Projet ROSCOV
(Robuste Ordonnancement de Systèmes de COntrôle de Vol)
Méthodes formelles temporisées appliquées à l'ordonnancement

L. Fribourg (LSV), D. Lesens (ASTRIUM)

Abstract

Le but du projet est de traiter un problème d'*ordonnancement* des tâches d'un logiciel de contrôle de vol, conçu par ASTRIUM, fonctionnant sur une architecture multi-processeurs.

Un enjeu majeur consiste à trouver un ordonnancement qui réduise significativement le *coût* de mise en oeuvre et de fonctionnement du système. Un autre défi consiste à trouver un ordonnancement qui soit à la fois *robuste* et *extensible* : on souhaite en effet trouver une solution qui puisse être réutilisée malgré certaines variations possibles de paramètres temporels et de configuration de l'architecture support.

Pour aborder ce problème, on adoptera des techniques de modélisation par *automates temporisés*. Les méthodes d'analyse à base d'automates temporisés ont en effet déjà montré leur intérêt dans le cadre classique d'ordonnancement par *jobshop* (ou "atelier à cheminements multiples"), dans lequel on cherche à trouver une suite appropriée d'affectations de tâches à des machines de façon à minimiser le temps total de production. En outre, des extensions des méthodes d'analyse pour automates temporisés avec paramètres ont permis récemment de fournir des mesures de robustesse pour les solutions générées.

1 Modélisation du problème d'ordonnancement

1.1 Le problème de base : Job-Shop

Au niveau le plus abstrait, le problème d'ordonnancement peut se définir comme suit : un ensemble de tâches à remplir en utilisant un ensemble borné de ressources disponibles et réutilisables. Chaque tâche est caractérisée par sa durée, par les ressources nécessaires à son accomplissement et par les relations de préséance de la tâche avec les autres tâches. Un conflit intervient entre deux tâches quand leur demande simultanée d'un type de ressource dépasse la disponibilité de cette ressource. Le rôle d'un ordonnanceur est de résoudre ces conflits en décidant à laquelle des tâches en compétition doit être donnée la ressource en premier. Differents ordonnancements conduisent naturellement à différents ordre d'exécution des tâches et le but d'un ordonnancement optimal est de trouver un ordonnanceur qui se comporte de façon optimale par rapport à un certain critère d'évaluation.

L'application classique du problème d'ordonnancement vient de l'ingénierie industrielle : comment utiliser un nombre fini de machines d'une usine de façon à fabriquer différents produits de façon efficace (jobshop). En informatique, ce type de problèmes apparait par exemple, quand on veut allouer du temps CPU et des périphériques dans un système d'exploitation multi-tâche, ou quand on veut allouer des registres dans un CPU, ou allouer des canaux de communication dans un réseau.

La forme la plus simple de modélisation du problème d'ordonnancement s'appelle le problème du *jobshop*. Il s'agit d'un ensemble fini $J = \{J^1, \ldots, J^m\}$

de "jobs" à exécuter sur un ensemble fini de machines $M = \{m_1, \ldots, m_k\}$. Chaque job J^i est une suite finie de tâches à exécuter l'une après l'autre, chaque tâche étant caractérisée par un couple de la forme (m, d) avec $m \in M$ et $d \in \mathcal{N}$, où m désigne le nom de la machine et d la durée de son utilisation requise par la tâche. Chaque machine ne peut exécuter qu'une tâche en même temps, et on ne peut exécuter au plus qu'une seule tâche d'un job à la fois. Dans la version simple du problème de job-shop, les tâches ne peuvent être préemptées (interrompues par une autre tâche), une fois qu'elles ont démarré.

La succession des couples (m, d) à l'intérieur d'un job J induit une relation de **préséance** entre tâches : l'exécution d'une tâche (m, d) doit être terminée avant le début d'exécution de la tâche suivante dans la suite.

Par ailleurs, la durée d d'une tâche peut être vue comme une **WCET** (Worst Case Execution Time) de la tâche.

