IMOCA : une architecture à base de modes de fonctionnement pour les systèmes de contrôle de processus

Goulven Guillou, Jean-Philippe Babau

IMOCA : une architecture à base de modes de fonctionnement pour les systèmes de contrôle de processus

Goulven Guillou *, Jean-Philippe Babau *

*Lab-STICC/UMR 6285, UBO, UEB
20 Avenue Le Gorgeu
29200 Brest, France
goulven.guillou@univ-brest.fr
jean-philippe.babau@univ-brest.fr

1 Introduction

Les systèmes de contrôle de processus sont de plus en plus utilisés pour assister l'activité humaine, y compris dans des environnements incertains. Un système de contrôle utilise des capteurs pour obtenir des informations sur le processus à contrôler et sur son environnement physique (voir figure 1). A partir de ces informations, des commandes sont déterminées par le système de contrôle puis appliquées au processus via des actionneurs. De tels systèmes sont souvent spécifiques à chaque domaine, voire à chaque processus à cause des contraintes liées à l'environnement. Les engins spatiaux, les véhicules terrestres ou lunaires, les bateaux ou encore les drones sont des exemples typiques de processus visés par l'étude.

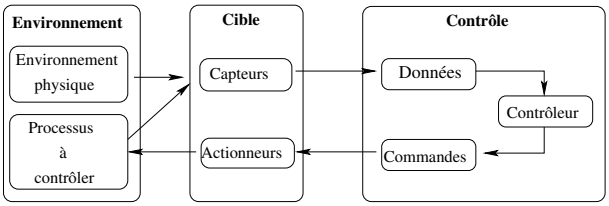


FIG. 1 – *Système de contrôle.*

De par les spécificités des processus à contrôler, les systèmes sont souvent développés au cas par cas, ce qui pose des problèmes de maîtrise, de mise au point du logiciel et au final de déploiement. Dans ce contexte, disposer d'outils logiciels d'aide au développement et des outils de simulation devient crucial. Les méthodes orientées objets et, plus récemment, l'ingénierie dirigée par les modèles (IDM (Bézivin, 2006; Terrier et Gérard, 2006)) offrent un cadre pour la maîtrise du développement de ce type d'applications. Un des points clé est la maîtrise de l'architecture, soit l'organisation et l'interaction des divers composants logiciels du système. L'architecture à base de modèle SAIA (Deantoni et Babau, 2006), qui s'appuie sur une approche MDA (Model Driven Architecture (OMG, 2003)), se concentre plus particulièrement

sur la séparation des préoccupations en proposant trois modèles appelés PIM[1], PM[2] et PSM[3]. Dans SAIA, le modèle PIM est le modèle qui intègre les politiques de contrôle au travers de lois de commande. Pour les systèmes visés soumis à des perturbations fortes, le contrôle est très complexe à mettre au point. Il doit être adaptatif au contexte en faisant intervenir plusieurs lois de commande. On doit ainsi pouvoir agir de différentes manières sur un même actionneur selon le contexte environnemental. Dans le domaine du contrôle, la gestion des diverses lois de commande se fait classiquement via l'introduction de modes de fonctionnement (Feiler et al., 2004; Borde et al., 2009) : un mode détermine les lois de commande à appliquer à un instant donné. Une fois ce cadre conceptuel posé, un des problèmes majeur et complexe auquel est confronté l'architecte logiciel est la définition de ces modes de fonctionnement et leur paramétrage. Ceci nécessite de répondre aux trois questions suivantes :

1. Quels sont les modes de fonctionnement pertinents selon les contextes ?
2. Comment enchaîner les modes ?
3. Comment adapter les modes en fonction du contexte ?

Pour répondre à ces questions, le document présente l'architecture à base de modèles IMOCA (pour archItecture for MOde Control Adaptation). L'architecture IMOCA est un extension de SAIA. Et, parce qu'il est important de séparer l'application de contrôle du processus lui-même, IMOCA est basée sur trois couches appelées cible, interprétation et contrôle. Le contrôleur est lui-même composé de trois contrôleurs dits `réactif`, `expert` et `adaptatif` fonctionnant de manière parallèle et asynchrone. Le contrôleur réactif est chargé de l'application d'une loi de commande choisie par le contrôleur expert (choix du mode) et paramétrée par le contrôleur adaptatif (paramétrage du mode).

La suite de ce document est organisée comme suit : après une présentation de l'état de l'art, l'architecture IMOCA est décrite ainsi qu'une démarche de mise en place d'IMOCA, démarche destinée à aider le concepteur du système à choisir, combiner et paramétrer les modes. Enfin, en guise de validation, une solution pour le cas du pilotage automatique des voiliers est présentée.

2 Etat de l'art

Il existe de nombreuses approches IDM pour construire, modéliser et vérifier des systèmes de contrôle de processus. Comme le rappelle (Terrier et Gérard, 2006), l'application des principes de l'IDM aide à la maîtrise du développement des systèmes visés en prônant la séparation des préoccupations et l'automatisation des phases d'intégration, de validation et de génération de code. En suivant ces principes, IMOCA s'intéresse aux deux aspects principaux des systèmes de contrôle que sont la communication avec l'environnement et la politique de contrôle.

Dans une approche IDM pour les systèmes de contrôle, les études préconisent une séparation claire entre la plate-forme de communication (ici les capteurs et les actionneurs) et l'application (ici le contrôle) (Agrawal et al., 2004). En particulier, basé sur le paradigme MDA (OMG, 2003; Deantoni et Babau, 2005), le style architectural SAIA (Deantoni et Babau, 2006) sépare les entrées/sorties de haut-niveau (les données du contrôle) des données de bas niveau

[1] Platform Independent Model
[2] Platform Model
[3] Platform Specific Model

(le monde vu au travers des capteurs/actionneurs). Cette dernière vue correspond à une plate-forme abstraite explicite (Almeida et al., 2005). Pour cette partie, IMOCA reprend globalement les mêmes principes que ceux développés dans SAIA.

Au niveau du contrôle, on trouve dans la littérature diverses approches pour intégrer des politiques d'adaptation au contexte. Tout d'abord, l'introduction de modes de fonctionnement est classique dans les applications de contrôle. Ils permettent via des automates de définir une politique d'adaptation en fonction d'événements, qui peuvent être reliés au contexte. Il existe beaucoup de travaux concernant la mise en place de modes dans les architectures, comme par exemple ceux liés à AADL (Feiler et al., 2004). Ces travaux apportent soit des outils de formalisation à base d'automates, par exemple des réseaux de Petri temporisés, pour la validation du comportement (absence de deadlock, comportement temporel) des systèmes comme dans (Bertrand et al., 2008) ; soit des outils de gestion de la reconfiguration des modes à l'exécution comme dans (Borde et al., 2009). L'objet n'est pas ici de citer tous les travaux sur l'introduction de modes de fonctionnement, mais on retiendra l'intérêt de leur introduction comme élément de base d'un contrôleur adaptatif au contexte.

