IDM 2006

Ingénierie dirigée par les modèles appliquée à la conception d'un contrôleur de robot de service

Dave Thomas* — Claude Baron* — Bertrand Tondu*

* *Laboratoire d'Étude des Systèmes Informatiques et Automatiques*
INSA-DGEI de Toulouse
135 avenue de Rangueil, F-31077 Toulouse cedex 4
{dave.thomas, claude.baron, bertrand.tondu}@insa-toulouse.fr

RÉSUMÉ. *Les approches basées modèles marquent une réelle démarche d'évolution dans la conception de systèmes au sens large. Les différentes étapes du développement d'un système, jusque là essentiellement supportées par des documents textuels (informels), sont de plus en plus formalisées afin d'éviter les ambiguïtés, garantir leur complétude et leur cohérence et d'assurer au plus tôt l'adéquation de la solution décrite et de l'expression du problème.*

Dans le cadre de travaux sur une robotique de service anthropomorphe, nous définissons et expérimentons une approche de modélisation évolutive pour le développement du logiciel de commande temps-réel d'un prototype de bras à sept axes actionnés par muscles artificiels. A partir du modèle d'analyse orienté métier plusieurs étapes sont définies afin de guider la conception vers un modèle d'implémentation en pratiquant par transformations successives. Les modèles servent alors de référence à la fois pour le logiciel final ainsi que pour les analyses et les vérifications.

ABSTRACT. *Model-based engineering is going to modify the way you design systems. It encourages replacing textual documents with formal representations in order to enhance verification, coherency and requirements fulfilment.*

The purpose of this paper is to make an overview of how we considered the model-based contribution in support to control system engineering, based on the design of a seven-degrees-of-freedom robot-Arm driven by pneumatic artificial muscles for humanoid robots.

MOTS-CLÉS : *MDA, UML, AADL, temps-réel, robotique.*

KEYWORDS: *MDA, UML, AADL, real-time, robotics.*

2ème soumission IDM 2006 – 1er juin 2006

1. Introduction

1.1. *Contexte et besoins*

Par opposition à la robotique industrielle, dont les applications sont essentiellement manufacturières, la robotique de service vise à sortir le robot du cadre contraignant de l'atelier ou de la chaîne de montage. La fédération internationale de robotique (IFR) a suggéré en 1997 la définition suivante : "A service robot is a robot that operates partially or fully autonomously to perform services useful to the well-being ... of humans and equipment. They're mobile or manipulative or a combination of both" (cité dans Service Robot [Schraft, 2000], page 2). La robotique de service couvre donc un très large champ d'application, du BTP à l'agriculture, en passant par la surveillance, le marketing, les loisirs, la médecine et l'assistance aux personnes. La sécurité active et la fiabilité sont, par conséquent, des impératifs fondamentaux de la mise en œuvre des robots de service.

L'équipe de robotique du LESIA développe depuis plusieurs années une plate-forme expérimentale définie autour d'un bras-robot anthropomorphe à 7 degrés de liberté actionnés par muscles artificiels de McKibben ([Tondu, 2005]). La mise en place d'une telle plateforme nécessite les compétences de plusieurs domaines d'ingénierie tels que la robotique, la mécanique, l'automatique ou l'informatique.

Cette plateforme inclut un système informatique complexe. Ce dernier est constitué d'un logiciel temps-réel, appelé contrôleur de robot, et de son support d'exécution reposant sur un exécutif temps-réel, dans notre cas VxWorks. Le contrôleur de robot héberge les lois de commande et les algorithmes de dynamique et de cinématique élaborés par les roboticiens et les automaticiens. L'approche traditionnelle de développement, organisée autour d'un codage à la main et d'une implémentation mono-tâche, a très vite montré ses limites vis-à-vis de la complexité de la plateforme à 7 axes et nous avons mis en évidence la nécessité d'une démarche guidant le concepteur. Sans cette méthodologie appropriée, il devient très difficile de réaliser le logiciel qui doit être à la fois flexible pour s'adapter aux différents prototypes et sûr de fonctionnement, afin de satisfaire les contraintes de sécurité.

1.2. *Évolution des réflexions*

Nos travaux se sont tout d'abord orientés vers l'emploi des méthodes SADT et SART ([Carroll, 1999]) pour la conception fonctionnelle et temps-réel de nos contrôleurs de robot. Les résultats ont montré des avantages notables dans la maîtrise du logiciel. Pourtant, pour les roboticiens, un certain nombre de schémas utilisés dans ces méthodes restaient difficilement projetables sur la plateforme réelle et un manque de lisibilité subsistait. L'approche objet présentait l'avantage de réconcilier le domaine du logiciel avec le monde réel en basant les démarches sur la représentation des éléments concrets (les objets). Aussi, une modélisation à l'aide du langage UML du contrôleur a été réalisée, permettant d'améliorer la compréhension du logiciel ([Guiochet, 1999]). Ensuite, le modèle a été remanié dans le but d'accroître la flexibilité et la possibilité de réutilisation de certaines parties

(composants) du logiciel ([Thomas, 2004]). Une architecture de référence a alors été proposée afin d'abstraire les parties spécifiques de bas-niveau pour concentrer les efforts sur la partie fonctionnelle.

