
      SystemTime_type timeout;
    ! MessageArea_type message;
      MessageSize_type length;
      ReturnCode_type return_code; )
 (| (| { {board_ID, timeout} --> return_code } when C_return_code          (d.1)
    | { {board_ID, timeout} --> {message, length} }
            when (return_code = #NO_ERROR)                                 (d.2)
    |)
  | (| board_ID ^= timeout ^= C_return_code                                 (s.1)
    | return_code ^= when C_return_code                                    (s.2)
    | message ^= length ^= when (return_code = #NO_ERROR)                  (s.3)
    |)
  |) where boolean C_return_code;
```

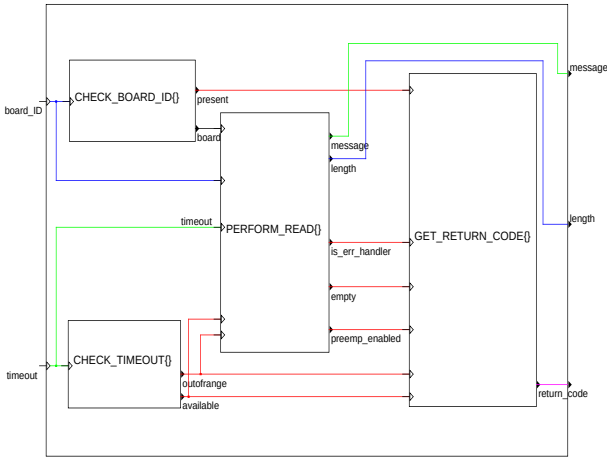
FIGURE 2 – Description abstraite du service *read_blackboard*.

cal `C_return_code`, de type booléen, porte la valeur *vrai* [1]. Ce booléen apparaît pour l'instant dans l'abstraction comme un signal local dont la définition sera explicitée lors des raffinements. Cela peut être interprété comme "il existe un signal `C_return_code` tel que les propriétés exprimées dans l'abstraction sont satisfaites". Dans `(s.1)`, on spécifie que ce signal est synchrone avec les entrées du service (i.e. chaque fois qu'il y aura une demande de lecture, `C_return_code` indiquera si un code retour est renvoyé ou non). La propriété `(s.3)` exprime le fait que les messages ne sont récupérés que lorsque le code retour vaut *NO_ERROR*. Les lignes `(d.1)` et `(d.2)` donnent des relations de dépendance entre les entrées et les sorties. En SIGNAL, la notation `x --> y` exprime une relation de dépendance entre deux signaux `x` et `y` à l'intérieur d'un même instant logique (on dit que `x` *précède* `y` à cet instant). Par exemple dans la propriété `(d.2)`, les signaux `timeout` et `board_ID` précèdent les signaux de sortie `message` et `length` lorsque le diagnostic de la requête est `NO_ERROR`.

Le niveau de détail offert par une description telle que celle de la FIG. 2 suffit par exemple pour vérifier la conformité d'un composant ainsi défini, lors de son intégration au sein d'un système décrit dans le même formalisme. Ici, le programme spécifie des propriétés d'interface du service *read_blackboard*. Il précise notamment la condition sous laquelle un message est renvoyé lors d'une requête utilisant le service (i.e., **quand** un message peut être récupéré suite à une demande de lecture). Par contre, il ne décrit pas

1. Lors de l'appel du service *read_blackboard*, la réception d'un code retour n'est pas forcément immédiate. Par exemple, lorsque le *blackboard* est vide et que la valeur du paramètre d'entrée `timeout` n'est pas l'infini (identifié dans [2] par une constante *INFINITE_TIME_VALUE*), le processus appelant doit attendre : soit l'écriture d'un message dans le *blackboard*, soit l'expiration du compteur de temps déclenché (le code retour prend alors la valeur *TIMED_OUT*). Le booléen `C_return_code` est introduit ici pour représenter les instants logiques auxquels un code retour est produit à la suite d'un appel. Ce signal sera défini ultérieurement lorsqu'on affinera la description du service.

exactement **comment** le message est récupéré.




```
(| board_ID ^= timeout ^= present ^= outofrange ^= available ^=
     C_return_code
 | board ^= empty ^= when present
 | message ^= length ^= when (not empty)
 | is_err_handler ^= when empty when available
 | preemp_enabled ^= when (not is_err_handler)
 | C_return_code := (when ((not present) or outofrange)) default
                    (when empty when (not available)) default
                    (when ((not preemp_enabled) default is_err_handler))
                   default (when (not empty)) default false
 | return_code ^= when C_return_code
 |)
```

FIGURE 3 – Raffinement de *read_blackboard* et relations d'horloges entre les signaux mis en évidence.

Un inconvénient des spécifications qu'on trouve dans [2] est quelque fois leur manque de précision, d'où des ambiguïtés qui ne sont pas forcément perceptibles de prime abord. En l'occurrence, il existe deux mises en œuvre possibles du service *read_blackboard*. Elles sont dues ici aux différentes interprétations possibles pour la valeur du message récupéré par un processus qui se trouvait précédemment bloqué en attente de lecture de message sur un *blackboard* :

1. Certaines implémentations peuvent considérer que ce message est *celui qui vient d'être écrit* dans le *blackboard* par le processus qui provoque le déblocage du processus en attente.
