IMOCA: a Model-Based Code Generator for the Development of Multi-Platform Marine Embedded Systems

Goulven Guillou, Jean-Philippe Babau

IMOCA: a Model-Based Code Generator for the Development of Multi-Platform Marine Embedded Systems

Goulven Guillou and Jean-Philippe Babau

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

Abstract. Process control systems embedded in disturbed environments are usually developed case by case for specific deployment platforms and their behaviours closely depend on the characteristics of the environment. The obtained code is not portable and not reconfigurable. In order to help the software development of such applications, IMOCA offers architectural modelisation tools. The associated code generator allows to product adaptative and reconfigurable code for a simulator as well as embedded code for various platforms. This approach has been tested on NXT bricks, Arduino boards and Armadeus boards.

Keywords: sofware architecture, control, code generation

1 Introduction

Embedded systems in an unpredictable and disturbed environment, like underwater control systems, have to take into account various situations by considering different strategies. Their development requires large parameters configuration in order to ensure safety and efficiency of the controlled system.

The configurable parameters are used to characterize the context (environment interpretation and its evolution), the execution platform and the control part of the system. The tuning of the parameters is based on simulations for cost reasons and on real testing for safety reason. At the end, the system has to be equipped with adaptation and/or learning abilities to adjust some parameters in-line. Therefore, there is a strong need for online and offline tuning tools. Unfortunately in industry, the development of the software for such systems is mainly focused on the code efficiency. The produced embedded code is dedicated to a given platform for a specific application in a specific context. The obtained code is difficult to maintain and to adapt for new applications and new contexts. Portability and reusability are limited.

To tackle these limitations, we propose in [2] a model-based software architecture integrating high adaptive capabilities, the IMOCA approach. Once the architecture model is established, a key point is then the code generation. In this

paper, we present the implementation of the model-based code generator from the architecture model IMOCA. Even if the deployed code remains specific, we propose generic models of code generation to allow:

- the control of quality and efficiency of the generated code: the code is optimized for static parts and modular for adaptive part;
- the integration, and so the reuse, of existing specific legacy code is facilitated: domain functions such as control law, communication protocol, acquisition policy can be easily integrated;
- the code generation for different target platforms: the architecture model and most of the code generator is independent of a given platform, specific aspects are encapsulated in a platform abstraction view;
- the model and the code generator integrate testing, adaptation and tuning facilities: design facilitates declaration of adaptive parameters and functions, parameters can be tested with a generated simulator, code integrates reconfiguration capabilities for adaptive parts.

The generator incorporates tuning aspects and takes into account various platforms, including testing platforms. The simulator is written in Java, while the generated embedded code is written in C language (or family of C language). To generate optimized embedded code, the code generator is based on the principles presented in [5].The models and tools have been tested on NXT bricks, Arduino boards and Armadeus boards for simple control applications, sand yacht control and autonomous sailing boat VAIMOS control.

After a presentation of the IMOCA architecture model, we present the structure and the underlying principles of the code generator while using it with two applications on two different execution platforms.

2 Related Works

Component-based approaches [11] for the conception of the embedded systems are relatively classical and allow to deal with the sofware complexity [7] and the well-known *separation of concerns* in Model Driven Architecture (MDA) terminology [4]. In particular, software evolution ability in order to take into account either the platform maintenance or system behavior adaptation or parameters tuning [6] can be viewed as software flexibility [9]. Generally, efforts to increase the flexibility lead to conflicts with respect to the material resources of the execution platform[10]. In [3] binding each component to an adaptation policy allows software evolution, but this solution assumes that the evolutions are predictable. [5] tries to conciliate the software evolution at *run-time* with the hard resource constraints of the embedded platforms. Relying on this work, we focus on the conception of *domain-specific* models of components (we focus on a specific domain without modeling specific features of the components), then we generate optimized code which embeds the necessary elements for the reconfiguration [5].

3 The Architecture Model IMOCA

IMOCA for archItecture for MOde Control Adaptation is an architecture model dedicated to the development of process control systems embedded in a disturbed environment. This architecture is composed of three layers called `Target`, `Interpretation` and `Control` (see figure 1). `Target` with its `Actuators` and `Sensors` is a platform-specific model of I/O. `Control` uses `Data` (an ideal view of the environment) to compute ideal `Commands` to act on the environment. The `Interpretation` layer realizes the adaptation between the `Target` layer and the `Control` layer by linking `Sensors` and `Actuators` on the one hand, and `Data` and `Commands` on the other hand. In this way `Control` and `Target` are independent like in SAIA [8] and this allows the development independently of specific sensors and actuators technologies. This independence is important in the context of embedded systems because material platforms may be various and may evolved (change or add a sensor for example).

