Génération de code multi-plates-formes pour la mise au point de modèles IMOCA

Goulven Guillou, Jean-Philippe Babau

Génération de code multi-plates-formes pour la mise au point de modèles IMOCA

G. Guillou
Lab-STICC/UMR 6285, UBO, UEB
20 Avenue Le Gorgeu
29200 Brest, France
goulven.guillou@univ-brest.fr

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

Résumé
Les systèmes de contrôle de processus embarquées en environnement perturbé sont généralement développés au cas par cas pour des plates-formes de déploiement spécifiques, avec un comportement dépendant étroitement des caractéristiques de l'environnement. Le code obtenu est non portable et non reconfigurable. Pour aider à maîtriser le développement logiciel de telles applications, IMOCA offre des outils de modélisation au niveau architectural. Le générateur de code associé permet de produire du code adaptatif et reconfigurable pour un simulateur ainsi que du code embarqué pour différentes plates-formes. L'approche est appliquée à une brique NXT Lego ainsi qu'à un mini char à voile équipé d'une carte Arduino.

Mots Clef
Architecture logicielle, contrôle, génération de code

1. INTRODUCTION
Les systèmes embarqués agissant en environnement fortement perturbé doivent être capables de prendre en compte diverses situations en prévoyant un certain nombre de stratégies.

Un point clé du développement est la configuration des nombreux paramètres du logiciel permettant d'assurer la sécurité du système contrôlé. Ces paramètres interviennent pour à la fois caractériser les changements de contexte, c'est-à-dire les évolutions de l'environnement, et régler les différentes lois de commande qui contrôlent les actionneurs. Leur mise au point nécessite de l'expérimentation qui s'effectue soit de manière réelle soit, pour des questions pratiques et de coût, de manière simulée. Le système doit également être doté de capacités d'adaptation et/ou d'apprentissage pour régler certains paramètres. Il en résulte un besoin d'outils de mise au point de ces paramètres à la fois hors ligne et en ligne.

Dans l'industrie, le développement et la mise au point de ce type de logiciel restent centrés sur le code. Le code embarqué obtenu est dédié à une plate-forme donnée, pour une application spécifique. Il est développé au cas par cas, il n'est ni portable, ni modifiable et donc non réutilisable.

Afin de maîtriser le développement logiciel de telles applications, nous proposons d'utiliser une approche à base de modèles d'architecture logicielle pour des logiciels embarqués adaptatifs, soit l'approche IMOCA [6]. Ce type d'approche à composants facilite la reconfiguration hors ligne de l'architecture d'une application donnée (ajout ou suppression d'un capteur ou d'un actionneur par exemple). Une fois les modèles d'architecture établis, un point clé est la maîtrise de la génération de code. Au delà de la qualité recherchée pour le code, on souhaite ici faciliter la mise en place de modèles et d'outils pour la mise au point du logiciel. Si le code déployé demeure spécifique, on souhaite proposer des modèles de génération de code génériques pour aider à :
- maîtriser la qualité et l'efficacité du code obtenu ;
- la réutilisation de code métier existant (lois de commande, protocoles de communication ...) ;
- cibler plusieurs plates-formes d'exécution : le modèle d'architecture est indépendant d'une plate-forme, le générateur de code doit l'être pour les parties concernées ;
- générer des outils de mise au point : de nombreux paramètres doivent pouvoir être reconfigurés selon les performances obtenues lors des essais ;

Dans ce papier qui concerne des travaux en cours, nous nous basons sur les travaux présentés dans [10] pour la mise en place d'un générateur de code à partir du modèle d'architecture IMOCA, générateur qui intègre les aspects de mise au point et de prise en compte de plates-formes diverses (dont des plates-formes de mise au point). Les aspects temps réel ne sont pas présentés ici, on peut en trouver une approche dans [8].