2. Pour d'autres implémentations, il s'agit plutôt du message présent dans le *blackboard* au moment où le processus débloqué reprend son exécution. Ce qui veut dire qu'il ne recevra pas forcément le message du processus à l'origine de sa libération, puisqu'entre-temps il peut y avoir eu d'autres messages dans le *blackboard*.

Ces ambiguïtés justifient la nécessité d'une démarche comme celle qui est adoptée ici, où les descriptions abstraites permettent d'avoir des spécifications assez générales qu'on pourra raffiner selon l'interprétation voulue. Comme nous pouvons le remarquer, le programme SIGNAL de la FIG. 2 englobe les deux interprétations du service *read_blackboard*. Il ne précise pas la valeur d'un message reçu lors d'une lecture. Il indique seulement la condition sous laquelle ce message peut être lu. La description donnée sur la FIG. 3 explicite le modèle précédent du service.

Les sous-programmes `CHECK_BOARD_ID` et `CHECK_TIMEOUT` vérifient la validité des paramètres d'entrée `board_ID` et `timeout`. Lorsque ces derniers sont valides, `PERFORM_READ` tente alors de lire un message dans le *blackboard* spécifié. Après quoi, il renvoie le dernier message disponible dans le *blackboard* (son adresse mémoire et sa taille sont respectivement spécifiées par `message` et `length`) et transmet toutes les informations nécessaires à `GET_RETURN_CODE` qui doit déterminer le diagnostic de la requête.

Chacun des sous-programmes distingués dans le modèle de la FIG. 3 est à son tour explicité en adoptant la même démarche que pour le modèle global du service. La description finale (basée sur la seconde interprétation de *read_blackboard* - voir discussion ci-dessus) résultant de ce processus de raffinement peut être trouvée dans [6].

La modélisation des autres services APEX suit une démarche analogue à celle adoptée pour le service *read_blackboard*. À l'aide de ces services, nous pouvons à présent décrire la gestion des processus ainsi que les communications et synchronisations entre ceux-ci. La section suivante présente la modélisation de l'exécutif temps réel (c'est-à-dire le *partition-level OS*), qui est chargé de contrôler l'exécution des processus au sein d'une partition.

2.2 Modélisation de l'exécutif temps réel

Les notions prises en compte pour modéliser le fonctionnement du *partition level OS* sont essentiellement : la *gestion des tâches* (par exemple, création, suspension ou changement de priorité d'une tâche), l'*ordonnancement* (incluant la définition des descripteurs de tâches et de l'ordonnanceur), la *gestion du temps* (par exemple, la mise à jour de compteurs de temps) et les *communications* et *synchronisations* entre tâches.

L'interface APEX comprend une grande partie des services réalisant ces notions. C'est le cas notamment des services de gestion de processus, des services de communication et synchronisation et des services de gestion du temps. Néanmoins, cela est insuffisant pour avoir une description complète des fonctionnalités du *partition level OS*. Ainsi, nous avons ajouté à notre bibliothèque des services supplémentaires permettant de décrire l'ordonnancement des processus au sein d'une partition et la mise à jour de compteurs de temps. La description et l'utilisation de ces services pourra être trouvée dans [10] et [12]. Ici, nous nous limiterons à la présentation de l'interface du *partition level OS* (cf. FIG. 4) pour expliciter la façon dont celui-ci interagit avec le reste des objets modélisés dans la partition, à savoir les processus.

Sur la FIG. 4, l'entrée `Active_partition_ID` représente l'identificateur de la partition active, sélectionnée par l'OS de niveau supérieur, c'est-à-dire le niveau mo-

FIGURE 4 – Interface de l'exécutif temps réel au sein de la partition.

dule [2]. Elle dénote un ordre d'exécution lorsqu'elle identifie la partition courante. La réception du signal `initialize` correspond à la phase d'initialisation de la partition : création de l'ensemble des mécanismes et processus contenus dans celle-ci. Les entrées `Active_partition_ID` et `initialize` sont externes à la partition : elles proviennent du niveau supérieur (c'est-à-dire le module, non représenté ici).

Le signal `dt` indique la durée d'exécution des instructions effectuées par le processus actif de la partition (on y reviendra dans la section 2.3). C'est une information produite par le processus, qui est destinée à mettre à jour l'ensemble des compteurs de temps utilisés dans la partition. Le *partition-level OS* est responsable de cette mise à jour. Contrairement aux signaux `Active_partition_ID` et `initialize`, `dt` est interne à la partition.

Le signal `Active_process_ID` identifie le processus actif désigné à chaque réordonnancement dans la partition. Il est déterminé en fonction de la politique d'ordonnancement. Il est envoyé à tous les processus de la partition.