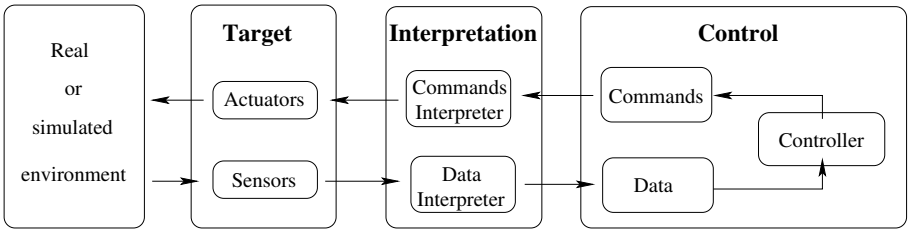

Fig. 1. Principles of IMOCA approach

The `Controller` is composed of three sub-controllers. The `ReactiveController` uses `Data` to compute a `Command` based on a control law. The `ExpertController` is in charge of defining the current control law. It is based on a finite state automaton that manages running modes. Each state is associated with a `Mode` which is itself associated with a control law. A state change is linked to a change of state of the environment (a function of `Data` which returns a boolean). Finally, an `AdaptativeController` adjusts different parameters of the control law with respect to a look-up table in which appear all the possible `Configuration`. Based on this three collaborating layers, the `Controller` allows to answer the three following requirements: controlling the process with the `ReactiveController` by applying an adapted control law thanks to the `ExpertController`, and finally, adjusting the control laws with respect to the context in order to keep a high quality of control with the `AdaptativeController`.

4 Code Generation

The model of the software architecture design is an instance of the meta-model IMOCA, expressed using ecore. The code generator leans on Acceleo technology which allows to define code generation principles through an Ecore meta-model.

4.1 Parameters configuration

A software application based on the architecture model IMOCA integrates variability to adapt the system to a specific context or a specific platform. Variability is first implemented through parameters to characterize, for example, a threshold in a filter adapter, or a coefficient for a control law. Second, the `ExpertController` automaton defines a dynamic behavior through running modes. To tune parameters and modes, we propose to generate simulation tools to evaluate the impact of different parameter values and specific actions on the system.

In this paper, we target the generation of an adaptive and reconfigurable embedded code. "Adaptive code" means the application is able to take into account the evolution of its environment through the `adaptativeController`. For each environment context, specific values of parameters are defined at design time using the simulation. Then, the `adaptativeController` adapts online the values depending on the context. "Reconfigurable code" means that the embedded values of the parameters can be modified online without the need to recompile the code. The reconfiguration capability concerns parameters only, the architecture of the application cannot be modified online.

In this version, a Java simulator is generated. The generation of this simulator is based on high-level data (sensors and actuators are not considered here) and includes all the controllers. The designer can test different control laws and parameter setting through a dedicated generated User Interface. Each `Data` can be controlled and each new `Command` is printed on a control screen. To view the real effect of the controller, is is necessary to implement an environment simulator for the system itself (as in [8]). The latter must be connected to the commands sent by the generated simulator.

4.2 Taking into Account the Behavior

The architecture model IMOCA is a declarative model. It allows to focus on specific features of the target application (activation periods, number of operating modes, ...). However, IMOCA is not a programming language, the expected behavior is not described explicitly. The expected behavior is implemented by the code generator, following the IMOCA semantic.

For this purpose we distinguish two parts. The former corresponds to the operational semantic of IMOCA which has been given in the previous section. The code generator is directly in charge of this part. The behavior is expressed in a simple C code (no pointer) to be reusable in different languages. The latter is specific to application-dependent part and is encapsulated in domain-specific

libraries. The code generator is in charge to produce glue code to link these two parts *via* a simple library integration.

According to these principles, the `ExpertController` and the `adaptiveController` automata are generated together with a general controller and the code dedicated to `Data`. For the `Sensors`, the `Interpreters` and the `ReactiveController`, only the declaration and glue code is generated. To complete the code, the user provides specific libraries to deal with sensors acquisition (filters, ...), actuators management, `Interpreter` and control laws.