1.3. *Les approches MDA et MDE*

Séparer explicitement solution fonctionnelle (logique) et solution technologique (physique) est nécessaire pour faciliter le déploiement d'un logiciel sur différentes plateformes d'exécution. L'approche MDA (Model Driven Architecture [OMG, 2003]) propose à ce sujet de s'appuyer sur les modèles pour supporter les deux types de représentations. Des liens peuvent ensuite être tissés entre les modèles pour permettre de les assembler de manière automatisée et produire le logiciel final.

L'ingénierie dirigée par les modèles (MDE, Model Driven Engineering) est la généralisation de l'approche MDA [Favre et al., 2006]. Les avantages annoncés de l'IDM sont nombreux : indépendance vis à vis des évolutions technologiques, meilleure maîtrise de la complexité, meilleure réutilisation etc. Le nombre de vues augmentant, les modèles utilisés et leur sémantique associée sont de mieux en mieux définis et, petit à petit, les modèles occupent ainsi une position centrale dans le processus de développement, celle de prototype virtuel du système à réaliser. En effet, l'évolution des ateliers logiciels (tels que I-Logix™ Rapsody, Telelogic™ Tau, Artisan Software™ Studio) et des langages autorise aujourd'hui la construction de modèles exécutables (cf. Executable Systems Design with UML2.0 [Niemann, 2004]). L'analyse et la simulation sont alors permises et facilitent la correction au plus tôt d'erreurs détectées jusqu'à maintenant seulement pendant les phases de tests, après codage. L'utilisation d'une représentation simulable (le modèle) peut ainsi réduire significativement les temps et les coûts des itérations de maintenance corrective (débogage). Enfin, l'approche généralise également la transformation de modèles. En effet, que ce soit pour appliquer un patron de conception, pour séparer les préoccupations à travers différentes vues ou pour réaliser des fusions, les transformations seront utilisées pour modifier, créer ou même supprimer certains éléments du modèle. De même, la génération de code ou de documents constitue un autre type de transformation (modèle vers texte) qui oriente les modèles vers une utilisation plus productive. Plus généralement, la transformation de modèles servira de moyen pour automatiser certaines tâches ou, du moins, les accélérer.

Cet article présente une mise en œuvre de ces concepts à travers la conception du contrôleur de robot à 7 axes présenté paragraphe 1.1.

La première partie, section 2, est consacrée à la démarche de modélisation qui, faisant référence au MDA, mène à construire tout d'abord une architecture fonctionnelle indépendante de la plateforme puis l'architecture dynamique et physique sur laquelle elle repose.

La deuxième partie, section 3, s'intéresse alors à la vérification et à la validation du logiciel à partir de ces modèles.

Enfin, la troisième partie a pour but de résumer la démarche suivie et de faire la synthèse sur les concepts employés et les leçons apprises.

2. Modélisation du contrôleur de robot

La modélisation du contrôleur de robot consiste à définir l'architecture du logiciel de contrôle du robot ainsi que les interfaces nécessaires en vue de sa projection sur son support d'exécution. Cette section présente tout d'abord la phase d'analyse des besoins depuis laquelle nous sommes partis pour construire les modèles. Les éléments de l'architecture fonctionnelle en sont déduits et dérivés progressivement. Enfin, la prise en compte des contraintes d'implémentation est abordée après avoir modélisé le comportement à l'exécution en décrivant l'architecture dynamique

2.1. *Analyse des besoins*

Dans le but de supporter la phase de capture des besoins, nous avons fait le choix d'une approche aujourd'hui devenue classique, basée sur le langage UML. Les cas d'utilisation décrivent les besoins selon le point de vue de l'utilisateur et les illustrent de manière graphique à l'aide de diagrammes. Cet apport visuel favorise les discussions avec le client, permet de situer facilement le périmètre du système ainsi que ses interfaces avec le monde extérieur. Ensuite, la description des scénarios est réalisée à l'aide de diagrammes de séquences à travers lesquels sont analysés les interactions de haut niveau et les objets primaires du système.

A ce premier niveau d'analyse, une architecture statique d'objets se dessine. La spécification peut être complétée d'une description comportementale sous forme de machines à états associées aux objets. On obtient alors un **modèle d'analyse** (cf. Figure 1.a) composé des objets représentatifs du système et de leurs associations.

2.2. *Modèle indépendant de la plateforme*

2.2.1 Architecture statique

Après avoir traité les fonctionnalités du système, le concepteur s'efforce de faire évoluer l'architecture afin de répondre à des besoins moins fonctionnels et tendre vers un modèle structurel d'implémentation en décomposant les objets. Par exemple, la plateforme robotique est avant tout un environnement d'expérimentation qui doit faciliter l'introduction de nouveaux algorithmes et de nouveaux prototypes de bras ou, plus généralement, de systèmes articulés. La flexibilité et la réutilisabilité sont des propriétés recherchées dans le but de mettre en œuvre un contrôleur générique. Notre démarche consiste à illustrer le travail du concepteur à la fois pour capturer son savoir-faire et pour identifier ses besoins. Par exemple, pour chaque lien qui existait entre deux objets une interface a été créée pour réduire le couplage (cf Figure 1.b), ce qui pourrait facilement être automatisé au niveau outil à l'aide d'une transformation de modèles.