Enfin, le signal `timedout` (un tableau de booléens) indique aux processus si oui ou non le compteur de temps qu'ils auraient déclenché est arrivé en butée : pour chaque processus pid, `timedout`$[pid]$ vaut *vrai* si le compteur de temps associé à pid est arrivé en butée, sinon il vaut *faux*. Ce signal est émis par le *partition level OS* en direction des processus.

2.3 Modélisation des processus

Les processus représentent les entités élémentaires d'exécution au sein d'une application définie selon une architecture de type IMA. Pour modéliser un processus, nous le décomposons en fragments appelés *blocs*. Ces derniers représentent les actions atomiques effectuées par le processus. Leur exécution est instantanée et le processus ne peut être interrompu qu'entre l'exécution de deux blocs. Le fonctionnement global du modèle d'un processus est décrit ainsi par l'enchaînement d'exécutions de blocs résultant de son découpage.

Concrètement, la démarche de construction du modèle repose sur une idée très simple et bien connue dans la conception des systèmes en général : *séparation du contrôle et du calcul*. Par exemple, c'est ce qu'on peut observer dans la *conception matérielle*, où un système est composé d'unités mettant en œuvre sa combinatoire (c'est-à-dire, sa partie contrôle) d'une part, et ses calculs d'autre part. Le modèle d'un processus est constitué ainsi de deux sous-parties appelées *CONTROL* (qui représente le flot de contrôle) et

2. À l'instar du modèle de processus, l'activation de chaque partition dépend de l'entrée *Active_partition_ID*, qui identifie la partition active courante. C'est un signal qui est fourni par le *module-level OS* qui est en charge de la gestion des partitions dans un module.

COMPUTE (l'ensemble des actions exécutées). C'est ce qui est illustré sur la FIG. 5.

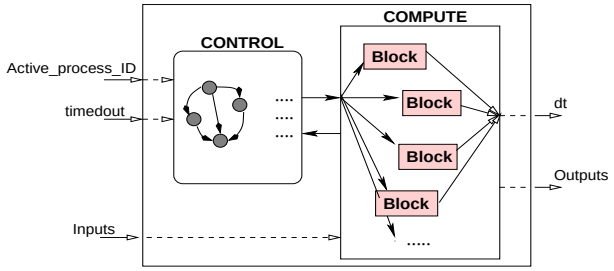


FIGURE 5 – Modèle générique d'un processus.

Le processus peut être vu comme un composant réactif dans lequel, chaque fois qu'un ordre d'activation est reçu de l'ordonnanceur, des actions sont sélectionnées pour être exécutées. Ainsi, l'entrée `Active_process_ID` identifie le processus actif dans la partition. Ce signal est reçu par tous les processus, qui en consultent la valeur de façon à déterminer s'ils peuvent ou non s'exécuter : il joue donc le rôle d'un ordre d'activation. L'entrée `timedout` quant à elle, sert à notifier à un processus bloqué sur l'appel d'un service APEX avec *time-out* (*read_blackboard*, par exemple), l'expiration du délai spécifié. La sortie `dt` dénote la quantité de temps écoulée à chaque exécution d'actions par un processus. Cette information est utilisée au sein du *partition-level OS* pour la mise à jour des compteurs de temps. Les trois signaux mentionnés ci-dessus (`Active_process_ID`, `timedout` et `dt`) sont communs à toutes les interfaces de modèles de processus. En plus de ceux-là, il faut distinguer les signaux qui représentent les entrées (resp. sorties) requises pour (resp. produites par) les calculs effectués dans la partie COMPUTE, qui varient d'un processus à l'autre.

Le composant CONTROL décrit le flot d'exécution du processus. Typiquement, il est représenté par un système de transition qui, selon son état courant indique les actions à exécuter lorsque le processus est actif. La spécification d'automates en SIGNAL se fait de façon assez naturelle et simple (elle pourrait être produite automatiquement à partir d'une description graphique). Le composant COMPUTE quant à lui, décrit les actions exécutées par le processus. Il est composé de *blocs* qui sont des fragments de code exécutés sans interruption (i.e de façon atomique). Dans le modèle, leur exécution est instantanée (temps logique). Cela requiert bien entendu quelques précautions à prendre quant à la façon de découper le code à exécuter dans les blocs [6]. Ainsi, à la fin de chaque exécution de bloc, un ré-ordonnancement de processus est systématiquement effectué pour désigner le prochain processus à activer. C'est à ce niveau que nous choisissons [3] de faire intervenir la préemption. Enfin, la durée d'exécution de chaque bloc du processus actif est obtenue via son signal de sortie `dt`. Cette quantité de temps résulte d'une estimation obtenue avant ou pendant l'exécution. On peut utiliser pour cela des techniques de calcul des durées d'exécution au pire cas [5], ou une technique de *profiling* basée sur SIGNAL [17], comme cela est illustré dans [12].

3. Selon la norme ARINC 653, tout processus est susceptible d'être préempté à tout moment par un autre processus de priorité plus grande et prêt à s'exécuter.