For example, the following code defines the `ExpertController` of a mini sand yacht and is, in fact, the implementation of a finite state automaton. Each state change is bound to the evaluation of the data `theta` which represents the heeling of the sand-yacht in degrees. For information, here we have three states (states 1, 2 and 3) respectively corresponding to a normal state, a (excessive) heel to starboard and a heel to port.

```
#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
```

4.3 Domain-Specific Code Integration

As previously said, generated code must be completed by adding domain-specific code. Since many dedicated libraries exist, the idea is to generate code in a way that allows a smooth integration. As in a component-based approach [4], IMOCA produces a set of signatures of functions for all relevant components (sensors, actuators, filters and reactive controller). Thus, the designer has to provide the set of corresponding implementations (user code) while respecting the static typing. In addition, we use the properties of Acceleo to add, in the generated code, portions of customizable code. For this purpose, a special place is pointed by a commentary and is available for the user (see below). By default, a code is provided which can be modified and completed. It should be noted that subsequent generations of code take into account these changes (automatically performed by Acceleo). This default code calls a user function for each domain-specific component, user function that we have to implement.

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

If the user keeps the call of the specific function, he must provide a function that respects the signature required by the code generator to the specific library:

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

4.4 Taking into Account Platforms

In order to address various execution platforms, we need to generate specific code for each of them. However, a significant part of the code generator must remain generic and independent of the target language. For example, the behavior of the `ExpertController` is independant of the target language.

To address this requirement, the code generator is based on three software layers. The first one is concerned with generating the specific behavior related to IMOCA. It can generate imperative code or object-oriented code. We add a parameter to each generative function to define the expected code (imperative or object-oriented). The second layer is based on the first one and is specific to the target language (Java or C for example). It is in charge of generating the files respecting the specific language features for the declaration of files, classes or functions. Currently, the specificities of the executive are included in the second

layer *via* the definition and the call of tasks. The last layer is in charge of general services.

For the first version of the code generator, the target language is based on the basic constructs of C language (assignment, control structures). Pointers are not used. Thus, we can rely on this layer to generate C, C++, Java code or any C-like language.

We have used the code generator for two applications deployed on different platforms. The former is a NXT Lego brick equipped with a motor and a touch sensor. We just control the speed of rotation of the motor by using the touch sensor. The latter is a mini sand yacht with an Arduino board Mega and an inertial motion unit (IMU). We have to try to keep the course and, in the same time, to avoid the capsize of the vehicle due to the wind action.

At the level of second layer, the first part of the code presented figure 2 allows to generate the file `Input.nxc` whereas figure 3 presents the same thing for the simulator (some lines of commentary have been removed).

We generate NXC source code for the NXT platform. NXC means Not eXactly C, a C-like language with some specific features. Generated files have an extension of `nxc` (here the files are `Input.nxc` and `Output.nxc`) and we retrieve the preprocessor invocations like in C. We `include` only implementation files, there is no interface file with an extension of `h`. Each data needs a declaration, the definition of their attributes (value, frequency, format ...) and of their access methods.

The equivalent Acceleo code to generate the simulator (see figure 3) produces Java code. A `Data` is viewed as a high-level data, that is to say here as a class. However calls like `[generateWriteDefinition(data,0)/]` remains identical to those used to generate code for the NXT brick.

In the first layer, Acceleo modules are parameterized by the type of the used language (`0` for object-oriented programming, and `1` for imperative programming). Figure 4 shows this case.

This leads for the case of the NXT brick (the commentaries have been removed) to:

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

whereas the generated Java code is:

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

4.5 Structure of the Code Generator

The code generator is modular with a set of specific modules for each component of IMOCA. For the first and second layers, a code generation module is proposed

```
[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"
```

Fig. 2. Fragment of the generateInputOutputNXC.mtl Acceleo module

```
[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]
```

Fig. 3. Fragment of the generateInputOutputJava.mtl module

```
[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]
```

Fig. 4. Acceleo template for generating object-oriented and imperative code

for each modeled entity (`Sensors`, `ExpertController`, ...). Utilities modules are used to complete the code generator in order to factorize and simplify common pieces of code.