On peut en outre affecter des **deadlines** au temps d'exécution d'un job, c.-à-d., une date maximale pour la fin d'éxécution de la dernière tâche du job.

Le rôle de l'ordonnanceur est de déterminer les *temps de démarrage* de chaque tâche.

Un objectif classique à atteindre peut être de minimiser le temps d'exécution globale de tous les jobs, c.-à-d. le temps de terminaision de la dernière tâche du dernier job en cours (*end-to-end duration* ou *makespan*).

Considérons l'ensemble $M = \{m_1, m_2, m_3\}$ et les deux jobs

$$J^1 = (m_3, 2), (m_2, 2), (m_1, 4) \text{ et } J^2 = (m_2, 3), (m_3, 1)$$

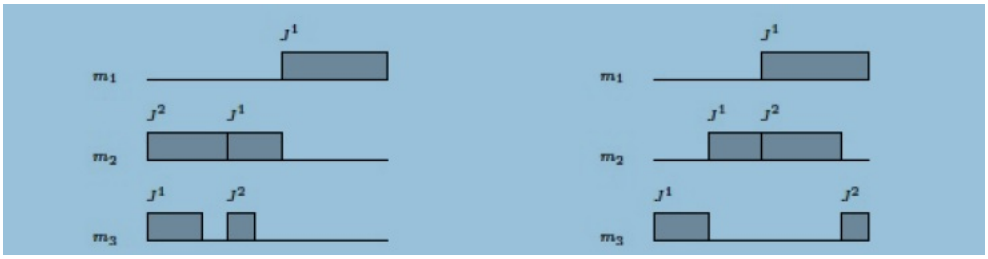

Figure 1: Deux ordonnancements pour un problème job-shop à 2 jobs et 3 machines

Deux ordonnancements S_1 et S_2 sont représentés sur la figure 1. La longueur de S_2 est 8, et correspond à l'ordonnancement optimal.

Il y a de très nombreuses variantes à ce cadre de base : on peut notamment autoriser certaines tâches à en interrompre d'autres moins prioritaires (*préemption*); on peut également rajouter des contraintes sur l'ordonnanceur, par exemple en imposant que certaines tâches s'exécutent avant une certaine date limite (*local deadline*). Ceci nous amème à un type plus général de modélisation du problème d'allocation et d'ordonnancement des tâches.

NB: Noter que dans le cas simple sans préemption, la date de fin *end* d'une tâche se déduit de la date de début *start* par ajout de la durée associée d de la tâche (WCET).

2 Traitement par automates temporisés

Les *Automates Temporisés* (TAs) sont des automates finis enrichis avec des variables réelles ("horloges") dont la valeur croît uniformément dans chaque état. Les horloges peuvent être remises à zéro lors des transitions d'un état à un autre, et on utilise des tests sur leurs valeurs pour autoriser ces transitions (gardes).

Comme il est montré dans [1], les TAs permettent de traiter naturellement des problèmes simples d'ordonnancement comme ceux du job-shop : on représente chaque job par un automate temporisé et on utilise des variables supplémentaires discrètes pour indiquer qu'une machine est libre ou en cours d'utilisation. En calculant l'ensemble des états accessibles par des méthodes symboliques, on obtient le graphe complet d'accessibilité, chaque chemin représentant un ordonnancements possible. Un exemple d'un tel graphe est présenté en figure 2.

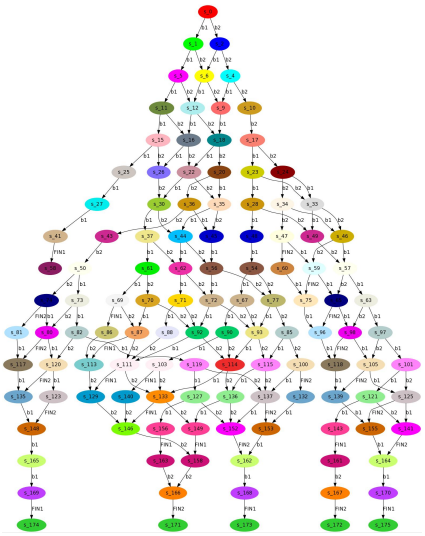

Figure 2: Exemple de traitement d'un problème de jobshop par des automates temporisés avec 2 jobs passant chacun par 4 machines

A partir du graphe d'accessibilité, il est facile d'extraire le chemin le moins "coûteux", c'est-à-dire celui qui optimise le critère considéré. On se ramène ainsi à traiter un problème du *plus court chemin* dans le contexte des TAs.