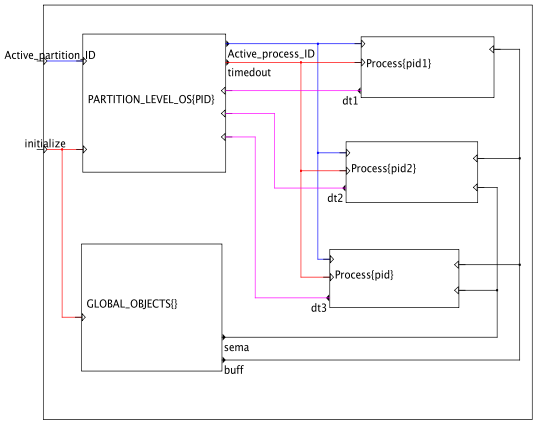


FIGURE 6 – Un exemple de modèle de partition.

La FIG. 6 illustre une vue globale d'un modèle de partition contenant trois processus. Deux mécanismes de communication et de synchronisation (respectivement représentés par les signaux `buff` et `sema`) sont créés et initialisés au sein de la partie `GLOBAL_OBJECTS`. Cette dernière apparaît ici pour des raisons de structuration afin de faciliter la compréhension du modèle.

En résumé, nous avons défini une bibliothèque SIGNAL de services APEX, ainsi que des modèles correspondant aux processus et partitions ARINC. Ces éléments permettent de décrire une application avionique, suivant l'architecture modulaire intégrée. L'utilisation de ces composants s'inscrit dans une méthodologie de conception par raffinements. Celle-ci prône une démarche basée sur une modélisation en langage SIGNAL. Ainsi, les transformations effectuées peuvent être prouvées valides dans le modèle sémantique du langage. La section suivante présente cette méthodologie.

3 Méthodologie de conception par raffinement de modèles suivant IMA

Par raffinement, nous entendons un ensemble de transformations permettant de définir progressivement, à partir d'une description initiale `P` (un programme SIGNAL), des descriptions transformées de la manière suivante : à chaque étape, on obtient une nouvelle description `Q`, qui résulte de la définition de nouvelles variables intermédiaires en ajoutant des équations à `P`. Ce raffinement peut modifier les propriétés non fonctionnelles (plus précisément, temporelles) de `P`. Par contre, il préserve les propriétés fonctionnelles.

Ainsi, dans la démarche de conception présentée ci-après, l'utilisation des modèles de composants ARINC introduits à la section précédente intervient après un certain nombre de transformations du programme SIGNAL initial représentant une application indépendamment d'une architecture particulière de mise en œuvre. Nous décrivons d'abord ces transformations. Ensuite, nous montrons comment est réalisé le passage à une description

de type IMA.

3.1 Transformations préliminaires de programmes

Les notions introduites ci-après sont issues de travaux réalisés lors du projet européen SACRES [3] [13]. Le but était de définir des schémas de génération de code distribué à partir de spécifications synchrones, et plus particulièrement, de programmes décrits en SIGNAL.

On considère les hypothèses suivantes :

1. les programmes considérés sont initialement *endochrones* [18], donc déterministes (pour simplifier, un programme endochrone peut être exécuté de façon autonome dans un environnement donné) ;
2. ils ne contiennent aucune définition circulaire ;
3. il existe une fonction *allouer*, qui définit la répartition du modèle fonctionnel d'une application sur celui d'une architecture matérielle (l'allocation peut être manuelle ou automatique).

On dispose d'un ensemble fini de processeurs $q = \{q_1, q_2, ..., q_m\}$, et d'une fonction $allouer : \{P_i\} \longrightarrow \mathcal{P}(q)$, qui affecte chaque programme P_i à un ensemble non vide de processeurs, avec possibilité de répliquer du code.

Enfin, pour présenter les transformations, nous considérons des programmes SIGNAL de la forme $P = P_1 \mid P_2 \mid ... \mid P_n$, où chaque sous-programme P_i peut lui-même être composé récursivement d'autres sous-programmes (i.e., $P_i = P_{i1} \mid P_{i2} \mid ... \mid P_{im}$).

Première transformation. Soit un programme SIGNAL $P = P_1 \mid P_2$, illustré sur la FIG. 7. Chaque sous-programme P_i (représenté ici par un cercle) est lui-même composé de quatre sous-programmes P_{i1}, P_{i2}, P_{i3} et P_{i4}. On souhaite répartir P sur deux processeurs, q_1 et q_2, suivant la fonction d'allocation définie comme suit :

$$\begin{array}{ll} \forall i \in \{1,2\}\ \forall k \in \{1,2\}, & allouer(P_{ik}) = \{q_1\}, \quad et \\ \forall i \in \{1,2\}\ \forall k \in \{3,4\}, & allouer(P_{ik}) = \{q_2\} \end{array}$$

Le programme P peut alors se récrire en : $P = Q_1 \mid Q_2$, où

$$\begin{array}{ll} Q_1 = & P_{11} \mid P_{12} \mid P_{21} \mid P_{22}, \quad et \\ Q_2 = & P_{13} \mid P_{14} \mid P_{23} \mid P_{24} \end{array}$$

Les sous-programmes Q_1 et Q_2, issus de ce partitionnement de P, sont appelés *s-tasks* [13]. Cette transformation fournit une description multi-processeur strictement équivalente au programme initial d'un point de vue sémantique (i.e. ensembles de comportements identiques) car il s'agit d'une simple récriture de programme.