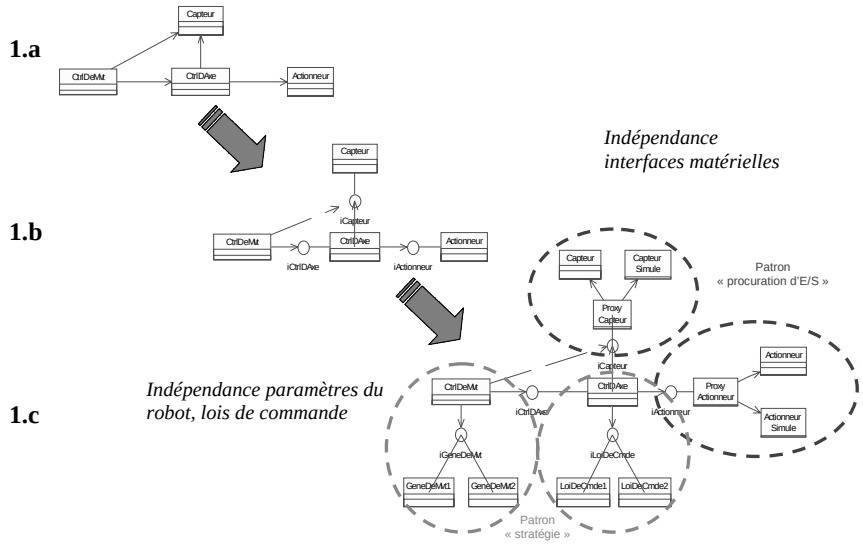

Figure 1. *Construction de l'architecture fonctionnelle par transformation*

2.2.1.1 *Patrons de conception*

Ensuite, Figure 1.c, nous nous sommes inspirés des Patrons de Conception (design patterns) dont il existe aujourd'hui de nombreux modèles regroupés en catalogues, le plus connu étant celui du GoF ("Gang of Four", auteurs de [Gamma et al., 1999]).

Par exemple, le patron « procuration d'entrée/sortie » (ou proxy) a été appliqué aux objets capteurs et actionneurs qui représentent les liens physiques avec le robot. Ce patron crée un mandataire à la place du capteur qui va nous permettre, selon le mode choisi, de relier les algorithmes de commande soit au robot physique, soit à un modèle simulé du robot (maquette java3D) sans avoir à modifier tous les objets qui utilisent l'objet capteur. D'autres fonctions sont permises par l'utilisation d'un mandataire comme le traitement du signal (le mandataire filtre les données du capteur) ou la protection contre les accès concurrents (cf. 2.2.2.2).

Le patron « stratégie » a également été appliqué pour séparer la partie contrôle et la partie purement algorithmique. Les classes qui implémentent les lois d'asservissement ou de génération de mouvement comprennent par exemple plusieurs méthodes et attributs liés à leur exécution (lancement de la tâche et périodicité par exemple) et à la communication, indépendamment de la méthode réalisant le calcul. Le patron stratégie « définit une famille d'algorithmes, encapsule chacun d'entre eux et les rend interchangeables. Le modèle Stratégie permet aux

algorithmes d'évoluer indépendamment des clients qui les utilisent. » [Gamma et al., 1999]. De plus, cette solution architecturale permet de disposer de plusieurs algorithmes et de choisir dynamiquement celui qui convient le mieux selon la tâche à effectuer.

2.2.1.2 *Cadre d'application*

L'ajout de ces points de connexion nous a permis de séparer, d'un coté, ce qui constituait le cœur fonctionnel du contrôleur de robot, et d'un autre coté, les objets et le code plus spécialisés concernant les algorithmes de commande et les interfaces avec le système articulé (cf. Figure 2). Tous ces concepts nous ont ainsi amené à définir ce que l'on nomme un cadre d'application (framework), un ensemble de classes abstraites dont les interactions sont définies et avec lequel le développeur peut créer simplement des applications spécifiques en injectant du code spécialisé.

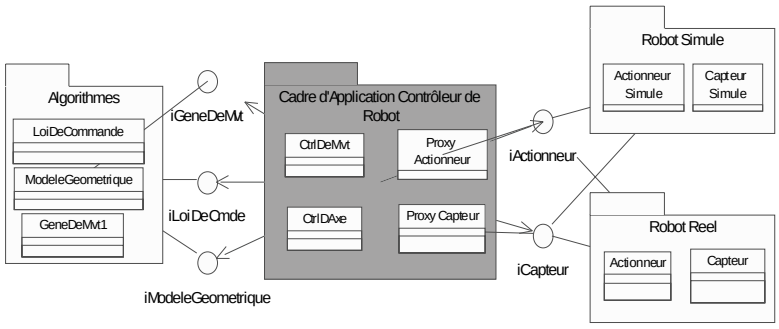

Figure 2. *Cadre d'application "Contrôleur de Robot" (vue simplifiée)*

2.2.2 Architecture dynamique

Le comportement temporel du contrôleur et surtout sa prédictibilité font partie des propriétés essentielles à vérifier dans le cadre des systèmes temps-réels. Il faut d'abord identifier les ressources nécessaires pour l'exécution du logiciel (tâches) puis mettre en œuvre les mécanismes de synchronisation et de communication.

2.2.2.1 *Expression de la concurrence*

Dans l'approche objet, chaque objet a, par construction, une structure propre, et habituellement, on lui associe une machine à états décrivant son comportement. Par conséquent, tous les objets sont potentiellement en concurrence. Faut-il alors associer une ressource d'exécution à chaque objet ? Heureusement non, et il est nécessaire d'identifier clairement quelles sont les sources de comportement concurrent, ce qui n'est pas toujours facile. De plus, en pratique, un objet pourra nécessiter aucune, une ou même plusieurs ressources d'exécution concurrentes.