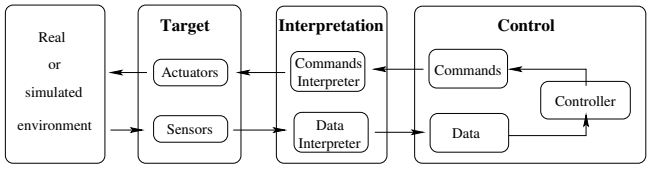
La suite s'organise comme suit. Nous commençons par présenter l'architecture IMOCA dédiée aux systèmes de processus de contrôle en environnement perturbé. Les principes du générateur de code sont ensuite exposés avant de l'appliquer sur deux exemples utilisant des plates-formes d'exécution différentes.

2. ETAT DE L'ART
Les approches à composants [11] pour la conception des systèmes embarqués sont relativement classiques et permettent de maîtriser la complexité du logiciel [4] ainsi que la séparation des préoccupations [1]. Plus particulièrement, la

capacité de faire évoluer le logiciel afin de faire face à la maintenance de la plate-forme, à l'adaptation des comportements du système ou encore à la mise au point de paramètres [3] peuvent être traités comme de la flexibilité logicielle [7]. En général les efforts visant à augmenter la flexibilité conduisent à des conflits vis-à-vis des ressources matérielles de la plate-forme d'exécution [9]. Dans [2] l'évolution est prise en compte en associant à chaque composant une politique d'adaptation mais cette solution suppose que les évolutions soient prédictibles. [10] essaye de concilier l'évolution du logiciel lors du *run-time* avec les contraintes strictes en ressources des plates-formes embarquées. En se basant sur ces travaux, l'idée que nous suivons dans cet article est de se concentrer sur la conception de modèles *métiers* de composants (on se focalise sur un domaine et on ne modélise que les caractéristiques spécifiques des composants), puis de générer du code optimisé qui embarque les éléments nécessaires à la reconfiguration [10].

3. L'ARCHITECTURE IMOCA

IMOCA pour archItecture for MOde Control Adaptation est une architecture dédiée au développement de systèmes de contrôle de processus en environnement perturbé. Trois couches dites `Target`, `Interpretation` et `Control` (voir figure 1) la composent. `Target` avec ses `Actuators` et ses `Sensors` est un modèle de la plate-forme matérielle. `Control` récupère des `Data` (une vue idéale de l'environnement) pour évaluer des `Commands` qu'il transmet aux `Actuators`. La couche d'`Interpretation` réalise l'adaptation entre la couche `Target` et la couche `Control` en faisant le lien entre les `Sensors` et les `Actuators` d'une part, et les `Data` et les `Commands` d'autre part. De cette manière `Control` et `Target` sont indépendants comme dans SAIA [5] ; ce qui permet le développement indépendamment d'une technologie de capteurs et d'actionneurs. Cette indépendance est essentielle dans le domaine de l'embarqué où les plates-formes matérielles peuvent être variées et subir des évolutions (remplacement ou adjonction de capteurs par exemple) au cours du temps.



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

Figure 1: Principes de l'approche IMOCA

Pour sa part, l'application de contrôle est constituée de trois composants fonctionnant en parallèle et de manière asynchrone. Le `ReactiveController` utilise des `Data` directement issues des `Sensors` pour calculer une `Command` qui s'évalue *via* une loi de commande imposée par l'`ExpertController`. Ce dernier est basé sur un automate d'état gérant des modes de fonctionnement. Un changement d'état est lié à un changement d'état de l'environnement (une condition sur les `Data`). Et à chaque état est associé un `Mode` lui-même lié à une loi de commande. Enfin, un `adaptativeController` ajuste les différents paramètres de la loi de commande en ce référant à une look-up table dans laquelle figurent l'ensemble des `Configuration` possibles. La décomposition du contrôleur en trois parties permet de répondre aux trois exigences suivantes : contrôler en permanence le processus avec le `ReactiveController` en appliquant une loi de commande, savoir utiliser une bonne loi et savoir les enchaîner avec l'`ExpertController`, et enfin, ajuster les lois en fonction du contexte pour conserver une bonne qualité de contrôle avec l'`AdaptativeController`.