Seconde transformation. On souhaite raffiner le niveau de granularité de la décomposition effectuée lors de la première transformation. Pour cela, on se place au niveau de chaque processeur et on procède à un partitionnement des *s-tasks*. Celles-ci sont vues alors comme un ensemble de *nœuds*, représentant les entités d'exécution atomique. Deux choix sont envisageables concernant le niveau de granularité de ces nœuds : soit tout

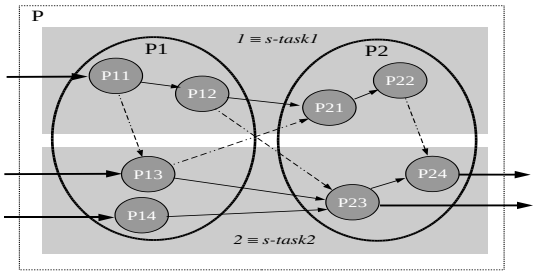


FIGURE 7 – Décomposition d'un programme SIGNAL P en deux *s-tasks*.

nœud consiste en une construction primitive de SIGNAL ; soit un nœud peut être constitué de plusieurs constructions primitives. On obtient une description très fine en adoptant la première possibilité. Cependant, la seconde peut être plus judicieuse du point de vue de l'exécution car davantage d'actions peuvent être effectuées de façon atomique. Une façon efficace de construire de tels nœuds est de choisir comme critère le fait que les expressions regroupées au sein d'un nœud dépendent d'un même ensemble d'entrées. Cela repose essentiellement sur une analyse de *sensitivité*, effectuée sur le programme SIGNAL.

Nous dirons qu'il y a un chemin de causalité entre deux nœuds n_1 et n_2 s'il existe au moins une situation où l'exécution de n_2 dépend de celle de n_1 (provoquant, par la même occasion, l'exécution des éventuels nœuds intermédiaires).

Definition 1 *Deux nœuds N_1 et N_2 sont sensitivement équivalents ssi pour chaque entrée e, les propriétés 1 et 2 sont équivalentes :*

1. *il existe un chemin de causalité entre e et N_1,*
2. *il existe un chemin de causalité entre e et N_2.*

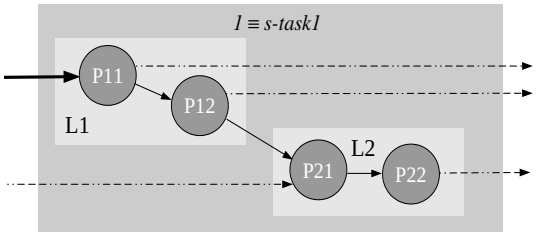


FIGURE 8 – Décomposition de Q_1 en deux nœuds.

Des nœuds sensitivement équivalents peuvent donc être regroupés au sein d'un même sous-ensemble de nœuds qui est exécutable de façon atomique : ces sous-ensembles sont appelés *lignées*. Une propriété importante de celles-ci est que toute entrée précède toute sortie. Si le graphe de départ est endochrone, chaque lignée l'est aussi, ce qui rend son exécution déterministe. La FIG. 8 montre la décomposition de Q_1 en deux lignées $L1$ et L_2. L'entrée du sous-programme $P11$ (flèche en gras) est à l'origine une entrée de P.

Les autres flèches mises en évidence sur le schéma sont des communications locales à P. Elles représentent des signaux échangés par les *s-tasks*. Nous pouvons remarquer qu'après cette seconde transformation, l'équivalence sémantique stricte du programme initial et du programme résultant est toujours préservée.

Les deux transformations présentées ci-dessus décrivent un partitionnement de programmes SIGNAL selon une architecture générique multi-tâche multi-processeur. L'instanciation d'une telle représentation dans le modèle IMA consiste à utiliser les composants définis à la section 2 (services APEX, processus, partitions) pour décrire les concepts correspondants.

3.2 Instanciation de programmes SIGNAL dans le modèle IMA

Pour aborder la modélisation à l'aide des composants ARINC, nous nous plaçons d'abord au niveau d'un processeur. La démarche pourra ensuite être généralisée à un ensemble quelconque de processeurs. Ainsi, d'après les transformations ci-dessus, un processeur est vu comme un graphe où les nœuds sont des lignées. Le partitionnement d'un programme SIGNAL suivant le modèle d'architecture IMA est réalisable à travers les étapes suivantes :

Étape 1 : Transformation d'un programme SIGNAL en graphe de lignées. Cette tâche est réalisable de façon automatique grâce au compilateur.