Cette concurrence est exprimée de plusieurs façons :

– explicitement : en UML, une classe sera soit active soit passive. Chaque instance d'une classe active est associée à au moins un flot d'exécution propre. Les autres objets, dit passifs, ne sont essentiellement utilisés que pour la manipulation de données et s'exécutent sous le contrôle des objets actifs qui interagissent avec eux ;

– implicitement dans le cas où, par exemple, les machines à états font référence à des états concurrents (composites ou imbriqués) dans lesquels plusieurs activités sont suivies au même moment.

Au niveau implantation, sur une plateforme monoprocesseur, une ressource d'exécution sera représentée par une tâche qui correspond à une exécution séquentielle de fonctions (une méthode d'un objet par exemple). Par contre, il est difficile de connaître a priori, par exemple, dans quel flot d'exécution (au sein de quelle tâche) vont s'exécuter les activités associées aux autres méthodes. Dans le paragraphe suivant, nous tentons de prendre en considération les différents cas qui peuvent se présenter.

2.2.2.2 *Communication entre objets*

Dans l'approche objet, les communications se font par échanges de messages, habituellement représentés par des appels de méthodes.

Considérant un cas simple de communication d'un objet actif A vers un objet B, si tout d'abord, l'objet B est passif, la communication sera généralement de type synchrone, c'est-à-dire que l'objet A appelle une méthode de l'objet B et l'exécute au sein de son flot d'exécution (cf. 2.2.2.1). Si l'objet A est le seul à accéder à l'objet B qui ne communique pas avec d'autres objets, aucun problème de concurrence ne peut apparaître. Par contre, si l'objet B est accédé depuis plusieurs objets actifs (ie. plusieurs tâches) de manière asynchrone ou si il est lui-même actif, son intégrité est à contrôler. En effet, tout attribut d'un objet, modifiable à l'aide d'une méthode de celui-ci, devient une ressource critique. En UML, à travers l'attribut *concurrency*, on peut indiquer pour chaque opération si elle peut être utilisée de manière concurrente ou non. Il faudra alors peut-être interdire l'accès concurrent à l'aide de mécanismes d'exclusion mutuelle par exemple ou réaliser un appel asynchrone. UML2.0 a d'ailleurs introduit la notion de signaux qui sont traités de cette manière par envoi / réception d'évènements. Ainsi, une approche générique d'implémentation de ces objets consiste à leur associer une file d'évènements et chaque appel de méthode publique ne fait qu'y déposer un message. La tâche associée réceptionne alors les requêtes de manière séquentielle lorsqu'elle l'a prévu.

2.2.2.3 *Indépendance vis-à-vis de la plateforme*

Dans les langages de haut niveau tels que Java ou Ada, il est possible d'indiquer si une méthode sera protégée ou non contre les accès concurrents. Avec d'autres, comme dans notre cas, l'utilisation du langage C++ et de l'exécutif VxWorks, ces mécanismes nécessitent l'appel de primitives spécifiques fournies par l'exécutif.

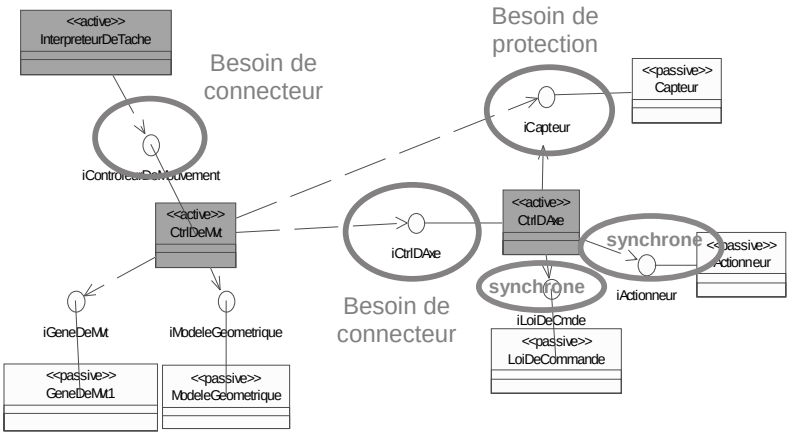

Figure 3. *Architecture dynamique et représentation des besoins*

Par conséquent, si nous modifions dès le modèle afin d'y introduire des éléments spécifiques à notre plateforme, il en deviendra dépendant. Pour rendre le modèle indépendant de cette plateforme tout en explicitant les besoins (voir Figure 3), plusieurs solutions sont envisageables.

La première inclut l'utilisation d'une API[1] standard comme Posix ou Apex ou d'une définie par le concepteur. Changer de plateforme (de système d'exploitation par exemple), supposera alors de redéfinir une implémentation de cette API.

La seconde consiste à modéliser, d'un coté, les besoins du logiciel (en ajoutant des attributs au modèle : tagged values), et d'un autre, les mécanismes fournis par la plateforme (sous forme de patterns) ainsi que les besoins qu'ils couvrent. Le choix est alors laissé au concepteur pour sélectionner et associer les modèles. Par exemple, il est possible de considérer trois types de communication : synchronisation (envoi / réception d'évènement), échange de donnée et transfert de donnée. Une synchronisation peut être implémentée en Posix en utilisant un événement, un sémaphore binaire ou plus simplement par polling en scrutant une variable booléenne. Au concepteur de choisir le mécanisme qu'il souhaite, et d'appliquer le pattern correspondant (qui va créer par exemple les objets et les appels de primitives nécessaires). Cette solution présente l'avantage d'utiliser les fonctions natives de l'exécutif (meilleures performances). De plus, des règles pourraient être vérifiées pour contrôler et valider les choix du concepteur.