4. GÉNÉRATION DE CODE

4.1 Introduction

La mise au point d'une application construite selon l'architecture IMOCA nécessite la caractérisation des divers composants, et en particulier des `Data` qui fournissent les éléments de contexte décrivant l'évolution de l'environnement physique du processus. Il est également nécessaire de trouver les lois de commande adaptées en déterminant leurs paramètres en fonction du contexte. Enfin il faut savoir les enchaîner, c'est-à-dire construire l'automate qui définit le comportement de l'`ExpertController`. Pour facilité cette tâche, il est intéressant de générer des outils de mise au point tels que des IHM permettant le réglage des paramètres, le choix de valeurs pour les divers paramètres ou encore des outils de simulation pour évaluer l'effet des actions autorisées sur le comportement du système.

Dans ce papier nous visons à la fois la génération d'un simulateur pour la mise au point ainsi que la génération de code embarqué adaptatif et reconfigurable, le tout, à partir d'un même modèle. Par "code adaptatif" nous voulons dire que l'application est capable de s'adapter aux évolutions de son environnement alors que par "code reconfigurable" nous entendons code dont certains paramètres peuvent être modifiés en ligne sans avoir à recompiler le code, l'architecture de l'application, elle, ne pouvant être modifiée en ligne mais seulement hors ligne. Le générateur de code s'appuie sur la technologie Acceleo qui permet simplement de définir des principes de génération de code en parcourant un méta-modèle Ecore.

4.2 Prise en compte du comportement

Pour le développement d'une application de contrôle, le concepteur s'appuie d'abord sur le méta-modèle IMOCA pour instancier son propre modèle. C'est une opération de configuration de briques de bases fournies par l'éditeur IMOCA (un éditeur graphique IMOCA, basé sur l'outil Sirius d'Obeo, est actuellement en cours de développement).

Le modèle d'architecture IMOCA reste un modèle déclaratif. Il permet de se concentrer sur certaines caractéristiques spécifiques de l'application visée (les périodes d'activation, le nombre de modes de fonctionnement, ...). Par contre, IMOCA n'étant pas un langage de programmation, le comportement attendu n'est pas décrit explicitement. Il faut donc préciser le comportement lors de la génération du code.

A cet effet, on distingue deux parties. La première partie correspond à la sémantique opérationnelle d'IMOCA selon la description donnée dans le chapitre précédent (par manque de place, elle n'est pas formellement décrite ici). Cette partie est prise en compte directement par le générateur de code.

La deuxième partie est spécifique aux applications et est encapsulée dans des bibliothèques métier. Le générateur de code permet de faire le lien entre les deux parties via un outil simple d'intégration de bibliothèques, en produisant un code glue.

Selon ces principes, pour les `Sensors`, les `Interpreters` et le `ReactiveController` le code glue est généré. Par contre, pour l'`ExpertController` et l'`AdaptativeController` des automates sont générés ainsi qu'un contrôleur global et le code pour les `Data`. Pour compléter le code, l'utilisateur fournit des bibliothèques, forcément spécifiques, pour l'acquisition des capteurs, la gestion des actionneurs, les filtres et adaptateurs (`Interpreter`) et les lois de commande.