Étape 2 : Regroupement des lignées en modèles de partitions et de processus. La transformation du graphe de lignées associé à un processeur commence par le choix d'une représentation en termes de partitions IMA. Il s'agit précisément d'identifier les lignées (i.e. les sous-ensembles du graphe) qui vont s'exécuter au sein d'une même partition. On rappelle qu'une partition s'exécute sur un processeur à la fois, et celui-ci peut se voir affecter plusieurs partitions [2]. Dans notre exemple, on décide de modéliser le graphe associé à $Q1$ (cf. FIG. 8) par une seule partition. Les partitions étant choisies, le graphe correspondant à chacune d'entre elles est à son tour décomposé en sous-graphes. Ces derniers contiennent les lignées devant s'exécuter dans un même processus. Sur notre exemple simple, les lignées associées à la "partition $Q1$" forment l'ensemble des blocs d'instructions d'un seul processus. Le partitionnement d'un graphe de lignées en partitions et processus se fait de façon *a priori* arbitraire. Cependant, il est plus judicieux de regrouper au sein d'une même partition les lignées qui dépendent fortement les unes des autres. On procède alors à l'instanciation à proprement parler dans le modèle IMA.

Étape 3 : Instanciation dans le modèle IMA. On procède en deux phases : on instancie d'abord les processus et ensuite, on fait de même avec les partitions. Les éléments de notre modélisation IMA sont rappelés brièvement sur la FIG. 9. Le symbole "|" dénote la composition synchrone. On définit les transformations suivantes :

1. Description du processus associé à un ensemble de lignées :
   - La partie *CONTROL* du processus associé est construite en se basant sur les dépendances entre les lignées. On choisit un ordonnancement basé sur une exécution séquentielle des lignées ;
   - Chaque lignée est "plongée" dans un bloc au sein d'un processus ;

$$
\begin{array}{rcll}
 & & p - OS & (\textit{OS} \text{ de niveau partition}) \\
 & & cont_i & (\text{partie } \textit{control} \text{ d'un processus } p_i) \\
 & & b_{ij} & (\textit{bloc}) \\
 & & & \\
p & ::= & p - OS \mid p_1 \mid ... \mid p_n & (\textit{partition}) \\
p_i & ::= & cont_i \mid comp_i & (\textit{processus}) \\
comp_i & ::= & b_{i1} \mid ... \mid b_{im_i} & (\text{partie } \textit{compute} \text{ d'un processus } p_i)
\end{array}
$$

FIGURE 9 – Éléments de la modélisation IMA.

- Les communications internes entre les lignées d'un même sous-graphe associé à un processus sont modélisées à l'aide de variables d'état (i.e. définies à l'aide de la construction primitive de SIGNAL appelée *retard*), locales au processus. Elles servent à mémoriser les données échangées. En ce qui concerne les communications entre sous-graphes de lignées, on utilise les services APEX. Il s'agit de communications entre processus. Pour chaque entrée (resp. sortie) d'un sous-graphe, on ajoute dans la partie *COMPUTE* du processus associé, un bloc contenant un appel à un service de communication ou de synchronisation qui lui correspond. Lorsque le processus devient actif, ce bloc doit être exécuté juste avant (resp. après) le bloc qui contient la lignée concernée par l'entrée (resp. la sortie). Le choix des appels dépend ici du type de communication souhaitée. Par exemple, si cette dernière utilise une file bornée de messages, on préférera les services d'un *buffer* à ceux d'un *blackboard*, qui est approprié pour des échanges de messages via un tampon à une place. Quant aux appels servant pour la synchronisation, on utilisera les services associés au sémaphore.

2. Description de la partition associée à un ensemble de lignées :
   - On ajoute aux processus définis ci-dessus le composant correspondant au système d'exploitation de niveau partition (contenant entre autres l'ordonnanceur, qui détermine l'ordre d'exécution des processus) ;
   - On crée les mécanismes de communication et de synchronisation utilisés dans les services APEX ajoutés au sein des processus (par exemple, pour un appel du service *send_buffer*, il faut créer le *buffer* dans lequel les messages sont stockés). Cette création peut se faire par exemple dans la partie `GLOBAL_OBJECTS` de la partition, comme sur la FIG. 6.

Exemple. *Un modèle de processus résultant de la transformation de $Q1$ est représenté de façon non détaillée sur la* FIG. *10. Il comprend six blocs dont deux qui contiennent les lignées $L1$ et $L2$. Les autres blocs ont été ajoutés pour les communications : r, s et w dénotent respectivement une lecture (par exemple, receive_buffer ou read_blackboard), une notification d'un événement (comme set_event), et une requête d'attente d'un événement (par exemple, wait_event). L'automate de la partie contrôle décrit un ordre d'exécution des blocs, qui respecte les contraintes de précédence entre les lignées du graphe.*

Pour obtenir la partition correspondante, nous renvoyons au point 2 de l'étape 3, dans cette section.

Sur un processeur où il y a plusieurs partitions, on adopte la même démarche pour

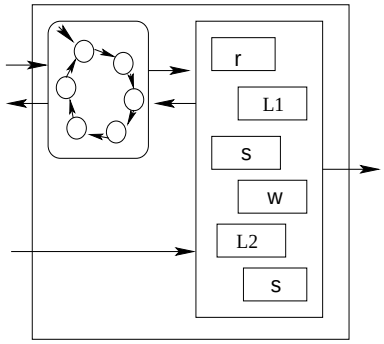


FIGURE 10 – Modèle de processus associé à la *s-task* $Q1$.