Pour autant, l'adaptation au contexte ne peut se baser sur la seule introduction de modes de fonctionnement. Comme le précise (Horn, 2001), il est aussi nécessaire d'introduire plusieurs niveaux de contrôle au travers de divers types de contrôleurs. (Brun et al., 2009) décrit plusieurs politiques d'adaptation au contexte dont le type *Model Reference Adaptive Control* en s'appuyant sur une observation de l'environnement et une expertise du contrôle pour assurer une robustesse dans le comportement adaptatif.

D'autres travaux comme (Maker et al., 2013) montrent qu'à l'exécution, l'adaptation peut s'appuyer sur un simple paramétrage de l'application en fonction du contexte. Dans ce cadre, la difficulté réside dans l'évaluation et la mise au point de paramètres pertinents pour pré-tabuler ce que l'on appelle des *look-up tables* qui seront embarquées dans le système. Le nombre de paramètres est forcément limité pour des raisons de performance et de prédictibilité.

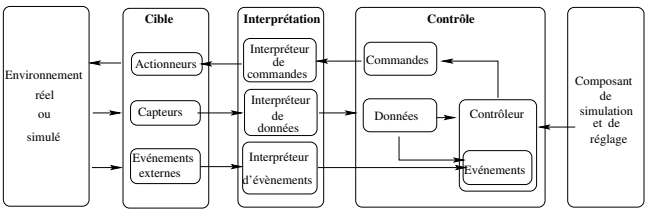
On retient de ces études qu'un contrôleur adaptatif au contexte doit intégrer des modes de fonctionnement, diverses politiques de contrôle, mais aussi être paramétrable. IMOCA est une architecture (Garlan, 2000) qui a pour objectif de répondre à tous ces points en intégrant les divers paradigmes précités. IMOCA propose des modèles et outils pour le choix, la mise au point et le paramétrage des modes et des contrôleurs. En particulier, IMOCA propose un composant de mise au point des tables pour le contrôle (Li et al., 2006) par simulation via une interface dédiée. IMOCA s'appuie sur les principes de l'IDM.

3 Architecture IMOCA

3.1 Principes de l'architecture IMOCA

L'architecture IMOCA est basée sur SAIA (Deantoni et Babau, 2005) qui est elle-même une approche MDA pour permettre le développement d'un système de contrôle de processus indépendamment d'une technologie de capteurs et d'actionneurs. Cette indépendance est primordiale pour limiter l'impact d'un changement de capteur ou d'actionneur sur l'application. En effet, dans un environnement fortement perturbé, c'est-à-dire soumis à de rapides et importantes variations (température, pression, accélération ...) les pannes peuvent être courantes. De plus, dans un contexte économique concurrentiel, les optimisations de matériel sont inces-

santes. L'architecture IMOCA présentée figure 2 est constituée de trois couches dites `Cible`, `Interprétation` et `Contrôle`. La `cible` est un modèle de la plateforme matérielle et le `contrôle` utilise une vue idéale de l'environnement, les `données`, pour prendre des décisions appelées `commandes` qui sont transmises aux `actionneurs`. L'adaptation entre la `cible` et le `contrôle` est réalisée par la couche d'`interprétation` qui se charge de faire le lien entre `capteurs` et `actionneurs` d'un côté, et `données` et `commandes` de l'autre côté. Cette configuration assure l'indépendance de la partie contrôle vis-à-vis de la plateforme matérielle. De plus, certains `événements externes` peuvent intervenir (panne de capteur, changement de consigne par exemple). Ils influencent directement le contrôle. Enfin,



Les cadres représentent des composants, les flèches décrivent des flux d'information.

FIG. 2 – *Principes de l'architecture IMOCA.*

un composant de simulation et de réglage est ajouté fournissant les outils nécessaires à la mise au point de l'application de contrôle. Il est apte à prendre la main sur cette dernière afin de permettre à un expert de tester différentes lois de commande, de valider leurs enchaînements et de règler leurs paramètres.

Les principaux composants de l'architecture IMOCA sont détaillés ci-après.

3.2 Données

Pour pouvoir choisir et paramétrer la bonne loi de commande, l'application de contrôle a besoin de se faire une image de l'environnement appelée contexte. La construction du contexte se fait en deux étapes. Dans la première, l'environnement est mesuré à l'aide de `capteurs`. Les données issues de ces capteurs sont ensuite transformées en `données` de haut-niveau. Au sein de la `cible`, le capteur se réduit à des spécifications purement techniques masquant les aspects bas-niveau. Un `capteur` se caractérise par (voir figure 3 où un capteur est dénoté `Sensor`) un nom `sensorName`, une valeur courante `value` de type `double`, une fréquence `frequency`. Des méthodes `set_data` et `get_data` de mise à jour et de lecture de la valeur du capteur existent pour chaque classe. Ces méthodes n'apparaissent pas dans le métamodèle car leur signature et comportement par défaut sont générées automatiquement. Le comportement est complété par la suite en fonction de l'expertise métier. Les `capteurs` délivrent des données brutes qu'il va falloir interpréter pour obtenir des `données` de haut-niveau. L'ensemble des `données` est déterminé par expertise du domaine d'application en répondant à la question "Quelles sont les données qui permettent de prendre des décisions par rapport au contrôle ?". Il s'agit de données de haut-niveau qui ne comportent aucun détail sur la manière de les obtenir. Les `données` ou `data` figure 3 sont caractérisées par un nom `dataName`, un `type`, une valeur `value`, une fréquence `frequency` ainsi que des