4.3 Prise en compte des plates-formes

Pour pouvoir adresser différentes plates-formes d'exécution, il est nécessaire de générer du code spécifique pour chacune d'elle. Pour autant, une partie non négligeable du générateur de code doit rester générique et indépendante du langage ciblée. Par exemple, le comportement du contrôleur expert est indépendant du langage ciblé. Pour répondre à ce besoin, le générateur de code s'appuie sur deux couches logicielles. la première se préoccupe de générer le comportement spécifique à IMOCA. Elle peut générer du code impératif ou du code objet (un paramètre de la génération de code). La deuxième couche s'appuie sur la première et est spécifique au langage visé (C ou Java par exemple). Elle est en charge de la génération des fichiers en respectant les spécificités des langages pour la déclaration des fichiers, des classes ou des fonctions. Pour le moment, les spécificités de l'exécutif utilisé sont intégrées dans la deuxième couche via la définition et l'appel des tâches.

De même, afin de simplifier le travail, dans un premier temps, le générateur de code générique fait l'hypothèse que le langage cible s'appuie sur les constructions basiques du langage C (affectation, structures de contrôle). Les pointeurs ne sont pas utilisés dans cette couche. Ainsi, on peut s'appuyer sur cette couche pour générer du code C, du code Java ou tout idiome de ces langages.

4.4 Structure du générateur de code

Le générateur de code est modulaire avec un ensemble de modules spécifiques pour chaque composant IMOCA. Par exemple, il existe un module de génération de code pour l'`ExpertController` et un pour les `Sensors`. Ce module s'appuie sur les deux couches présentées ci-avant. Des modules utilitaires permettent de compléter le générateur de code afin de factoriser et simplifier l'écriture de certaines opérations. Pour la première couche (comportement), les composants simples (`ReactiveController`, `Sensor`, `Actuator` et `Interpreter`) sont construits selon un même principe. Une classe (ou une structure en code impératif) est générée qui gère des objets qui contiennent une donnée, une fonction d'initialisation, une fonction de configuration, des fonctions d'accès et de modification et une fonction spécifique d'exécution (un "run" pour chaque filtre et pour chaque loi de commande du contrôleur réactif). Les autres composants de contrôle sont générés simplement au travers d'une fonction spécifique implémentant les automates. Ces composants sont complétés par une couche service (*pattern facade*), un programme principal (le *main*) et des outils de communication (cf. ci-après).

4.5 Intégration de code métier

Le code obtenu de manière automatique est un code qui est complété en ajoutant du code métier. Typiquement, le concepteur doit préciser les protocoles de communication utilisés par les différents capteurs et actionneurs embarqués, choisir les filtres pour construire les données et agir sur les actionneurs, et enfin décrire les lois de commande prévues. Puisque de nombreuses bibliothèques dédiées existent, l'idée est ici de générer du code de telle manière qu'il autorise leur intégration sans difficulté. Comme dans une approche à composant [1], IMOCA produit un ensemble de signatures de fonctions pour tous les composants concernés (capteurs, actionneurs, filtres et contrôleurs réactifs). Le concepteur n'a alors qu'à fournir l'ensemble des implémentations (code utilisateur) correspondantes, en respectant le typage statique. De plus, nous utilisons les propriétés d'Acceleo pour réserver, dans le code généré, des portions de code personnalisables.

4.6 Tests et mise au point

Pour aider à la mise au point et au test du code, le code généré est modulaire. L'idée est de considérer indépendamment chaque composant d'IMOCA. Si un langage objet est ciblé, une classe est générée pour chaque élément (chaque capteur, chaque donnée, chaque contrôleur). Ainsi, des tests unitaires peuvent être mis en place pour chaque classe générée. On agit de la même façon pour un langage impératif, avec un fichier généré pour chaque composant.

De plus, pour aider à la mise au point des lois de commande, un simulateur Java est généré. La génération du simulateur est basée sur les données de haut-niveau (les capteurs et actionneurs ne sont pas ici considérés) et reprend l'ensemble des contrôleurs. Le concepteur peut ainsi tester différentes lois de commandes. Pour visualiser l'effet de ses choix, il doit bien sûr implémenter un simulateur du système à contrôler (comme dans [5]). Ce simulateur est à relier aux commandes envoyées par le simulateur généré.