Dans de nombreux cas, et en particulier dès que le nombre de machines ou de jobs augmente, une synthèse naïve de ce graphe d'accessibilité entraîne des problèmes (temps de calcul très long, mémoire nécessaire trop importante). De nombreuses techniques ont été développées afin de réduire la taille de ce graphe, en regroupant certains étatss, en ne considérant pas les chemins qui violent certaines conditions (par exemple, la longueur du chemin courant dépasse une borne donnée). On peut également envisager des solutions de parallélisation (voir Section 5.3).

Les automates utilisés peuvent également être paramétrisés, afin d'ajuster la valeur de certaines bornes temporelles (telles que le WCET). Cette paramétri-

sation permet également de quantifier la robustesse (ou flexibilité) des ordonnancements trouvés. Les graphes d'accessibilité obtenus sont en général plus large que dans le cas non paramétrisé et par conséquent l'utilisation de techniques visant à limiter l'exploration est essentielle.

3 Cadre plus général d'ordonnancement

3.1 Extension avec périodicité et préemption des tâches

Dans un type de représentation plus sophistiquée [7], on tient compte du caractère *périodique* des tâches ainsi que de leur caractère *préemptible* (c.-à-d., interruptibles par une tâche plus prioritaire).

La fonctionnalité d'un système est alors représentée sous forme de graphe orienté acyclique (DAG). Chaque noeud du DAG représente une tâche (par exemple, une suite d'instructions à exécuter sur un CPU); chaque arc représente une préséance entre deux tâches (la tâche-source doit être achevée avant le démarrage de la tâche-destination). Les tâches peuvent donc être préemptées.

Le graphe des tâches peut être constitué de plusieurs *partitions* déconnectées. Chaque partition déconnectée représente une "machine virtuelle" indépendante.

Chaque tâche a une *période* entière qui détermine le nombre de fois où la tâche doit être exécutée. La *super-période* est le ppcm de toutes les périodes des tâches. Toutes les tâches du graphe sont ordonnancées à l'intérieur d'une super-période. L'origine du temps correspond au démarrage de la super-période. Une tâche peut avoir à être répétée en ajustement avec la super-période (ceci correspond à une "expansion du graphe des tâches" (TGE)). Voir figure 3.

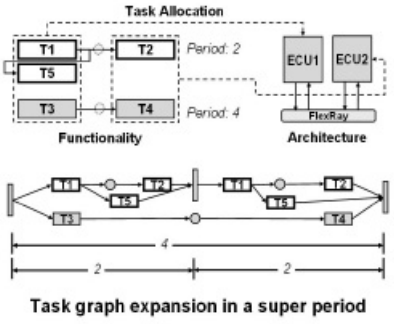

Figure 2. Mapping and task graph expansion

Figure 3: Graphe d'allocation et priorité des tâches et son expansion

Dans la modélisation de [7], il y a 7 paramètres temporels qui concernent chaque tâche: la *période* , la *deadline* (date limite à laquelle la tâche doit avoir été exécutée), la *release* (temps de relâche), le *WCET* (durée maximale d'exécution) le *start* (temps de début d'exécution), le *finish* (temps de fin d'exécution), le *idle* (variable intermédiaire donnant la durée d'inactivité de la tâche).

Seul le temps *start* est une variable déterminée par l'ordonnanceur. Toutes les autres variables font partie de la spécification du problème (ou peuvent être immédiatement déterminées en fonction de *start* à partir de la spécification). En d'autres termes, étant donné une spécification, un *ordonnancement* est une fonction qui associe à chaque tâche sa date de début *start*.