Finalement, les deux solutions présentées rejoignent l'idée d'une bibliothèque générique (ou API), de fonctions pour la première et de patterns pour la seconde plus en accord avec l'approche MDA. Dès lors, nous avons entrepris la séparation et

[1] Applications Programming Interface

la modélisation des éléments spécifiques à la plateforme afin de mieux maîtriser le comportement temporel du contrôleur suite à des évolutions architecturales. Il s'agira ensuite de bien identifier quel composant de la plateforme utiliser pour remplir les besoins (de communication, d'exécution) de l'architecture dynamique.

2.3. *Modèle de la plateforme*

Dans cette partie il s'agit de modéliser les services spécifiques à la plateforme qui sont utilisés pour remplir les besoins de l'architecture dynamique (cf. 2.2.2).

Étant donné que l'environnement de modélisation utilisé n'intègre pas encore de solution de transformation de modèles à modèles facilement configurable les modèles ont été réalisés en vue de générer du code directement à travers une bibliothèque d'objets des mécanismes offerts par l'exécutif.

2.3.1 Bibliothèque temps-réel

A partir des librairies écrites en C fournies avec l'exécutif temps-réel VxWorks nous avons créé une bibliothèque de classes permettant d'utiliser de manière simplifiée les mécanismes offerts par le système temps-réel (cf. Figure 4) : tâche, événement (utilisant un sémaphore binaire), sémaphore d'exclusion mutuelle, chien de garde, activations périodiques et gestion du temps.

Par exemple, pour répondre aux besoins des objets actifs, il est nécessaire de pouvoir créer des tâches périodiques ou apériodiques (déclenchées sur événements), ce que permettent les classes de base Processus, ProcessusPeriodique et ProcessusDeclenche (cf. Figure 4). Chaque objet actif hérite de l'une de ces classes selon le type d'activation de l'objet. A l'instanciation de l'objet, une tâche sera lancée, exécutant une méthode virtuelle redéfinie dans la sous-classe correspondante à l'objet actif.

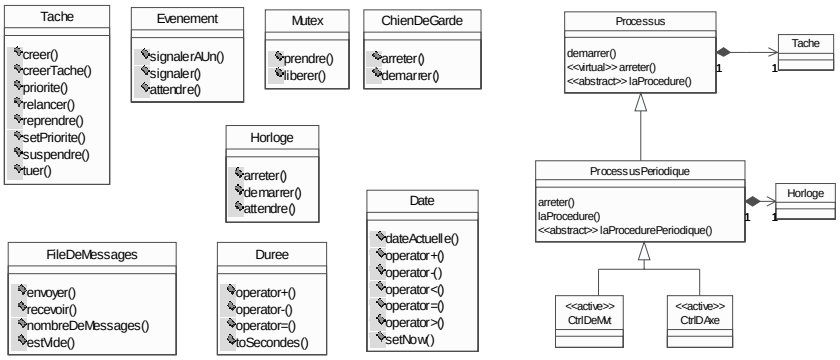

Figure 4. *Bibliothèques d'objets VxWorks*

2.3.2 Connecteurs : éléments de communication

Afin de gérer les flots de données entre objets actifs et au lieu de protéger l'accès aux méthodes des objets actifs, nous avons préféré mettre en place un objet partagé ayant une interface générique, la classe ElementDeCommunication (Figure 5) utilisée comme classe d'association. Deux méthodes, lire() et ecrire() permettent de mettre à jour ou de récupérer la valeur de cette variable partagée. Ensuite, selon les besoins, le concepteur choisit parmi les objets offerts de son exécutif. Par exemple, la classe VariableProtegee, qui implémente cette interface, associe un mécanisme de protection des données (sémaphore d'exclusion mutuelle) au travers de ces méthodes, garantissant qu'aucun processus ne pourra passer outre et la classe BoiteAuxLettres utilise une file de messages.

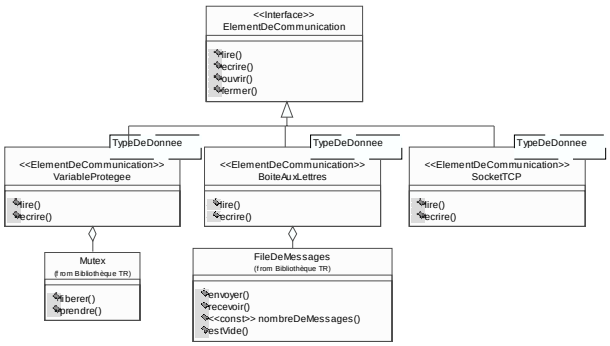

Figure 5. *Classe d'association ElementDeCommunication*

2.4. *Modèle final*

Le modèle d'implémentation final est déduit des deux modèles précédents (cf. 2.2 et 2.3). Les éléments du modèle de la plateforme sont utilisés pour répondre aux besoins identifiés sur le modèle indépendant de la plateforme (cf. Figure 3). Dans notre cas, nous nous sommes limités aux besoins concernant l'exécution temps-réel du logiciel, à savoir l'exécution concurrente, la communication et la synchronisation. Par exemple, les classes actives héritent de la classe Processus afin de leur associer une tâche au sens de l'exécutif choisi (VxWorks). Concernant le besoin de protection identifié pour la classe Capteur, un nouveau mandataire est utilisé mettant en œuvre un sémaphore d'exclusion mutuelle pour conditionner l'accès à l'objet Capteur. Les communications de données, précédemment réalisées par appel de méthode, se font maintenant à travers des objets de la classe ElementDeCommunication (cf. 2.3.2). Il s'agit plus que d'une simple association de modèles et l'obtention du modèle final nécessite l'application de nouveaux patrons de conception réalisant la fusion entre les modèles. La transformation de modèles représente donc réellement un point important dans ce type de démarche.