4.7 Reconfiguration

Un objectif important recherché ici est la capacité à générer du code reconfigurable dans le sens où certaines valeurs de paramètres doivent pouvoir être modifiées en ligne. Les paramètres associés à des reconfigurations peuvent être liés soit à des `Data` décrivant des changements de contexte, soit à des coefficients intervenant dans les lois de commande (typiquement les valeurs de P, I et D dans une régulation PID). Peuvent également être sujets à reconfiguration les caractéristiques des filtres entre les `Sensor` et les `Data` (typiquement la largeur temporelle d'une fenêtre glissante). Pour préparer la génération de code, chaque paramètre peut être noté comme reconfigurable (la propriété `IsControllable`, par défaut à `false`, est mise à `true`). Si il existe au moins un paramètre reconfigurable, on génère une interface cliente (pour le moment en Java) pour permettre une modification des valeurs des paramètres reconfigurables. Coté embarqué, une tâche serveur se charge de récupérer les modifications et de les répercuter à l'exécution sur les paramètres (appel protégé des *setter* correspondants). Cet outil est particulièrement pratique lors de la mise au point du logiciel. Le

comportement du système peut alors être testé sans avoir à arrêter et à recompiler l'ensemble de l'application [10].

5. EXEMPLES

Nous allons maintenant présenter quelques exemples de code généré pour deux applications déployées sur des plates-formes différentes. La première est une brique NXT Lego équipée d'un moteur et d'un capteur de contact. L'application consistant à contrôler la vitesse du moteur en utilisant le capteur de contact. La deuxième est une carte Arduino Mega équipée d'une centrale inertielle et montée sur un mini char à voile. L'application de contrôle consiste à conserver un cap tout en évitant que le char ne se retourne sous l'action du vent.

La figure 2 présente un extrait de code Acceleo concernant la première couche du générateur et portant sur les `Data`. Au niveau de la deuxième couche, le première partie du code de la figure 3 permet de générer le fichier `Input.nxc` alors que la figure 4 présente son équivalent pour le simulateur (codes expurgés de quelques lignes de commentaires).

```
[template public
 generateWriteDeclaration(data : Data, lang : Integer)]
void Write[name.toUpperFirst()/]
     ([if(data.oclIsKindOf(EnumeratedData))]int
     [else]
[syntaxType(data.oclAsType(ContinuousData).type,lang)/]
     [/if] value) ;
[/template]
```

Figure 2: Fragment du module Acceleo generateDataUtils.mtl

Le code généré pour la plate-forme NXT est du NXC (Not eXactly C) un langage proche du C comme son nom l'indique avec quelques éléments spécifiques. Les fichiers générés sont d'extension `nxc` (ici c'est `Input.nxc` et `Output.nxc`) et l'on retrouve les invocations au préprocesseur comme en C. Les `include` portent sur des fichiers d'implémentation, il n'y a pas de fichier d'interface d'extension `h` comme en C. Chaque donnée nécessite une déclaration, la définition de ses attributs (valeur, fréquence, format ...) et de ses méthodes d'accès.

Le code Acceleo équivalent pour générer le simulateur (figure 4) produit du Java. Une `Data` y est vue comme une donnée de haut niveau, c'est-à-dire ici comme une classe, mais les appels comme `[generateWriteDefinition(data,0)/]` sont identiques à ceux utilisés pour générer du code pour le NXT.

Dans la première couche, les modules Acceleo sont paramétrés par le type de langage utilisé (`0` pour de l'objet, et `1` pour de l'impératif). La figure 5 illustre ce cas.