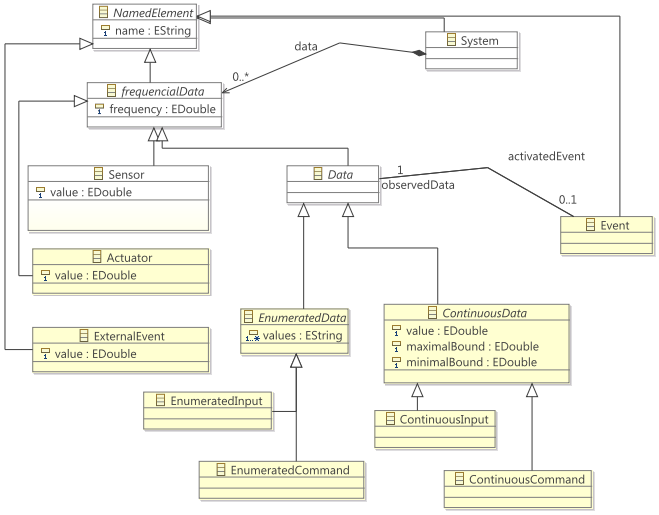


FIG. 3 – *Métamodèle simplifié des données, commandes, capteurs, actionneurs et évènements.*

opérations de mise à jour et de lecture. Deux grandes catégories de données sont distinguées, les données continues `ContinuousData` pouvant prendre toute valeur entre deux bornes `minimalBound` et `maximalBound` et les données de type énuméré `EnumeratedData` ne pouvant prendre qu'un nombre fini de valeurs. Certaines données, pour indiquer une modification de l'environnement, peuvent générer un événement.

L'`interpréteur de données` est chargé de faire le lien entre les `capteurs` et les `données` (figure 4). Les données issues des `capteurs` sont filtrées, on peut choisir par exemple de ne prendre qu'une donnée sur trois ou d'effectuer une moyenne sur fenêtre glissante ou encore d'utiliser un filtre de Kalman. Ensuite la valeur obtenue est formatée c'est-à-dire convertie dans une unité du Système International. Cela comporte un décalage d'offset ainsi qu'une mise à l'échelle par multiplication par un coefficient de calibration. Ensuite la valeur est stockée dans un buffer circulaire permettant des traitements d'ordre statistique : moyenne, écart type, pente de la droite de régression... Enfin une phase de combinaison permet de fournir une donnée à la couche de contrôle. Cette phase peut être réduite à la transmission de la valeur filtrée et formatée si la donnée nécessaire au contrôle correspond à la donnée délivrée par le capteur.

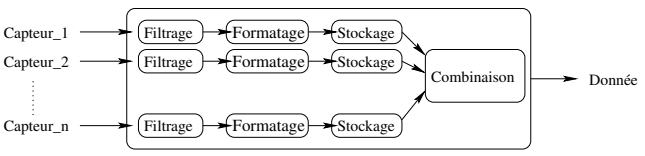


FIG. 4 – *La couche d'interprétation pour une donnée.*

A partir des données, le contrôle agit pour produire les commandes. La couche de contrôle est maintenant détaillée.

3.3 Contrôle

3.3.1 Principes

Le contrôle s'effectue à l'aide de lois de commande, chaque loi est associée à un mode de fonctionnement. L'application doit toujours être capable d'appliquer un mode dit courant. D'autre part, il est connu que les lois de commande sont très sensibles aux variations de l'environnement. Il s'avère donc nécessaire soit de changer de mode courant (pour répondre à la question 2 de l'introduction "Comment enchaîner plusieurs modes ?"), soit de modifier ses paramètres (pour répondre à la question 3 de l'introduction "Comment paramétrer les modes ?"). Pour cela, l'application de contrôle est elle-même composée de trois entités fonctionnant en parallèle et de manière asynchrone (figure 6). La première dite `reactiveController` est chargée à haute fréquence (fréquence du capteur le plus rapide, environ $100Hz$) du contrôle temps réel du système, elle utilise des données qui correspondent directement à celles des capteurs (filtre élémentaire sans phase de combinaison dans la couche d'interprétation) pour calculer les commandes à appliquer à partir du mode imposé par l'`expertController`. Ce dernier est contrôlé par un automate (liste de `Transition`) qui change d'état sur un `trigger` (événement) (figure 5). Enfin, l'`adaptativeController` ajuste, à basse fréquence (de l'ordre de $0.1Hz$), les différents paramètres de la loi de commande afin de la rendre plus performante par rapport à l'environnement mesuré.

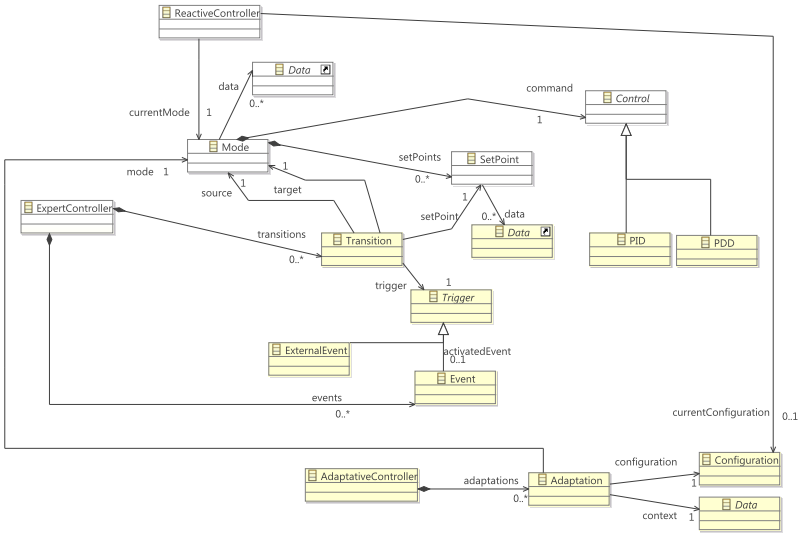


FIG. 5 – *Les différents éléments du contrôleur.*

3.3.2 Contrôleur réactif et modes

Les `Modes` (voir figures 5 et 6) se caractérisent par un nom `modeName`, un ensemble de paramètres `configuration`, un ensemble de consignes `setpoints` et une loi de commande `control`. Le contrôleur réactif effectue une boucle de type perception/action qui consiste à lire un certain nombre de données pour nourrir un mode imposé par le contrôleur expert et délivrer une commande. La loi de commande est une fonction qui dépend à la fois de paramètres, de données directement interprétables et de consignes qui proviennent du contrôleur expert. Par exemple, pour une régulation de type PID de vitesse d'un véhicule, les paramètres sont les coefficients P, I et D (voir figure 5), les données la vitesse du véhicule et la consigne est fournie par le conducteur. Des `data` sont dédiées à la surveillance de données capteurs (temps réel) qui sont sensées rester dans un certain intervalle de valeurs afin d'assurer une utilisation robuste de la loi de commande. En cas de dépassement un événement est généré. Dans le cas par exemple du pilotage d'un voilier, si la régulation se fait sur l'angle de gîte, il est important de surveiller le cap du bateau pour éviter de gros écarts de route (Guillou, 2010). Cette surveillance doit se faire sur des données temps réel pour obtenir une bonne réactivité car un changement de contexte est par essence lent à repérer. Pour une donnée issue d'un capteur, sortir de l'intervalle spécifié signifie que la loi de commande n'est plus adaptée. Le contrôleur expert commute alors sur une autre loi. Cette nouvelle loi peut être prévue dans l'enchaînement normal des lois pour l'accomplissement d'une tâche donnée, ou être une loi par défaut dans le cas où le comportement observé n'est pas conforme à l'expertise effectuée. Il est donc important de disposer d'au moins une loi robuste à la plupart des contextes pour pouvoir s'y réfugier en cas de nécessité.