chacune d'entre elles. Un modèle d'ordonnanceur de partitions est nécessaire au niveau du processeur. La politique de gestion à adopter est alors basée sur du temps partagé (comme dans une politique *round robin*). On ajoutera donc aux partitions un composant correspondant au système d'exploitation de niveau module. Ce dernier est similaire à l'OS de niveau partition dans le sens où sa définition repose sur les services APEX. Par contre au sein du module, l'ordonnancement est basé sur une politique à temps partagé, au lieu d'une politique préemptive comme c'est le cas dans une partition.

4 Conclusion et travaux liés

Nous avons présenté une méthodologie de conception descendante, par raffinement de modèles, d'applications avioniques selon une architecture modulaire intégrée (IMA). Cette méthodologie repose d'une part, sur des transformations systématiques de programmes, et d'autre part, sur l'utilisation de composants décrits en SIGNAL, à partir des spécifications de la norme avionique APEX-ARINC. Pour une application donnée, on considère initialement une spécification SIGNAL de celle-ci, indépendamment de tout choix de mise en œuvre. Ensuite, après partitionnement et placement de l'application sur une architecture multi-processeur, des transformations préservant la sémantique des programmes sont effectuées sur chacune des parties. Enfin, des modèles de composants ARINC sont utilisés pour raffiner les descriptions résultantes sur chaque processeur, selon l'architecture IMA.

Au-delà du fait que notre démarche promeut une activité de conception dans un cadre formel (pour faciliter la validation notamment), elle apporte des éléments de solution au délicat problème du partitionnement dans les systèmes avioniques. Actuellement, dans ces systèmes (par exemple, l'Airbus A380), peu de fonctions de niveau critique élevé adoptent une architecture modulaire intégrée, à cause des difficultés liées à leur partitionnement. Cela fait que la solution fédérée est encore attractive. L'analyse de sensitivité décrite ici permet d'identifier les dépendances entre différentes parties d'une fonction avionique à décomposer en partitions IMA. Elle favorise ainsi des regroupements judicieux de parties au sein des partitions (par exemple, des parties fortement dépendantes iraient dans une même partition).

Parmi les travaux existants comparables à notre étude, nous mentionnons d'abord l'approche SYNDEX [14]. Celle-ci permet d'obtenir la distribution et l'ordonnancement d'une application sur une architecture matérielle multi-processeur en se basant sur des critères de performance de l'architecture cible (temps de réponse déterminé à partir d'un graphe logiciel de l'application, valué par des temps d'exécution des processus sur le type de processeur considéré, et les durées des communications). L'approche SYNDEX peut ainsi être qualifiée de "quantitative", tandis que la nôtre est plutôt "qualitative". D'une part, la répartition de l'application sur les processeurs est sous la responsabilité du concepteur (en général, un expert qui connaît le type de processeur qui convient pour l'exécution efficace des parties de l'application). Ensuite, au niveau de chaque processeur, une analyse de sensitivité est effectuée afin de dégager les différentes parties représentables à l'aide des modèles d'entités IMA (cf. section 3.1). Les deux approches peuvent cependant être vues comme complémentaires au sein d'une méthodologie. En effet, on pourrait considérer d'abord une répartition de l'application utilisant des critères quantitatifs ; ensuite, l'analyse de sensitivité serait appliquée localement à chaque processeur.

D'autres approches auxquelles la nôtre peut être comparée sont celles prônées par AUTOFOCUS [16] et GIOTTO [15]. Elles utilisent chacune un modèle proche du modèle synchrone pour concevoir des systèmes embarqués distribués. Dans AUTOFOCUS [16], parmi les éléments de base, on distingue des *composants* qui communiquent par l'intermédiaire de *ports* reliés par des *canaux*. Ces éléments, qui permettent de donner une description structurelle d'un système, constituent les objets de base des *diagrammes de structure du système*. La description comportementale d'un composant est exprimée au moyen de *diagrammes à états-transitions*. Ce sont des machines à états hiérarchiques comprenant un ensemble d'états de contrôle, des transitions et des variables locales. Les interactions entre composants sont données en termes de traces d'événements qui se rapprochent des diagrammes de séquences UML. Dans AUTOFOCUS, la sémantique comportementale est *synchrone*. On considère une horloge globale suivant laquelle les modèles évoluent. Cependant, une particularité est que les calculs effectués au sein des composants et les communications entre ces composants se déroulent de façon exclusive. La description d'un système distribué dans AUTOFOCUS prend en compte deux aspects : le modèle *logique* qui représente le graphe logiciel, et le modèle *technique* qui symbolise l'architecture l'implantation. La description finale du système est obtenue en projetant le modèle logique sur le modèle technique. Pour cela, la sémantique est étendue avec une interprétation GALS (Globalement Asynchrone Localement Synchrone) [21] : la sémantique du modèle technique n'est pas synchrone ; pour chaque canal logique qui doit être projeté sur un canal technique, on supprime les synchronisations entre les composants qu'il relie, et on rajoute des *buffers* et un ordonnanceur global pour ces canaux. Une fois le modèle final obtenu, on peut faire de la vérification de propriétés (en utilisant des outils comme le *model checker* SPIN), ou bien de la simulation. AUTOFOCUS permet aussi de faire de la synthèse de séquences de tests. Quant à la méthodologie proposée par GIOTTO [15], elle repose sur une sémantique *time-triggered* dont l'avantage est la prédictibilité des comportements temporels du système décrit. Un modèle GIOTTO d'une application est donné sous forme de tâches avec un ordonnanceur. Il est donc proche de celui que nous avons défini. Cependant, les tâches sont essentiellement périodiques.