Ce qui donne pour le cas du robot NXT (les commentaires ont été retirés) :

```
void InitControls() {
InitGo();
Init_Stop();
}
```

```
[template public
             generateInputOutputNXC(aSystem : System)]

[file ('Input.nxc', false, 'UTF-8')]

#ifndef INPUT
#define INPUT

/* file for inputs */

[for(data : Data | aSystem.data->
                    select(oclIsTypeOf(EnumeratedInput)
                     or oclIsTypeOf(ContinuousInput)))]
[generateDataAttributeDeclaration(data,0)/]
[/for]

[for(data : Data | aSystem.data->
                    select(oclIsTypeOf(EnumeratedInput)
                     or oclIsTypeOf(ContinuousInput)))]
[generateWriteDefinition(data,0)/]
[generateReadDefinition(data,0)/]
[/for]

#endif

[/file]

[file ('Output.nxc', false, 'UTF-8')]

#ifndef OUTPUT
#define OUTPUT

/* file for outputs */

#include "OutputToActuator.nxc"
```

Figure 3: Fragment du module Acceleo generateInputOutputNXC.mtl

```
[template public
            generateInputOutputJava(aSystem : System)]
[for(data : Data | aSystem.data->
                    select(oclIsTypeOf(EnumeratedInput)
                     or oclIsTypeOf(ContinuousInput)))]
[file (name.toUpperFirst().
                      concat('.java'), false, 'UTF-8')]

/**
 *  Class for [name/] data
 */

package gener[aSystem.name/];

public class [name.toUpperFirst()/] {

protected [generateDataAttributeDeclaration(data,0)/]

public [generateWriteDefinition(data,0)/]

public [generateReadDefinition(data,0)/]
}
[/file]
[/for]
```

Figure 4: Fragment du module generateInputOutputJava.mtl

```
[template public generateInitControlsDefinition
                    (aSystem : System, lang : Integer)]
void InitControls() {
// [protected ('for Init Adapters Definition')]
// user code
// [/protected]
[for(control : Control | aSystem.modes.command)]
[if (lang=0)]myReactiveController.[/if]
                                  Init[control.name/]();
[/for]
}
[/template]
```

Figure 5: Template Acceleo permettant de générer du code objet et impératif

alors que le code Java généré est :

```
public void InitControls() {
myReactiveController.InitGo();
myReactiveController.Init_Stop();
}
```

Pour l'intégration de code spécifique, l'utilisateur dispose d'un emplacement indiqué par des commentaires (cf ci-après). Par défaut, un code est proposé qui peut être modifié et complété. Il est à noter que les générations de code ultérieures tiennent compte de ces modifications (utilisation des capacités offertes par Acceleo). Ce code par défaut fait appel à une fonction utilisateur pour chaque composant dit *métier*, fonction spécifique qui reste à implémenter.

```
bool UpdateBoolSensorTouch(){
  bool aBoolSensorTouch;
  // Start of user code for ReadBoolSensorTouch
                                          definition
  aBoolSensorTouch=UserReadBoolSensorTouch(touchPort);
  // End of user code
  boolSensorTouch = aBoolSensorTouch;
  return aBoolSensorTouch;
}
```

Si l'utilisateur conserve l'appel de fonction spécifique, il doit fournir dans la bibliothèque de fonctions métier, une fonction qui respecte la signature imposée par le générateur de code.

```
bool UserReadBoolSensorTouch(int port) {
      return ((Sensor(port)==0)) ;
   }
```

Le générateur de code a également été expérimenté pour générer le code de contrôle d'un char à voile sur une plate-forme Arduino. L'exemple suivant donne le code du contrôleur expert qui implémente un automate à états finis. Chaque changement d'état est lié à l'évaluation d'une donnée `theta` représentant l'angle de gîte en degré de la plate-forme. Pour information, on a ici 3 états (states 1, 2 et 3) correspondants respectivement à un état normal, une gîte (excessive) sur tribord et une gîte sur babord.

```
#ifndef EXPERT
#define EXPERT

/**
 * file expert controller
*/

int state = 1;

int getState(int theta) {

switch(state)
{
case 1 :
if (theta > 20) { state = 2; }
if (theta < -20 ) { state = 3; }
break;
case 2 :
if (theta < -20 ) { state = 3; }
if (theta >= -20.0 && theta <=20) { state = 1; }
break;
case 3 :
if (theta > 20) { state = 2; }
if (theta >= -20.0 && theta <=20) { state = 1; }
break;
}
return state;
}

#endif
```