Rappelons qu'un ordonnancement (admissible) satisfait implicitement la contrainte suivante : chaque PU (*Processing Unit*) exécute, au maximum, une seule tâche à la fois. Rappelons également que, dans le cadre de [7], les tâches peuvent être préemptées, c.-à-d., interrompues par une tâche plus prioritaire.

Un exemple d'ordonnancement est présenté en Figure 4 avec, en indication, les paramètres temporels relatifs à la tâche 3.

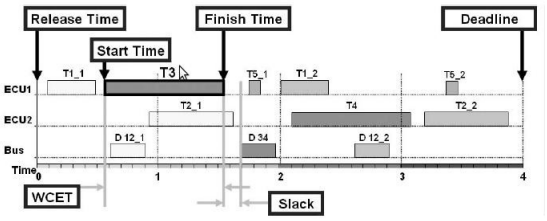

Figure 4: Illustration des différents paramètres temporels sur la tâche T3

3.2 Autres critères et objectifs d'ordonnancement

Dans ce nouveau cadre comme pour le job-shop classique, on peut continuer à s'intéresser à trouver l'ordonnancement admissible qui minimise la date d'exécution de la dernière tâche à l'intérieur d'une super-période.

Cependant, il existe d'autres critères de qualité (et donc, d'optimisation) pour les ordonnancements. Des critères comme la *réutilisabilité* ou la *robustesse* sont présentés dans [7]. On peut s'intéresser ainsi à des ordonnancements qui continuent à respecter les contraintes d'admissiblité même quand certaines variations du système sont effectuées.

Différents ordonnancements sont présentés en Figure 5. Chacun optimise un critère de qualité différent.

4 Etude de cas ASTRIUM

4.1 Formalisation schématisée du futur FCS (Flight Control System)

L'étude de cas qui nous intéresse concerne le système de contrôle de vol qui sera embarqué sur les futurs véhicules spatiaux (ATV "évolution", Ariane "6", etc. voir figure 6).

Le rôle du système est de contrôler l'attitude, la trajectoire et la vitesse du véhicule. Dans ce but, le système calcule périodiquement les ordres qui doivent être appliqués aux actionneurs de vol en réponse aux mouvements