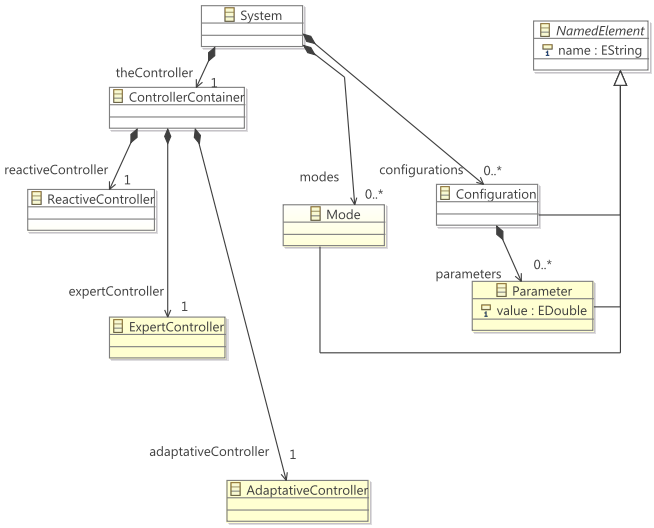


FIG. 6 – *Contrôle et modes.*

Pour prendre en compte les évolutions de l'environnement, il faut être capable d'adapter les paramètres des modes afin de maintenir un bon niveau de performance dans le nouveau

contexte. En effet, plutôt que de changer de mode, il se peut qu'un changement de paramètre soit plus pertinent. Par exemple, dans le cadre de la régulation en cap d'un voilier, l'augmentation de l'amplitude des angles de barre peut suffire pour répondre à l'augmentation de celle des vagues. C'est le contrôleur adaptatif présenté dans la partie suivante qui est chargé de cette adaptation.

3.3.3 Contrôleur adaptatif et table des paramètres

Le contrôleur adaptatif repère les caractéristiques basses fréquences de son environnement pour paramétrer les modes. Il utilise pour cela (figure 5) une table `Adaptation` donnant pour chaque mode et chaque contexte identifié (une `Data`) une `configuration`, c'est-à-dire les valeurs des différents paramètres intervenant dans le mode. La table est construite par expertise à l'aide notamment du composant de simulation et de mise au point. Un exemple de table des paramètres est la table 2 présentée dans la section 5.

Afin de garantir une utilisation cohérente de la table des paramètres il est nécessaire que deux contextes distincts ne puissent être valides simultanément. Enfin si aucun contexte proposé ne correspond au contexte courant il est important de prévoir des paramètres par défaut.

Le contrôleur expert, présenté maintenant, se charge de l'enchaînement des modes qui dépendent à la fois du type d'actionneur commandé, des objectifs à atteindre mais également de l'environnement.

3.3.4 Contrôleur expert et automate

Le contrôleur expert sélectionne la loi de commande et les consignes à appliquer. C'est un automate fini déterministe `Transition` (figure 5) muni d'un état initial où chaque état est associé à une loi de commande. Ses transitions sont étiquetées par des événements et lors d'un changement d'état (`trigger`) les consignes `setPoint` sont évaluées afin de pouvoir effectuer la commutation de modes. Ces consignes sont définies par expertise. Un exemple simple de comportement pour le contrôleur expert est modélisé figure 7 à l'aide de réseaux de Petri interprétés (David et Alla, 1992). Bien qu'il n'y ait pas ici de parallélisme sous-jacent, nous utilisons ce formalisme car il est couramment employé pour la modélisation des systèmes à événements discrets (Holloway et al., 1997). Une place correspond à un état et est associée à un mode. Nous avons ici un comportement simple basé sur deux modes. Le jeton indique le

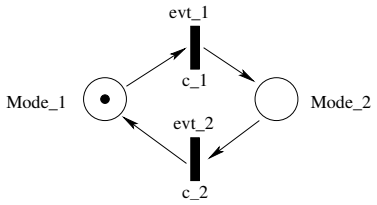


FIG. 7 – *Exemple de comportement simple pour le contrôleur expert.*

mode courant. Les transitions sont étiquetées par un événement `evt_i` et une fonction `c_i` servant à calculer les consignes. Lorsqu'un événement arrive et qu'une transition validée est étiquetée par cet événement, la fonction `c_i` est exécutée et la transition tirée.

3.3.5 Evènements

Les événements sont soit externes (mise en route du système ou changement de consigne par exemple), soit générés par certaines `data`. Ils indiquent un changement de contexte qui nécessite une ré-évaluation du mode courant, qui peut ne plus être approprié. Les événements provoquent l'évaluation de la fonction de transition de l'automate, la récupération des paramètres associés au mode obtenu et l'évaluation des consignes.

3.3.6 Modèles et code

A partir des modèles décrits, un générateur de code produit le code du système, ici en langage C/C++.

En fait, le générateur produit le code *glue* du système (l'appel de fonctions métiers) et une partie des contrôleurs. Le corps des fonctions *métier* reste très spécifique et est encapsulé dans des librairies. Elles concernent la couche `interprétation` (conversions usuelles, calibration, offset, outils statistiques, filtres ...), la couche `reactiveController` (PID, ...) et le composant de réglage des paramètres (éléments graphiques widgets, IHM ...). L'en-tête des fonctions est normalisé afin d'être intégrable (appelable) par le code généré.

La couche `expertController` est générée, elle correspond à une structure classique de code pour automate à base de *switch ... case*. La couche `adapatativeController` est de même entièrement générée, elle consiste à un calcul de contexte (appel d'une fonction métier) et au paramétrage de variables de contexte en fonction de valeurs pré-calculées et stockées dans des tables.