6. CONCLUSION ET PERSPECTIVES

Ce document présente une approche pour générer du code embarqué adaptatif et reconfigurable à partir de l'architecture IMOCA. Le code est destiné à des systèmes de contrôle de processus en environnement perturbé et peut être généré pour plusieurs plates-formes. Un simulateur Java est généré pour l'aide à la mise au point des lois de commande. Les cas d'un robot NXT ainsi que d'un mini-char à voile équipé d'une carte Arduino ont servi pour les expérimentations.

Ce travail est en cours et est actuellement enrichi pour intégrer de nombreux autres aspects de la génération de code. Nous visons d'autres plates-formes et nous cherchons à optimiser le code pour répondre aux strictes limitations en mémoire et en capacité de calcul de certaines d'entre-elles. Nous cherchons également à maîtriser la consommation des systèmes autonomes comme les drones en utilisant des politiques de contrôle qui prennent en compte des critères énergétiques. Enfin, nous cherchons à enrichir et à automatiser l'intégration de bibliothèques métiers existantes pour pouvoir traiter d'autres domaines d'application.

L'objectif de ce travail est de fournir un environnement complet de mise au point, à base de modèles, de générateurs de code pour les architectures IMOCA.

7. REFERENCES

[1] M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polakovic, M. Poulhiès, J. Pulou, S. Seyvoz, J. Tous, and T. Watteyne. Think : View-based support of non-functional properties in embedded

systems. In *ICESS '09 : Proceedings of the 2009 International Conference on Embedded Software and Systems*, pages 147–156, Washington, DC, USA, 2009. IEEE Computer Society.

[2] E. Borde, G. Haik, and L. Pautet. Mode-based reconfiguration of critical software component architectures. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1160–1165, 2009.

[3] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution : Research and Practice*, 13(1) :3–30, 2001.

[4] I. Crnkovic. Component-based software engineering for embedded systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 712–713, New York, NY, USA, 2005. ACM.

[5] J. DeAntoni and J.-P. Babau. 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*, pages 257–264, Montreal, 2005. IEEE Computer Society.

[6] G. Guillou and J. Babau. IMOCA : une architecture à base de modes de fonctionnement pour une application de contrôle dans un environnement incertain. In *7ème Conférence francophone sur les architectures logicielles*, Toulouse France, mai 2013.

[7] J. Mathieu, C. Jouvray, F. Kordon, A. Kung, J. Lalande, F. Loiret, J. Navas, L. Pautet, J. Pulou, A. Radermacher, and L. Seinturier. Flex-eWare : a Flexible MDE-based Solution for Designing and Implementing Embedded Distributed Systems. *Software : Practice and Experience*, 42(12) :1467–1494, 2012.

[8] A. Mehiaoui, E. Wozniak, S. T. Piergiovanni, C. Mraidha, M. D. Natale, H. Zeng, J.-P. Babau, L. Lemarchand, and S. Gérard. A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems. In B. Franke and J. Xue, editors, *LCTES*, pages 121–132. ACM, 2013.

[9] J. Navas, J.-P. Babau, and O. Lobry. Minimal yet effective reconfiguration infrastructures in component-based embedded systems. In *in proceedings of the ESEC/FSE Workshop on Software Integration and Evolution @ Runtime (SINTER'09)*, Aug. 2009.

[10] J. Navas, J.-P. Babau, and J. Pulou. Reconciling run-time evolution and resource-constrained embedded systems through a component-based development framework. *Science of Computer Programming*, 8 :1073–1098, 2013.

[11] C. Szyperski, D. Gruntz, and S. Murer. *Component Software : Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.