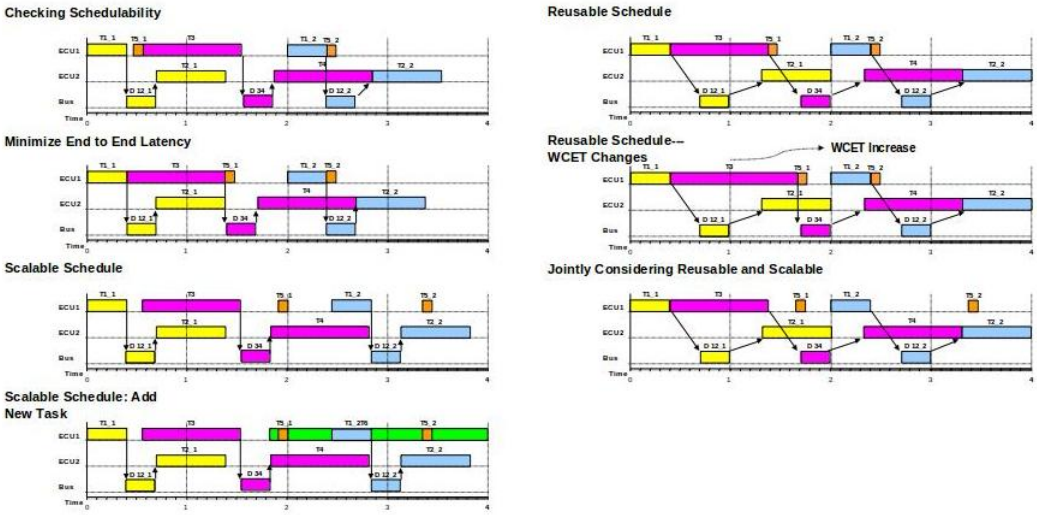

Figure 5: Différents ordonnancements optimisant des critères différents

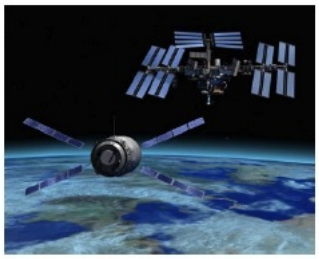

Figure 6: Véhicule ATV approchant la station orbitale ISS

du véhicule. Jusqu'à présent, l'architecture du système effectivement implémenté ne contient qu'un processeur (PU). Il contient en outre des organes de pilotage, des senseurs acquérant l'état du véhicule (GPS pour la position, accéléromètre pour la vitesse), et des actionneurs (moteurs). Le problème général de l'ordonnancement est de trouver les dates de début d'activation des sous-programmes du logiciel de façon à respecter toutes les contraintes de délais et de préséance entre tâches spécifiées.

Dans ce cadre restreint d'architecture centralisée avec un seul PU, le problème d'ordonnancement a été traité dans [5, 6]. Dans la suite, nous présentons le problème dans une version prospective plus générale : décentralisée sur plusieurs processeurs. ASTRIUM a en effet le projet de construire un nouveau logiciel de contrôle de vol qui, pour des raisons d'efficacité, pourra s'exécuter sur plusieurs processeurs, et non plus un seul. Le problème d'ordonnancement en environnement décentralisé est naturellement plus complexe car il faut prévoir, non seulement l'ordonnancement des tâches au sein d'une même PU, mais également gérer les échanges entre PUs par le biais d'un bus de communication.

Dans la formulation ASTRIUM, les tâches logicielles sont organisées en *partitions*. Informellement, une partition est une machine virtuelle permettant de raisonner sur le logiciel d'application indépendamment des PUs. Une partition exécute une suite d'unités logicielles, appelées *threads*[1] sur un même PU. Les partitions ainsi que les threads sont effectués de façon périodique. On a une contrainte d'exclusion qui dit que deux partitions qui s'exécutent sur le même PU ne peuvent pas être actives simultanément. En outre, une partition ne peut pas écrire dans l'espace-mémoire d'une autre partition. Plus formellement, le système est modélisé de la façon suivante :

- L'ensemble $\mathcal{C} = \{\mathcal{C}_i\}$ des process units (PUs)
- La communication entre PUs se fait grâce à un bus, appelé *Avionics Communication Mean* et noté α. Ce bus n'autorise le passage que d'une seule donnée (frame) à la fois à des instants **périodiques**.
- Une fonction de partition assignant un ensemble de partitions $\{\mathcal{P}_i\}$ à chaque PU
- Une fonction de thread assignant un ensemble de threads $\{\beta_i\}$ à chaque partition
- Chaque thread (et partition) s'exécute de façon **périodique** ; en outre, le temps maximal d'exécution d'une thread est connue (**WCET**).
- Les données sont échangées entre threads de façon *synchrone* : un thread lit ses inputs au démarrage de son exécution (**start**) et transmet ses outputs à la fin (**end**). Ceci permet de faire abstraction des données, et de ne considérer que les relations de préséance entre threads (voir ci-dessous, la notion de flows).
- un ensemble $\{\mathcal{F}_i\}$ de *end-to-end flows* : il s'agit d'un flux d'information circulant à travers des threads, représenté sous la forme d'un chemin orienté.[2] Chaque flow induit une relation de **préséance** entre les threads qu'il traverse. En outre, la durée maximale d'écoulement de bout en bout d'un flow est une contrainte donnée par la spécification (**deadline**).

En récapitulation, la spécification donne les informations suivantes :
- pour chaque thread : la période, le WCET, le deadline, la relation de préséance avec les autres threads (induite par les graphes de flow) ;
- pour chaque partition : la période ;
- pour chaque PU : les partitions qu'il exécute ;
- pour le bus de communication α : la période d'acheminement des données ;
- pour chaque flow: le deadline.