3. Vérification et validation

Comme nous l'avons énoncé en introduction les modèles peuvent servir de support pour les activités de vérification et de validation du système (logiciel dans notre cas).

Certaines analyses peuvent être conduites directement en utilisant les modèles précédents (cf. section 2). Par exemple, des simulations comportementales, à travers l'animation des machines à états et des interactions entre les objets, sont utilisées pour vérifier les scénarios attendus.

D'autres analyses nécessitent l'extraction ou la transformation des modèles réalisés dans un formalisme adapté à un type de vérifications désirées. Par exemple, il serait avantageux de pouvoir déduire un réseau de Petri global modélisant le comportement du système afin de mettre en œuvre les outils de preuve formelle associés.

Dans cette section, nous présentons un exemple portant sur l'analyse d'ordonnancement de l'architecture dynamique. Nous avons pour cela préféré le langage AADL ([SAE, 2004]) au langage UML. En effet, ce nouveau langage offre une syntaxe et une sémantique permettant d'exprimer formellement le comportement temporel des objets : temps d'exécution des tâches, périodes d'activation, ressources partagées et algorithmes d'ordonnancement utilisés, etc...

3.1. *Présentation du langage AADL*

AADL est un langage émergeant développé sous l'autorité de la SAE (Society of Automotive Engineers) pour répondre aux besoins spéciaux des systèmes embarqués temps-réel critiques tels que les systèmes avioniques. En particulier, le langage peut décrire des vues fonctionnelles de composants échangeant des flots de contrôles ou de données, et leur associer des aspects non-fonctionnels précis tels que les exigences temporelles, les modèles de fautes et d'erreurs, le partitionnement temporel et spatial et les propriétés de sûreté et de certification.

La description d'une architecture en AADL consiste essentiellement à la représentation d'une architecture dynamique de logiciel associée et liée à une plateforme d'exécution. En pratique, elle se décrit à l'aide de composants et de leur composition. Cette composition est :

- hiérarchisée, les composants peuvent contenir d'autres composants et créer des sous-systèmes représentables sur une arborescence ;

- interconnectée, les composants communiquent entre eux par des liens fonctionnels (flots de données et de contrôles) et par des liens physiques (bus, mémoire, réseau, …) ;

Dix catégories de composants ont été définies dans le standard selon trois groupes :

- composants logiciels : data, subprogram, thread, thread group et process;

- composants matériels : processor, memory, bus, device ;

- composants système : system (élément composite pouvant contenir des éléments des deux groupes précédents).

Les caractéristiques non fonctionnelles, c'est-à-dire qui n'apparaissent pas directement au niveau du code, sont modélisées à l'aide de propriétés assez proches des tagged values d'UML2.0. Elles peuvent exprimer par exemple des contraintes, ou des hypothèses qui garantissent la cohérence des composants.

Dans le domaine des systèmes embarqués, certaines propriétés non fonctionnelles sont traitées comme des propriétés fonctionnelles importantes, déterminantes même, pour choisir une architecture ; c'est notamment le cas des propriétés temporelles. Le standard AADL prédéfinit plusieurs propriétés de ce type (cf. Figure 6). Le type et les unités des données peuvent être clairement définis.

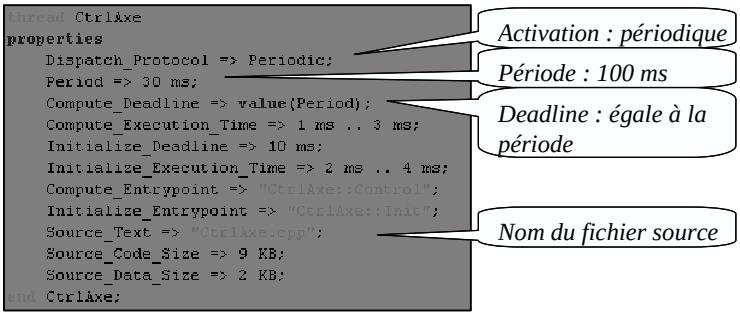

Figure 6. *Exemple de propriétés associées à un thread en AADL (cf. 3.2)*

3.2. *Transformation des modèles UML en AADL*

Afin de réaliser des analyses d'ordonnancement, nous nous limitons aux composants AADL suivant :

- process : représente un espace d'adressage virtuel ; il contient le code binaire d'une application et peut être protégé à l'exécution contre des écritures ou des lectures non autorisées ;

- thread : représente un flot d'exécution parallèle ;

- data : représentent les variables statiques présentes dans le code source. Elles peuvent être partagées entre les threads ou les process et des propriétés (cf.1.4.6) sont utilisées pour définir le type de mécanisme employé pour gérer les accès concurrents ;

- processor : représente une abstraction du matériel et du logiciel responsable de l'ordonnancement et de l'exécution des threads.