Le squelette du code et une partie des contrôleurs est donc obtenu par génération de code, puis le code final est obtenu par intégration et adaptation des librairies proposées. La figure 8 illustre cette démarche pour le cas d'une application de contrôle d'une maquette de char à voile. La maquette a été équipée d'un micro-contrôleur Arduino, d'une centrale inertielle UM6 et d'un servo-moteur permettant d'orienter la roue avant du char (la force propulsive étant elle fournie par le vent environnant). Le but est de diriger le char à voile dans une direction donnée en évitant les chavirages (les données de cap et de gîte sont récupérables via l'UM6). Le modèle `mySandYacht.imoca` (voir figure 8) est une instance simple d'`imoca` (la classe `adaptativeController` n'est ici pas instanciée). Le générateur de code Acceleo permet de produire un squelette de code (`sandChar.ino`) faisant référence aux librairies métiers nécessaires. Les librairies métiers (`imocaLibrary`, `arduinoLibrary` ...) intègre notamment des fonctions d'initialisation spécifiques (numéros des ports séries physiquement utilisés, fréquences des capteurs/actionneurs ...) afin d'obtenir un code compilable et effectivement exécutable sur la plateforme Arduino réelle.

4 Mise au point des modes

IMOCA ne garantit pas en soi que l'introduction des modes réponde aux trois questions posées en introduction :

1. Quels sont les modes de fonctionnement pertinents ?
2. Comment enchaîner plusieurs modes ?
3. Comment paramétrer les modes ?

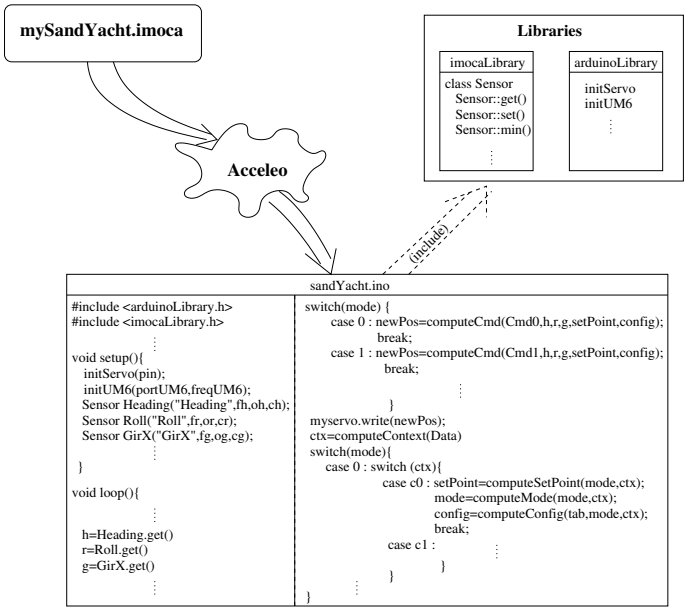


FIG. 8 – *Génération de code via Acceleo sous Eclipse.*

Le choix des modes de fonctionnement (question 1) possibles est d'autant plus difficile que de nombreuses données sont disponibles et combinables. Et la mesure de la qualité du contrôle est également un problème difficile car de nombreux critères, parfois qualitatifs plutôt que quantitatifs peuvent intervenir : robustesse, sûreté, sécurité, consommation, réponse à un échelon, conformité à une expertise effectuée de l'activité concernée... Le problème de l'enchaînement des lois de commande (question 2) est également difficile car dépendant de l'environnement, de la tâche à effectuer et de critères parfois difficilement quantifiables comme "avoir une belle trajectoire". Pour répondre à ce problème, nous proposons un outil d'expertise pour le choix des modes de fonctionnement (question 1), la manière de les enchaîner (question 2) et de les paramétrer (question 3). L'expert utilise l'outil de simulation et de mise au point de manière incrémentale par essais/corrections via le composant de simulation et de réglage. La mise au point se base sur 4 étapes :

Etape 1 mise à disposition d'un environnement réel ou simulé, construction de la couche `cible` et du `contrôleur réactif`

Etape 2 construction de la bibliothèque de lois de commande et de la table des paramètres

Etape 3 caractérisation des données ou contextes en fonction des capteurs

Etape 4 construction de l'automate et définition des évènements

La première étape a pour but de *jouer* avec le processus à contrôler. Le coût des plate-formes réelles ainsi que les contraintes et les risques liés à leur utilisation font qu'il est avantageux de simuler le processus ainsi que son environnement pour permettre le rejeu et ainsi tester différentes configurations. L'expert est d'autant mieux épaulé que la simulation est réaliste. Dans ce cadre, la réalité virtuelle par son côté immersif est particulièrement intéressante. Une fois cet environnement disponible la couche `cible` est développée ainsi que le contrôleur

réactif qui peut être en lien direct avec les capteurs dans un premier temps.

Ensuite la construction d'une interface permet de choisir et de construire différentes lois de commande. L'interface permet à l'expert de sélectionner celles qui semblent les plus pertinentes. Cette interface prend le rôle des `contrôleurs expert` et `adaptatif` et permet de définir les `données` nécessaires aux lois de commande sélectionnées. L'expert teste les lois dans différentes conditions et définit les paramètres qui paraissent les plus adaptés. L'établissement des paramètres conduit à la définition des contextes dans lesquels ils opèrent efficacement et à la construction du `contrôleur adaptatif`.

Dans une troisième étape, les contextes doivent être caractérisés en fonction des capteurs ce qui entraîne la définition de la couche d'interprétation de l'architecture.

On peut ensuite s'intéresser à la construction de l'automate, c'est-à-dire à la construction du `contrôleur expert`. L'enchaînement des modes dépend des tâches à effectuer (aller chercher un objet nécessite de se déplacer puis de manipuler un bras et enfin de revenir par exemple), de contraintes mécaniques (s'arrêter avant de pouvoir reculer par exemple et pour s'arrêter il peut être nécessaire de freiner avant) ou encore d'une analyse métier de comment il faut faire. L'élaboration de l'automate amène à définir les différents événements qui vont permettre de changer d'état. Ces événements sont liés d'une part à des changements de contexte et, d'autre part, à la surveillance temps réel de certains capteurs. Il est alors souvent nécessaire de définir de nouvelles `données` et de les relier aux `capteurs`.

5 Une étude de cas : le pilotage automatique des voiliers

Dans ce chapitre, l'architecture IMOCA et la méthodologie proposée précédemment sont mises en oeuvre sur un cas complet dans le but de développer un pilote automatique de voilier. C'est un problème difficile que d'affranchir le marin d'une tâche répétitive et fastidieuse tout en veillant à sa sécurité et en préservant une conduite performante le tout dans un environnement, mer, vent, voilier, par essence très perturbé. Les lois de commande choisies sont de la famille des PID. La validation repose sur l'utilisation d'un environnement virtuel offrant une excellente qualité d'expérience et permettant de comparer différents types de pilotage.