Le but du problème d'ordonnancement est de trouver, pour chaque thread de chaque partition (et pour chaque partition de chaque PU), la date de démarrage *start* (et de fin *end*) de son activation, en respectant toutes les contraintes

[1] correspondant grosso modo à la notion de tâche dans le modèle job-shop

[2] Par exemple, un senseur transmet une donnée d'accélération à des threads de navigation qui (calculent et) transmettent une position au système de contrôle qui (calcule et) transmet alors une consigne à un actionneur.

liées aux différentes valeurs de WCET, deadline, préséance, et période de la spécification.

Il existe en général plusieurs ordonnancements admissibles. Comme on l'a vu, on peut s'intéresser à différents critères d'optimisation : minimisation du temps de la dernière tâche accompli, minimisation du nombre de threads, du nombre de PUs, optimisation de la robustesse, de l'extensibilité, ...

4.2 Données de l'étude de cas schématisé

Dans l'étude de cas considérée, le système comprend trois PUs (voir figure 7): $\mathcal{C}_{Nav}$, $\mathcal{C}_{Seq}$ et $\mathcal{C}_{Ctrl}$ (en plus de $\mathcal{C}_{TM}$ pour la communication).

Le PU $\mathcal{C}_{Nav}$ (Navigation Processing Unit) est composé de deux partitions $\mathcal{P}_{Gyro}$ et $\mathcal{P}_{Nav}$. La partition $\mathcal{P}_{Gyro}$ est faite d'un seul thread $\mathcal{T}_{IntLoop}$. La partition $\mathcal{P}_{Nav}$ est faite de 3 threads $\mathcal{T}_{GyroMgt}$ (Gyroscope Management), $\mathcal{T}_{Nav}$ (Navigation Algorithm), $\mathcal{T}_{NTM}$ (Navigation Telemetry).

Le PU $\mathcal{C}_{Seq}$ (Sequential Processing Unit) est composé de deux partitions $\mathcal{P}_{Seq}$ et $\mathcal{P}_{GuiNav}$. La partition $\mathcal{P}_{Seq}$ (Sequential Partition) est faite de 2 threads $\mathcal{T}_{MVMSlow}$ (Slow Mission and Vehicle Management), $\mathcal{T}_{MVMFast}$ (Fast Mission and Vehicle Management). La partition $\mathcal{P}_{Gui}$ (Guidance Partition) est faite de 2 threads $\mathcal{T}_{Guidance}$ (Guidance Algorithm), $\mathcal{T}_{CtrLoop}$ (ControlLoop).

Le PU $\mathcal{C}_{Ctrl}$ (Control Processing Unit) est composé de deux partitions $\mathcal{P}_{Eng}$ et $\mathcal{P}_{Ctrl}$. La partition $\mathcal{P}_{Eng}$ (Engine Partition) est faite d'un seul thread $\mathcal{T}_{CtrlLoop}$ (Control Loop). La partition $\mathcal{P}_{Ctrl}$ (Control Partition) est faite de 3 threads $\mathcal{T}_{EngMgt}$ (Engine Management), $\mathcal{T}_{Control}$ (Control Algorithm), $\mathcal{T}_{CTM}$ (Control Telemetry).

Il y a 5 end-to-end flows de définis : le *flight control flow* (voir figure 8) ; le *management flow* (voir figure 9) ; le *telemetry flow* (voir figure 10).

Par ailleurs, toutes les périodes des 12 threads sont définies. [3]

Enfin les deadlines de chaque end-to-end flows sont eux aussi définis, ainsi que les préséances mutuelles des threads.

Rappel : le problème d'ordonnancement consiste à trouver les temps de démarrage de chaque thread pour chaque partition de chaque PU.