For the first layer (behavior), simple components (`ReactiveController`, `Sensor`, `Actuator` and `Interpreter`) are built according to the same principle. A class (or a structure in the case of imperative code) is generated which manages objects that contain a value, an initialization function, a configuration function, a getter, a setter and specific executive functions (a "run" for each filter and for each control law of reactive controller). The other control components are generated through a specific function which implements automata. Other components implement a service layer, a main program (the *main*) and communication tools.

Figure 5 shows a piece of Acceleo code which stands in the first layer of the generator and concerns the `Data`.

```
[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]
```

Fig. 5. Fragment of the generateDataUtils.mtl Acceleo module

To assist testing, the generated code is also modular. For object-oriented language, a class is generated for each element (each sensor, each data, each controller, ...). For an imperative language, a file is generated for each element. Thus, the structure helps on unit testing activity.

4.6 Reconfiguration

The code generator generates reconfigurable code in the sense that parameter values must be changed online. To prepare the code generation, each parameter can be recorded as reconfigurable (the value of the property `IsControllable` which is `false` by default, is set to `true`). If at least one reconfigurable parameter exists, a client interface (currently in Java) is generated to allow the tuning of the values of reconfigurable parameters. In the embedded code, a server task retrieves the changes and modifies online the corresponding parameters (this modification is done *via* a call to the corresponding setter). This tool is especially useful during prototyping phase. The behavior of the system can be tested without having to stop and recompile the whole application [5].

5 Conclusion and Future Works

This paper presents an approach to generate adaptive and reconfigurable embedded code based on the architecture model IMOCA. The code is designed for process control systems in disturbed environments and can be generated for multiple platforms. A Java simulator is generated to assist the user in the tuning of control laws. The case of an NXT robot and a mini sand-yacht equipped with an Arduino board have been used for experiments.

This work is in progress and we will extend it to include many other aspects of the code generation. We would like addressing other platforms and trying to optimize the code to take into account the limited memory resource and the limited computing power of some platforms. We also seek to control the consumption of autonomous systems, like drones, by using control policies based on energy criteria. Finally, we seek to enrich and automate the integration of existing specific libraries in order to deal with other application areas.

The aim of this work is to provide a comprehensive model-based environment for the development of code generators for architectures IMOCA.

References

1. Mehiaoui A., Wozniak E., Tucci Piergiovanni S., Mraidha C., Di Natale M., Zeng H., Babau J.-P., Lemarchand L., Gérard S.:A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems. LCTES, pp. 121–132. ACM (2013)
2. Guillou G., Babau JP.: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 (2013)
3. Borde E., Haik G., Pautet L.:Mode-based reconfiguration of critical software component architectures. In: Design, Automation Test in Europe Conference Exhibition, 2009. DATE 09., pp 1160-1165. (2009)
4. Anne M., He R., Jarboui T., Lacoste M., Lobry O., Lorant G., Louvel M., Navas J., Olive V., Polakovic J., Poulhi's M., Pulou J., Seyvoz S., Tous J., Watteyne T.: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, pp 147-156. IEEE Computer Society. Washington, DC, USA (2009)
5. Navas J., Babau J.-P., Pulou J.:Reconciling run-time evolution and resource-constrained embedded systems through a component-based development framework. Science of Computer Programming, 8, pp 1073-1098. (2013)
6. Chapin N., Hale J. E., Khan K. M., Ramil J. F., Tan W.-G.:Types of software evolution and software maintenance. Journal of software maintenance and evolution : Research and Practice, 13(1) pp 3-30 (2001)
7. Crnkovic I.:Component-based software engineering for embedded systems. In Proceedings of the 27th International Conference on Software Engineering, ICSE 05, pp 712-713. ACM, New York, NY, USA (2005)
8. DeAntoni J., Babau J.-P.: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, pp 257-264. IEEE Computer Society, Montreal (2005)

9. Mathieu J., Jouvray C., Kordon F., Kung A., Lalande J., Loiret F., Navas J., Pautet L., Pulou J., Radermacher A., Seinturier L.:Flex-eWare : a Flexible MDE-based Solution for Designing and Implementing Embedded Distributed Systems. Software : Practice and Experience, 42(12) pp 1467-1494 (2012)
10. Navas J., Babau J.-P., Lobry O.:Minimal yet effective reconfiguration infrastructures in component-based embedded systems. In proceedings of the ESEC/FSE Workshop on Software Integration and Evolution @ Runtime (SINTER09) (2009)
11. Szyperski C., Gruntz D., Murer S.:Component Software : Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York (2002)