5.1 Position du problème

En course à la voile au large et en solitaire le skipper pour pouvoir manœuvrer, régler, manger ou tout simplement dormir utilise un pilote automatique. Les pilotes automatiques commerciaux utilisés sont de simples régulateurs de cap et d'allure qui n'agissent que sur la barre *via* un unique actionneur. Ils sont peu performants et conduisent parfois à des situations critiques entraînant des dégats irréversibles comme des démâtages ou des chavirages. L'amélioration de ces pilotes est une demande forte des coureurs ainsi que des industriels les concevant.

5.2 Construction du pilote

5.2.1 Etape 1 : mise à disposition d'un environnement réel ou simulé, construction de la couche cible et du barreur réactif

Pour la mer et le vent le modèle `IPAS` (Parenthoën, 2004) pour Interactive Phenomenological Animation of the Sea implémenté dans le langage `ARéVi`[4] est utilisé. Un modèle de voilier, basé sur la dynamique des systèmes matériels, a été développé ; sa géométrie est définie *via* un fichier VRML qui permet un rendu de qualité. Par défaut, le bateau est muni d'un pilote de type commercial, c'est-à-dire un régulateur de cap. L'environnement offre la possibilité de cloner le bateau pour effectuer des comparaisons de performance sur la loi de contrôle. On peut voir figure 9 le bateau (en vert) dans son environnement physique avec son clone (en mauve).

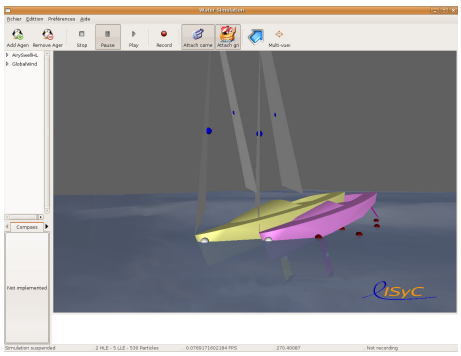


FIG. 9 – *Le bateau (en vert) et son clone (en mauve).*

Le bateau est équipé de tous les capteurs `Sensor` habituels : anémomètre, girouette, speedomètre, GPS, compas, sondeur, angle de barre, plus quelques uns plus spécifiques comme une centrale inertielle ou des jauges de contrainte[5]. Un actionneur `Actuator`, un vérin linéaire agissant sur les safrans à une vitesse constante, est également défini ainsi qu'une interface qui permet au skipper de notifier la consigne de cap `ExternalEvent` à suivre et de la modifier (à l'image des boîtiers de commande disponibles dans le commerce). L'environnement virtuel offre la possibilité à l'utilisateur d'agir à sa guise sur le vent, l'état de la mer et sur les réglages du voilier.

La définition des capteurs, de l'actionneur et des évènements externes, liés aux actions effectuées sur l'interface pilote, de la couche `cible` est ici directe.

5.2.2 Etape 2 : construction de la bibliothèque de lois de commande et de la table des paramètres

Au niveau du contrôle, le seul actionneur à commander est le vérin. Mais, les nombreux capteurs autorisent de multiples régulations. L'expertise a permis d'opter pour de la régulation de type PID car il s'agit d'une solution industrielle éprouvée et ses avantages et inconvénients

[4] ARéVi (Atelier de Réalité Virtuelle) est une bibliothèque de simulation et de rendu 3D.

[5] Les jauges de contraintes servent à mesurer les déformations des matériaux sur lesquels elles sont fixées.

sont bien connus par les marins donc exploitables. Pour autant, il existe de nombreuses implémentations (PPIDD, PDD, ...) de cette loi. Par exemple, le mode *cap* est une régulation de type PDD qui tient compte du roulis à cause de son importance pour l'anticipation :

$\theta_{barre} = K_c(K_{cap}(cap - consCap) + K_{lacet}lacet + am.K_{roulis}roulis)$

θ_{barre} est l'angle de barre absolu, `cap` la `data` donnant le cap magnétique du bateau, `consCap` la consigne de cap à respecter, `lacet` la vitesse de rotation en degrés par seconde du bateau par rapport à un axe vertical (lié à la centrale inertielle) et `roulis` la vitesse de rotation par rapport à l'axe du bateau (dirigé de l'arrière vers l'avant). K_{cap}, K_{lacet}, K_{roulis} et le gain K_c sont ajustés en fonction de l'environnement par le barreur adaptatif. Le terme am devant K_{roulis} vaut ± 1 en fonction de l'amure du bateau. Pour déterminer les bonnes lois et les paramétrer, une table de mixage (cf. figure 10) est créée. C'est l'interface du composant de mise au point. En sélectionnant le bouton correspondant, le mode de fonctionnement courant est choisi (les modes sont ici appelés `Cap`, `Gîte`, `Vitesse`, `Vent`, `Gîte rel` et `Vitesse rel` sur la figure). Et les différents paramètres ou coefficients (indiqués ici avec à leur gauche les modes dans lesquels ils interviennent) sont réglables à l'aide de *scroll bar*. Les 6 modes de barre retenus sont détaillés dans la table 1.

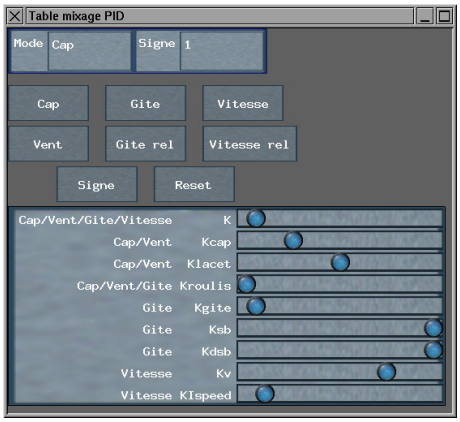


FIG. 10 – *Table de mixage employée pour le réglage des PID.*

Une partie de la table des paramètres choisis est donnée en aperçu dans la table 2.

5.2.3 Etape 3 : caractérisation des données en fonction des capteurs

De nombreuses données sont presque directement définissables à partir des capteurs au formatage près. C'est le cas de la vitesse, de la position, du cap, de la force et de la direction du vent apparent... D'autres données sont plus délicates à obtenir. Par exemple la direction et la force du vent réel sont issues d'un calcul faisant intervenir vitesse, cap du bateau et angle et vitesse du vent apparent. Le résultat peut être affiné en exploitant la centrale inertielle pour tenir compte de l'attitude du bateau.