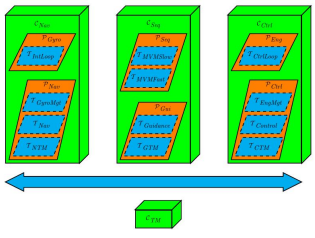

Figure 7: Architecture décentralisée (3 PUs) du système de contrôle de vol FCS (en vert : les PUs ; en orange : les partitions ; en bleu : les threads)

[3] 1000 Hz pour $\mathcal{T}_{IntLoop}$ et $\mathcal{T}_{CtrlLoop}$; 500 Hz pour $\mathcal{T}_{EngMgt}$; 100 Hz pour $\mathcal{T}_{Nav}$, $\mathcal{T}_{NTM}$ et $\mathcal{T}_{MVMFast}$; 50 Hz pour $\mathcal{T}_{Control}$; 10 Hz pour $\mathcal{T}_{GyroMgt}$, $\mathcal{T}_{MVMSlow}$, $\mathcal{T}_{GTM}$ et $\mathcal{T}_{CTM}$; 0.1 Hz pour $\mathcal{T}_{Guidance}$.

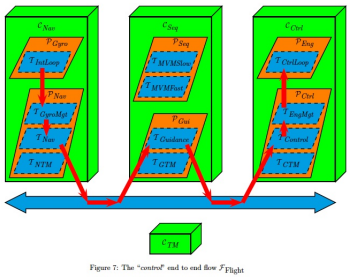

Figure 8: The 'control' end-to-end flow $\mathcal{F}_{Flight}$

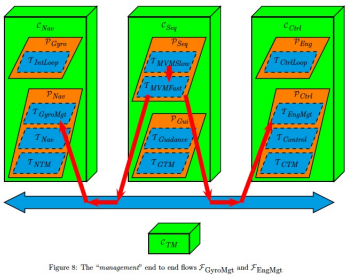

Figure 9: The 'management' end-to-end flow $\mathcal{F}_{GyroMgt}$ et $\mathcal{F}_{EngMgt}$

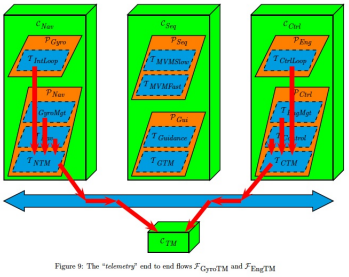

Figure 10: The 'telemetry' end-to-end flow $\mathcal{F}_{GyroTM}$ et $\mathcal{F}_{EngTM}$

5 Roadmap

5.1 Approche envisagée

Le but du projet est de résoudre un problème d'*ordonnancement* associé à l'étude d'un système de contrôle de vol (FCS) conçu par ASTRIUM. Au début du projet, nous traiterons une version simplifiée du FCS (dans la forme présentée Section 4.2), puis nous étudierons, dans un second temps, une version réaliste plus complexe.

A priori, la difficulté principale du problème d'ordonnancement dans cette étude de cas ne réside pas dans le nombre de tâches dont le logiciel est constitué : un ordonnancement admissible devrait être a priori relativement facile à trouver. En revanche, le problème critique pressenti consiste à trouver un ordonnancement (sub)optimal qui abaisse significativement le *coût* de mise en oeuvre et fonctionnement d'une solution naïve. Un autre défi de cette étude de cas consiste à trouver un ordonnancement qui soit à la fois *robuste* et *extensible* : on souhaite en effet trouver une solution qui puisse être réutilisée malgré certaines variations de paramètres temporels (WCET, deadline, période, ...) et de configuration d'architecture support.

Pour traiter ce problème, on adoptera des techniques à base de modélisation par *automates temporisés*. Ce formalisme et les méthodes associées ont déjà fait leurs preuves dans le cadre classique du problème du jobshop (voir Section 2). En outre, des extensions des méthodes pour les automates temporisés avec paramètres ont récemment permis de quantifier la robustesse des solutions trouvées [2].