Ainsi, à partir de l'architecture dynamique définie au chapitre 2, le modèle AADL (cf. Figure 7) a été construit selon un procédé automatisable, en associant par exemple un composant thread à chaque objet actif et un composant data à chaque ElementDeCommunication partagé. La cible temps-réel est composée d'un seul processeur relié aux capteurs et aux actionneurs par l'intermédiaire de deux cartes d'acquisitions.

Les valeurs des propriétés concernant le type d'activation (périodique, sporadique, apériodique, background, …), les périodes, les échéances et les temps d'exécution ont ensuite été rajoutées puisqu'elles n'apparaissaient pas dans le modèle UML.

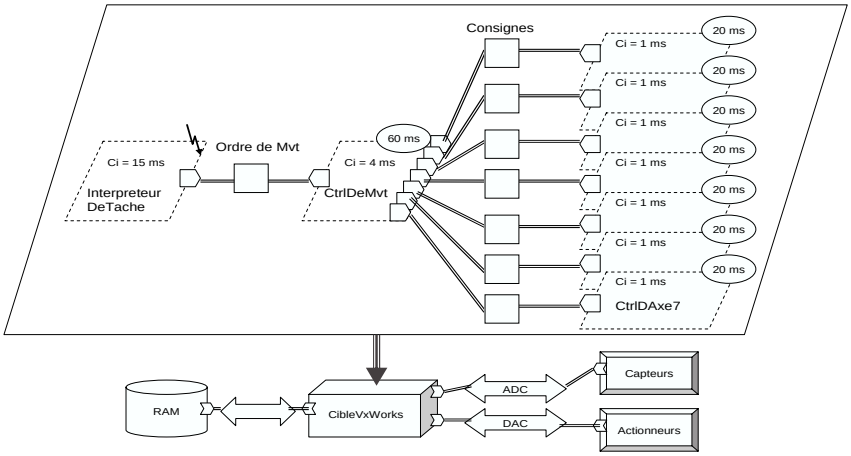

Figure 7. *Architecture dynamique décrite en AADL*

3.3. *Illustration : analyse d'ordonnancement*

La sémantique du langage AADL et les informations formalisées au sein du modèle peuvent ensuite être exploitées par des outils d'analyses, de simulation et de vérification. A l'heure actuelle, ce genre d'outils est encore rare. L'université de Brest a cependant adapté l'outil Cheddar ([Singhoff et al., 2004]), simulateur d'ordonnancement. Nous avons ainsi pu vérifier différents algorithmes et les simuler. Pour illustration, sur la Figure 8, l'exécution temporelle du contrôleur de robot a été estimée en appliquant un ordonnancement de type rate monotonic que nous avons pu comparer aux traces mesurées après codage sous l'outil WindView pour VxWorks.

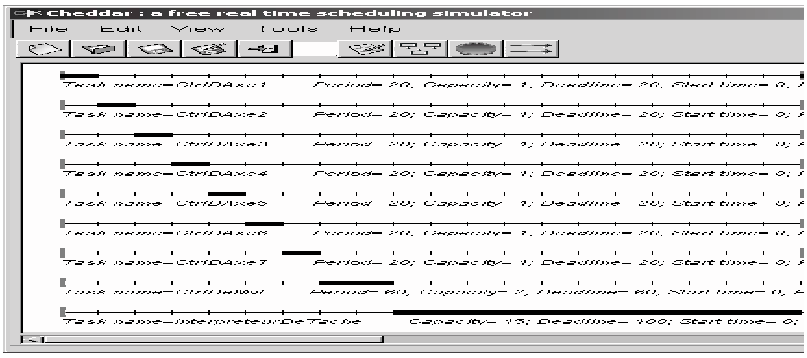

Figure 8. *Exemple de simulation d'ordonnancement avec l'outil Cheddar*

4. Résumé de la démarche

Les approches basées modèles nécessitent avant tout un processus de développement bien défini et une méthode rigoureuse. Avec le souci de mettre en œuvre une méthodologie permettant de rendre la conception et l'implémentation des logiciels de contrôle de robots les plus transparentes possibles, nous avons élaboré une démarche progressive permettant de favoriser la transformation des besoins utilisateurs en une solution concrète.

Ainsi, à partir d'une analyse des besoins guidée par les cas d'utilisation, il nous a paru assez facile d'identifier un premier modèle objet de haut niveau que nous avons appelé **modèle d'analyse** (cf. 2.1), qui correspond au modèle CIM[2] de la démarche MDA [OMG, 2003]. Celui-ci pourra servir de référence à la compréhension des fonctionnalités du logiciel.

Ensuite, nous avons montré que la transformation de ce modèle orienté métier vers un modèle structurel d'implantation pouvait être réalisé par itérations successives, chacune orientée vers l'apport de solutions améliorant la qualité du logiciel en vue de sa réutilisation par exemple (cf. 2.2.1). L'étape suivante a alors consisté à définir les contraintes dynamiques du système en identifiant d'abord les objets actifs puis les besoins de synchronisation et de communication qui en découlent. Cette étape fige le **modèle d'implémentation indépendant de la plateforme (PIM[3])**.