De plus, un certain nombre de contextes tels que `Largue` et `Mer Calme` sont nécessaires à l'adaptation des modes. Un contexte combine plusieurs informations. Pour `Largue` et `Mer Calme`, la première concerne les *allures* et la deuxième les *états de mer*. Chaque

MODES DE BARRE
Mode cap :
$\theta_{barre} = K_c(K_{cap}(cap - consCap) + K_{lacet}lacet + am.K_{roulis}roulis)$
Mode cap relatif :
$\theta_{barre} = K_c(K_{cap}(cap - consCap) + K_{lacet}lacet + am.K_{roulis}roulis) + \theta_{reel}$
Mode vent réel :
$\theta_{barre} = K_{vr}(K_{TWA}(TWA - consTWA) + K_{lacet}lacet + am.K_{roulis}roulis)$
Mode gîte :
$\theta_{barre} = sgn.K_g(K_{gite}(gite - consGite) + am.K_{roulis}roulis + K_{sb}(sensabarre - consSB) + K_{dsb}(\frac{dsensabarre}{dt}))$
Mode gîte relatif :
$\theta_{barre} = sgn.K_{gr}(K_{gite}(gite - consGite) + am.K_{roulis}roulis + K_{sb}(sensabarre - consSB) + K_{dsb}(\frac{dsensabarre}{dt})) + \theta_{reel}$
Mode gain :
$\theta_{barre} = -K_{gr}(K_{gite}(gite - consGite) + am.K_{roulis}roulis + K_{sb}(sensabarre - consSB) + K_{dsb}(\frac{dsensabarre}{dt})) + \theta_{reel}$

TAB. 1 – *La bibliothèque du barreur réactif.*

contexte correspond au final à une `donnée` de type énuméré. Certaines comme l'*état de mer* sont basées sur une analyse fréquentielle des mouvements du bateau et sur une double intégration des accélérations fournies par la centrale inertielle pour estimer la hauteur des vagues. Ensuite l'échelle de Douglas[6] sert de référence pour qualifier les différents états de mer. D'autres comme les *allures* du bateau sont basées sur l'angle au vent réel moyenné sur une dizaine de secondes.

Les données nécessaires sont au final très nombreuses. Outre les données continues qui donnent des valeurs de type capteurs, celles de type énuméré peuvent se classer en 5 grandes catégories en fonction de ce qu'elles permettent de caractériser : le bateau, la mer, le vent, la route du bateau et les contextes pour les modes de barre. Cette classification est relative au métier et non intégrée à l'architecture. Elle apparait dans les librairies de modèle et de code spécifique.

5.2.4 Etape 4 : construction de l'automate et définition des événements

La dernière étape porte sur la construction de l'automate, donc du `barreur expert`. La figure 11 en donne une version simplifiée sous la forme d'un réseau de Petri interprété. Dans cette vue, les transitions ne sont pas étiquetées.

Cette étape oblige à définir les derniers événements nécessaires et donc quelques nouvelles données. Par exemple le passage entre le mode *cap* et le mode *vent réel* peut se faire sur une notion de stabilité liée à la fois au vent, à la mer, aux mouvements de la barre et à l'allure du bateau. Les consignes se calculent elles directement à partir des données.

[6] L'échelle de Douglas est celle mondialement utilisée dans les bulletins météorologiques.

	Près Mer Calme	Largue Mer Calme	...	Près Mer Agitée	default
Mode cap	$K_c = 1$ $K_{cap} = 5$ $K_{lacet} = 10$ $K_{roulis} = 0$	$K_c = 1$ $K_{cap} = 5$ $K_{lacet} = 15$ $K_{roulis} = 10$	...	$K_c = 2$ $K_{cap} = 5$ $K_{lacet} = 10$ $K_{roulis} = 5$	$K_c = 1.25$ $K_{cap} = 5$ $K_{lacet} = 10$ $K_{roulis} = 0$
Mode vent réel	$K_{vr} = 1$ $K_{TWA} = 5$ $K_{lacet} = 20$ $K_{roulis} = 0$	$K_{vr} = 1.2$ $K_{TWA} = 5$ $K_{lacet} = 20$ $K_{roulis} = 20$	...	$K_{vr} = 2$ $K_{TWA} = 5$ $K_{lacet} = 10$ $K_{roulis} = 0$	$K_{vr} = 1.25$ $K_{TWA} = 5$ $K_{lacet} = 20$ $K_{roulis} = 0$

TAB. 2 – *Extrait de la table des paramètres.*

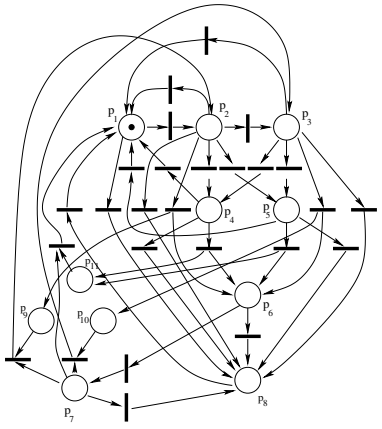


FIG. 11 – *Automate contrôlant le barreur expert.*

Cette mise au point incrémentale justifie pleinement l'utilisation de l'architecture, de la librairie métier et de l'environnement de mise au point. Le développement devient alors un assemblage formaté et adaptable de composants logiciels métier.

5.3 Expérimentation

L'environnement virtuel permet de tester le pilote dans des conditions de mer et de vent variées. Différents éléments du bateau sont réglables comme les voiles, la quille ou la position du skipper. Cependant, tester la qualité d'un pilotage est difficile car il s'agit toujours d'un compromis entre plusieurs critères comme performance, sécurité, consommation énergétique et trajectoire. Aussi le bateau est cloné pour pouvoir comparer le pilote construit selon l'architecture IMOCA, appelé *barreur virtuel*, avec un simple régulateur de cap proche d'un pilote commercial, appelé *PID*. Un certain nombre de cas (rafale, départ en surf, clapot ...) connus pour prendre en défaut ces pilotes commerciaux servent pour la comparaison. La figure 12 présente la trajectoire du *PID* et du *barreur virtuel*. Les deux voiliers naviguent travers au vent

dans 20 noeuds de vent par mer calme lorsque survient une rafale à 23 noeuds qui dure une quarantaine de secondes avant que le vent ne reprenne sa force initiale. Au départ (à gauche sur la figure) les deux voiliers sont quasiment superposés. Quand la rafale arrive ils abattent légèrement pour essayer de contrecarrer le surcroît de puissance mais ils finissent par partir au lof, c'est-à-dire par se tourner brutalement vers le vent. Le *barreur virtuel* passe en mode gîte et accompagne cette aulofée alors que le *PID* cherche à lutter contre l'écart à la consigne en donnant beaucoup de barre (voir la figure 13). Lorsque la rafale se termine le *barreur virtuel* repasse en mode cap et le clone finit par venir s'aligner à l'arrière de l'autre voilier.