5.2 Compétences et participation

David Lesens est ingénieur à ASTRIUM. Il est expert logiciel embarqué et est notamment responsable de la coordination des recherches dans le domaine des logiciels bords. Laurent Fribourg est directeur de recherche CNRS. Il travaille depuis une dizaine d'années au LSV sur le développement et l'application de méthodes formelles à base d'automates temporisés paramétrés. Durant les 2 ans de ce projet prévu (2012-2013), nous participerons de la façon suivante :

- du côté LSV, Laurent Fribourg, DR CNRS (10%) et Romain Soulat, doctorant (50%)

- du côté ASTRIUM, David Lesens, ingénieur (10%) et Pierre Moro, ingénieur (10%)

En outre, nous souhaitons recruter de mars à septembre 2012 un stagiaire de M2 qui travaillerait, sous la co-direction du LSV et ASTRIUM, à plein temps sur le sujet.

5.3 Implication potentielle d'un autre partenaire Farman

Contrairement à la forme des projets Farman classiques des années précédentes, ce projet implique non pas deux, mais un seul laboratoire Farman (le LSV), cette fois-ci en partenariat avec un industriel (ASTRIUM).

Nous souhaiterions par la suite faire participer un autre laboratoire Farman à ce projet. Il serait notamment très utile de *paralléliser* le code nécessaire à l'exploration de l'espace des états lors de nos traitements par automates temprisés. Ce domaine de parallélisation de code est un sujet d'expertise de laboratoires comme le CMLA (avec, par exemple, Florian De Vuyst) ou le LMT (avec, par exemple, Christian Rey), qui ont à traiter couramment des applications informatiques de grande dimension. Nous prévoyons, au fur et à mesure de l'avancement du projet, de tenir informés de nos résultats ces interlocuteurs naturels afin de soumettre à discussion nos travaux et préparer concrètement une coopération dans le domaine de la parallélisation.

5.4 Feuille de route des travaux

Commençons par souligner que le présent document est issu de la visite de D. Lesens (ASTRIUM) à Cachan en juin où il a exposé dans le cadre de l'Institut Farman les sujets potentiels sur lesquels il souhaiterait coopérer. Cette visite a été suivie de plusieurs heures de discussions téléphoniques avec L. Fribourg qui ont permis de cerner l'étude de cas, et de commencer la formalisation décrite ci-dessus. La suite prévisionnelle de la coopération est la suivante :

- 1er semestre 2012 : programmation de l'étude de cas dans le formalisme des automates temporisés et premières expérimentations à l'aide des outils de vérification disponibles au LSV ; discussions et ajustements de la modélisation en interaction avec ASTRIUM.

- 2ème semestre 2012 : étude de robustesse et d'extensabilité en faisant varier notamment les valeurs nominales temporelles (WCET, période, deadline) de la spécification.

- 1er semestre 2013 : Elaboration d'une seconde modélisation d'architecture plus réaliste, raffinant le niveau élémentaire de tâche, et complexifiant leurs relations de communication.

- 2ème semestre 2013 : Expérimentations et ajustements sur la seconde modélisation.

5.5 Financement demandé

Conformément à l'appel à projet Farman, le financement dans ce cadre de coopération avec un industriel, ne concerne que le partenaire académique, en l'occurrence le LSV.

Le montant du financement demandé est de 5Keuros en fonctionnement sur la durée du projet (soit 2 ans). Ce soutien servirait essentiellement à financer des missions pour des colloques de Recherche Opérationnelle, à la fois pour acquérir de nouvelles connaissances en ce domaine, et pour diffuser les prochains résultats que nous espérons obtenir dans le cadre du projet.

References

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with timed automata. *Theor. Comput. Sci.*, 354:272–300, March 2006.

[2] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, October 2009.

[3] Peter Brucker. The job-shop problem: Old and new challenges.

[4] A.S. Eshlaghy and S.A. Sheibatolhamdy. Scheduling in flexible job-shop manufacturing system by improved tabu searcg. *African Journal of Business Managment*, 5:4863–4872, 2011.

[5] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *SAC*, pages 527–534, 2010.

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

[7] Wei Zheng, Jike Chong, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 132–141, Washington, DC, USA, 2005. IEEE Computer Society.