La prise en compte du support d'exécution et de l'exécutif temps-réel utilisés a servi à définir une bibliothèque d'**objets de la plateforme (PDM[4])**. Le **modèle**

[2] CIM = Computation Independent Models (cf. [OMG, 2003])
[3] PIM = Platform Independent Models (cf. [OMG, 2003])
[4] PDM = Platform Description Models (cf. [OMG, 2003])

d'implémentation spécifique à la plateforme (PSM[5]) est le résultat de la transformation du modèle d'implémentation indépendant de la plateforme après que l'on ait sélectionné les objets de la plateforme répondant aux besoins identifiés (concurrence, synchronisation, communication).

Enfin, nous avons apporté l'exemple d'une autre transformation dans laquelle un changement de formalisme était effectué afin de pratiquer des analyses spécifiques d'ordonnancement sur le modèle final.

Cette démarche, présentée Figure 9, a été illustrée dans le cadre de la conception du contrôleur d'un prototype de robot.

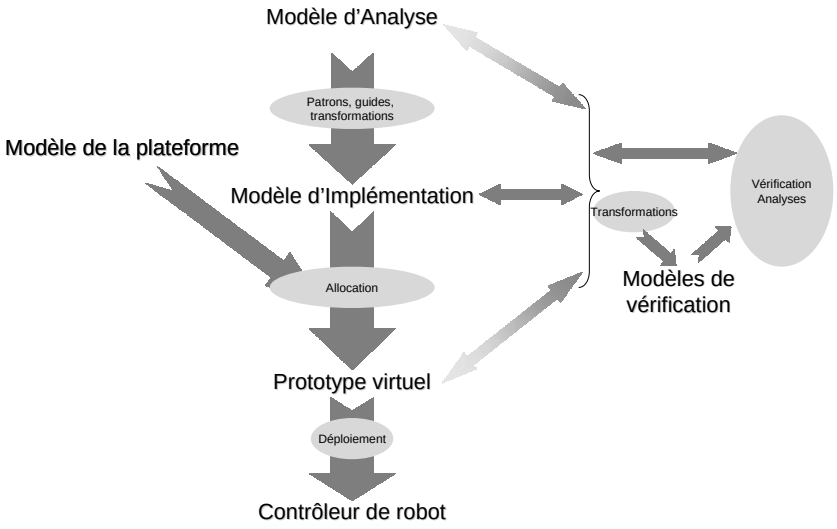

Figure 9. *Démarche de conception appliquée au contrôleur de robot*

5. Conclusion

Dans le domaine de la robotique de service, la satisfaction des besoins de sécurité essentiels sont avant tout dépendants de la parfaite maîtrise du robot. Nous nous efforçons de mettre en œuvre des mécanismes de traitement et de tolérance aux fautes orientés vers la sécurité du personnel. Or, le logiciel de contrôle du robot a une place primordiale dans la réalisation de ces fonctions. L'approche objet nous avait déjà permis d'améliorer la modularité, la flexibilité et l'indépendance vis-à-vis des algorithmes et des interfaces matérielles du contrôleur de robot. La modélisation de la plateforme apporte un meilleur contrôle de son fonctionnement grâce à la prise en compte explicite, au niveau modèle, des problèmes d'implantation. Les modèles

[5] PSM = Platform Specific Models (cf. [OMG, 2003])

représentent alors un support efficace à ces approches et leur utilisation tout au long du cycle de développement, de l'analyse des besoins à l'implantation, a facilité la mise en place d'une construction itérative en pratiquant par transformations et vérifications successives.

Bibliographie

Carroll L.F., « Vers la maîtrise du développement d'un contrôleur temps réel sûr de fonctionnement pour les robots manipulateurs », Thèse de doctorat, 1999, Institut National des Sciences Appliquées de Toulouse.

Favre J-M, Estublier J., Blay-Fornarino M., « L'ingénierie dirigée par les modèles. Au-delà du MDA », Traité IC2, série Informatique et Systèmes d'Information, Editions Lavoisier, 2006.

Gamma E., Helm R., Johnson R., et Vlissides J., « Design Patterns : Catalogue de modèles de conception réutilisables », 1999, Vuibert.

Guiochet J., « Conception Orientée objet d'un contrôleur de robot », Rapport DEA, 1999, Institut National des Sciences Appliquées de Toulouse.

Niemann S., « Executable Systems Design with UML2.0 », I-Logix™ www.ilogix.com, 2004

Object Management Group (OMG), « Model Driven Architecture Guide », version 1.0.1, juin 2003

Object Management Group (OMG), « UML Superstructure Specification », version 2.0, juillet 2004

Schraft R.D., Schmierer G., « Service Robot », AK Peters, Natick, MA (USA), 2000.

Singhoff F., Legrand J., Nana L., Marcé L., « Cheddar : a Flexible Real Time Scheduling Framework », *ACM Ada Letters journal*, 24(4):1-8, ACM Press. Also published in the proceedings of the ACM SIGADA International Conference, Atlanta, 15-18 November, 2004.

Society of Automotive Engineers (SAE) Aerospace Avionics Systems Division: « Architecture Analysis & Design Language Standards Document », version 1.0, nov. 2004.

Thomas D., « Analyse et mise en œuvre d'un contrôleur "générique" de robot manipulateur », Rapport DEA, 2004, Institut National des Sciences Appliquées de Toulouse.

Tondu B., Ippolito S., Guiochet J., Diadie A., « A Seven-degrees-of-freedom Robot-Arm Driven by Pneumatic Artificial Muscles for Humanoid Robots », *International Journal of Robotics Research*, vol. 24, n°4, MIT Press (April 2005), p. 257-274.