La méthodologie que nous avons présentée est mise en œuvre dans la plate-forme PO-

LYCHRONY [4], qui offre un outillage de transformation et d'analyse de modèles associé au langage SIGNAL. Une bibliothèque de modèles de composants ARINC [8, 9] est fournie avec l'environnement.

Références

[1] Airlines Electronic Engineering Committee. ARINC report 651-1 : Design guidance for integrated modular avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, Novembre 1997.

[2] Airlines Electronic Engineering Committee. ARINC specification 653 : Avionics application software standard interface. In *Aeronautical radio, Inc., Annapolis, Maryland*, Janvier 1997.

[3] A. Benveniste. Safety critical embedded systems : the SACRES approach. In *proceedings of Formal techniques in Real-Time and Fault Tolerant Systems, FTRTFT'98 school, Lyngby, Denmark*, Septembre 1998.

[4] D. Cofer and M. Rangarajan. Formal modeling and analysis of advanced scheduling features in an avionics RTOS. In *procedings of Conference on Embedded Software (EMSOFT'02), J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, p. 138-152*, 2002.

[5] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI), 22(5)*, pages 651–677, 2003.

[6] Abdoulaye Gamatié. Modélisation polychrone et évaluation de systèmes temps réel. In *Thèse de l'Université de Rennes I, IFSIC, France*, mai 2004.

[7] Abdoulaye Gamatié and Thierry Gautier. Modeling of modular avionics architectures using the synchronous language signal. In *Proceedings of the Work In Progress session, 14th Euromicro Conference on Real Time Systems, ECRTS'02*, pages 25–28, 2002.

[8] Abdoulaye Gamatié and Thierry Gautier. Synchronous modeling of modular avionics architectures using the SIGNAL language. In *Rapport de recherche INRIA numéro 4678*, Décembre 2002. Accessible à l'adresse `http://www.inria.fr/rrrt/rr-4678.html`.

[9] Abdoulaye Gamatié and Thierry Gautier. The SIGNAL approach to the design of system architectures. In *10th International Conference and Workshop on the Engineering of Computer-based Systems, Huntsville - Alabama*, April 2003.

[10] Abdoulaye Gamatié and Thierry Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*. Washington D.C., USA, Mai 2003.

[11] Abdoulaye Gamatié, Thierry Gautier, and Loic Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language

4. `http://www.irisa.fr/espresso/Polychrony`.

signal. *proceedings of Synchronous Languages, Applications, and Programming (SLAP'03). Portugal*, 2003.

[12] Abdoulaye Gamatié, Thierry Gautier, and Paul Le Guernic. An example of synchronous design of embedded real-time systems based on IMA. In *Proceedings of the 10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA'04)*. Gothenburg - Sweden, Août 2004.

[13] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symposium (SSS'99), Springer*. Huntingdon, UK, Février 1999.

[14] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *Formal Methods and Models for Codesign Conference, Mont Saint-Michel, France*, Juin 2003.

[15] T.A. Henzinger, B. Horowitz, and Ch.M. Kirsch. Embedded control systems development with Giotto. In *Proceedings of LCTES. ACM SIGPLAN Notices*, 2001.

[16] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus - a tool for distributed systems specification. In *proceedings of FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, P. 467-470*. Springer Verlag, LNCS 1135, 1996.

[17] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE*, pages 6/1–6/9. HP Labs, Bristol, UK, Février 1996.

[18] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. In *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. (c) World Scientific*, Avril 2002.

[19] B. Lewis, S. Vestal, and D. McConnell. Modern avionics requirements for the disbuted system annex. In *Ada-Europe'98, L. Asplund, Ed, LNCS 1411, Springer Verlag, p. 201-212*, 1998.

[20] A. Pnueli. Embedded Systems : Challenges in Specification and Verification. In *2002 Conference on Embedded Software, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Spr. Verlag., p. 252-265*, 2002.

[21] J. Romberg. Model-based deployment with autofocus : a first cut. In *14th Euromicro Conference on Real Time Systems (ECRTS'02), Work In Progress session*, pages 41–44. Vienna, Austria, Juin 2002.

[22] J. Rushby. Partitioning in avionics architectures : Requirements, mechanisms, and assurance. Technical report, Computer Science Laboratory SRI International, Menlo Park CA 94025 USA. Mars 1999.