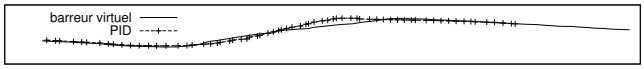


FIG. 12 – *Trajectoires des deux voiliers lors d'une rafale.*

Les courbes de la figure 13 permettent d'expliquer ce qui se passe en observant les angles de barre et les vitesses des deux bateaux.

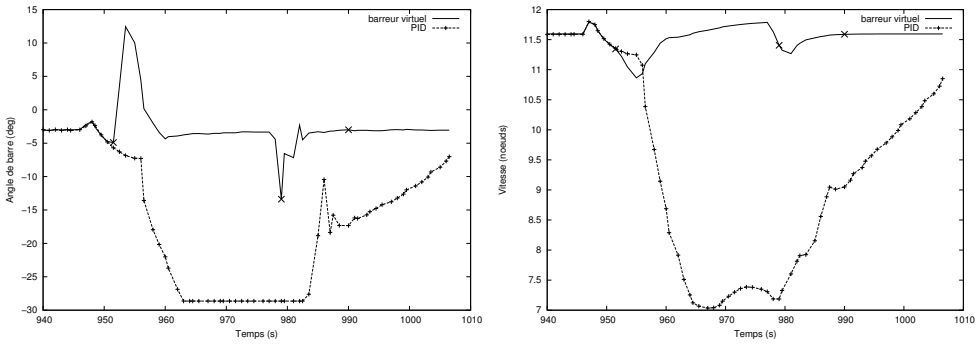


FIG. 13 – *Angles de barre et vitesses.*

Alors que le *barreur virtuel* change de mode (les changements de mode apparaissent avec des croix en X) et parvient ainsi à conserver une bonne vitesse, le *PID* cherche à compenser la déviation de trajectoire en donnant beaucoup de barre ce qui freine le bateau.

6 Conclusion

Ce document présente l'architecture IMOCA et une méthodologie dédiées aux systèmes de contrôle de processus par régulation en environnement perturbé. Le principe est d'effectuer de la commutation de modes en fonction des évolutions de l'environnement ; les modes, leurs paramètres, leur enchaînement étant déterminés par expertise du domaine d'application.

IMOCA offre des modèles et des outils pour la mise au point incrémentale des données et des paramètres du système de contrôle. Cette architecture et cette méthodologie ont été appliquées au cas du développement d'un pilote automatique pour voiliers et donnent un résultat opérationnel apportant un gain à la fois en performance mais également en sécurité.

Dans les perspectives, nous visons la mise en place d'un code opérationnel embarquable, adaptable et reconfigurable sur voilier réel. Nous pensons aussi augmenter les librairies disponibles pour viser d'autres domaines d'application.

Références

Agrawal, M., S. Cooper, L.Graba, et V. Thomas (2004). An open software architecture for high-integrity and high-availability avionics. In *Proceedings of DASC04, the 23rd Digital Avionics Systems Conference*.

Almeida, J., R. Dijkman, M. Sinderen, et L. Pires (2005). Platform-independent modelling in mda : Supporting abstract platforms. In U. Aßmann, M. Aksit, et A. Rensink (Eds.), *Model Driven Architecture*, Volume 3599 of *Lecture Notes in Computer Science*, pp. 174–188. Springer Berlin Heidelberg.

Bertrand, D., A. Déplanche, S. Faucou, et O. Roux (2008). A study of the AADL mode change protocol. In *Proceedings of the IEEE International Conference on Engineering Complex Computer Systems - ICECCS 2008*, Belfast, Ireland, pp. 288–293. IEEE Computer Society.

Bézivin, J. (2006). Model driven engineering : an emerging technical space. pp. 36–64.

Borde, E., P. H. Feiler, G. Haïk, et L. Pautet (2009). Model driven code generation for critical and adaptative embedded systems. *SIGBED Review 6*(3), 10.

Brun, Y., G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, et M. Shaw (2009). Software Engineering for Self-Adaptive Systems. Chapter Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70.

David, R. et H. Alla (1992). *Du Grafcet aux Réseaux de Petri*. Hermès.

Deantoni, J. et J. Babau (2005). A MDA-based approach for real time embedded systems simulation. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, Montreal, pp. 257–264. IEEE Computer Society.

Deantoni, J. et J. Babau (2006). SAIA : safe deployment of sensors based real time application. In *Workshop on Models and Analysis for Automotive Systems (held in conjunction with RTSS)*, Rio de Janeiro, Brésil.

Feiler, P., B. Lewis, S. Vestal, et E. Colbert (2004). An overview of the SAE Architecture & Design Language (AADL) Standard : A Basis for Model Based Architecture-Driven Embedded Systems Engineering. In *Architecture Description Language, workshop at IFIP World Computer Congress*.

Garlan, D. (2000). Software architecture : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA, pp. 91–101. ACM.

Guillou, G. (2010). *Architecture multi-agents pour le pilotage automatique des voiliers de compétition et Extensions algébriques des réseaux de Petri*. Ph. D. thesis, Université de Bretagne Occidentale.

Holloway, L., B. Krogh, et A. Giua (1997). A Survey of Petri Net Methods for Controlled Discrete Event Systems. *Discrete Event Dynamic Systems 7*(2), 151–190.

Horn, P. (2001). Autonomic Computing : IBM's Perspective on the State of Information Technology.

Li, Y., K. H. Ang, et G. C. Y. Chong (2006). PID control system analysis and design. *IEEE Control Systems Magazine 26*(1), 32–41.

Maker, F., R. Amirtharajah, et V. Akella (2013). Meloades : Methodology for long-term online adaptation of embedded software for heterogeneous devices. *Journal of Systems Architecture - Embedded Systems Design 59*(8), 643–655.

OMG (2003). Model driven architecture guide v1.0.1.

Parenthoën, M. (2004). *Animation phénomènologique de la mer : une approche énactive*. Ph. D. thesis, Université de Bretagne Occidentale.

Terrier, F. et S. Gérard (2006). Mde benefits for distributed, real time and embedded systems. In *DIPES, IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems*, Volume 225 of *IFIP*, pp. 15–24. Springer.