ImocaGen: A Model-based Code Generator for Embedded Systems Tuning

Goulven Guillou and Jean-Philippe Babau
Lab-STICC/UMR 6285, UBO, UEB, 20 Avenue Le Gorgeu, 29200 Brest, France

Keywords: Software Architecture, Code Generation, Control.

Abstract: IMOCA is a model-based architecture model dedicated to embedded process control systems in disturbed environment. These systems depend on various parameters which are difficult to set because they are bound to environment changes. In this paper we propose to extend IMOCA with the meta-model ImocaGen for managing the aspects of the code generation. ImocaGen allows to target multiple platforms and different programming languages, generates both embedded code as well as tuning and reconfiguration tools, takes into account different communication protocols and offers a mechanism for integrating handwritten code. This approach is tested on a basic control application for a NXT brick for which three generations are performed: the first one for a PC with an USB connection, the second one for an Android tablet with a Bluetooth connection and the last one for a simulator in Java.

1 INTRODUCTION

Nowadays, embedded systems are increasingly used in various domains like drones and environment monitoring. They integrate a large set of hardware and software components (sensors, actuators and control policies) and various technologies in various domains (mechanics, electronics, hydraulics ...). Classicaly, the embedded code remains static, optimized for a given platform, a specific application in a specific context. But, an unpredictable environment require large configuration capacity for the system. In uncertain environment, the application domain for this paper, remote monitoring, and sometimes remote control, has to be considered. A good example of such system in a disturbed environment may be an autonomous sailboat equipped with many sensors (anemometer, wind vane, speedometer, inertial motion unit, GPS ...) and actuators (ram, winch ...) on which the rudder and the sails are controlled.

At design time, remote monitoring and tuning is used to evaluate solutions. At run-time, remote monitoring and tuning is useful to ensure safety. At the end, there is a strong need to provide tuning tools because the system depends both intrinsic mechanical features and environment which are difficult to model and forecast. In this context, from high-level model, a code generator has to address different platforms, testing and tuning tools for quick and safe development.

For code generation, the first challenge is to target several platforms, for embedded code and for remote controlling device, several communication technologies (Bluetooth, Wi-Fi, USB ...) and protocols between an embedded system and its controlling device.

The second challenge is to generate tools for monitoring, testing, tuning and reconfiguring system parameters. In this paper a reconfigurable parameter is a parameter for which its embedded value can be modified online without the need to recompile the code.

Since a large amount of code is already available (protocols, control laws, filters, drivers ...) and in order to avoid code rewriting, the last challenge is to facilitate the integration of domain specific code and platform-specific libraries in the generated code.

To face these challenges, in this paper, we propose a model, called ImocaGen, for generating a code from an IMOCA architecture model. If IMOCA is dedicated to process control systems in uncertain environments, ImocaGen, for its part, is in charge of the aspects of the code generation (embedded system, remote control and tuning tools), considering the platforms, the programming languages and the communication technologies. Through a delegation mechanism, ImocaGen also facilitates the integration of legacy code and allows in this manner to deal with various heterogeneous technologies.

The remainder of the paper is organized as follows. Section 2 presents works relative to code gener-

Guillou, G. and Babau, J-P.
ImocaGen: A Model-based Code Generator for Embedded Systems Tuning.
DOI: 10.5220/0005804103900396
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 390-396
ISBN: 978-989-758-168-7
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

ation, simulation and tuning of control systems. Section 3 is dedicated to an overview of IMOCA and ImocaGen models. Section 4 details the code generation principles in order to take into account platforms, programming languages, remote control, tuning tools and code integration. Section 5 shows an illustrative application for which three generations are performed for three different platforms. Finally, section 6 concludes the paper and presents ongoing works.

2 RELATED WORK

Using based components approaches (Szyperski et al., 2002) and MDE methods for mastering the software complexity is quite classical. In the context of embedded systems, the wide variety of available sensors and actuators and the need to change or to upgrade them to cope with the requirements of open-ended environments forces to design multi-levels architectures to abstract the hardware and ensure the separation of concerns. Software requirements are very similar in the world of robotics and tools for defining component-based architectures have been proposed in this context like the middleware ROS (Quigley et al., 2009) or the domain specific modeling language (DSML) RobotML (Dhouib et al., 2012). With these development environments, even if the architecture is free (deliberative, reactive, hybrid ...), the hardware abstraction remains limited, supporting small hardware variability. The advantage of IMOCA is to propose an architecture with a clear separation between the data and the models of sensors/actuators which ensures a high level hardware abstraction like in SAIA (Sensors/Actuators Independent Architecture) (DeAntoni and Babau, 2005).

In IMOCA, a finite state machine serves to describe the behavior of a control component. This choice has been done for practical reasons and not for the use of validation tools. For this perspective, MontiArcAutomaton (Ringert et al., 2013) modeling language or V^3CMM (Alonso et al., 2010) offer this kind of tools which allow to ensure that the implementation refines its specification.

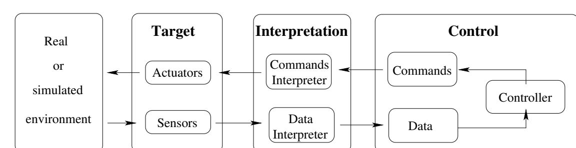
For the code generation, ORCCAD (Simon et al., 1997) is a software environment dedicated to the design and the implementation of advanced robotics control systems but allows only the generation of C++ code. BRICS [1], through model transformation, generates component frameworks usable in particular by ROS but does not generate directly embedded code. For its part, MontiArcAutomaton is multi-platform and allows to generate Java code and Python code whereas RobotML allows both to target a middleware like ROS and to generate a simulator. ImocaGen generates both embedded C-like code for multiple platforms, Java simulation tools and online tuning tools.

The use of simulation in embedded systems has been spread for cost and risk reasons. For dynamical systems, Simulink [2] is very popular and moreover allows to set the simulation for optimization purposes. Robotics dedicated middlewares like ROS, Player (Collett et al., 2005) or MIRO (Utz et al., 2002) integrates numerous libraries (physical engines, 3D libraries) for helping the designer to build simulated environments. For the moment ImocaGen does not integrate this kind of simulators but allows generating basic tools for simulating, monitoring and testing the application and also tuning tools for the setting at run time.

3 MODELS

IMOCA (Guillou and Babau, 2013) is an architecture model for process control systems using sensors, actuators and control laws. IMOCA is especially dedicated to uncertain and unpredictable environments. Aerial drones, terrestrial or underwater vehicles are good examples of these systems. For these applications the behavior of the system depends generally on numerous parameters which need long and tedious settings. IMOCA deals with data acquisition, control and adaptation according to the changes of the environment. IMOCA allows to focus on a number of parameters in order to tune the system. For its part, ImocaGen concentrates on code generation and plateforms integration. ImocaGen allows to generate both the embedded code for the controller and tuning tools. The former is a server and the latter is a client and they communicate each other by using a protocol and a particular technology.

3.1 IMOCA: Model for Control



Frames represent components, arrows describe data flow

Figure 1: Principles of IMOCA approach.

IMOCA is based of three layers called `Target`, `Interpretation` and `Control` (see figure 1).

[1] http://www.best-of-robotics.org/

[2] http://fr.mathworks.com/products/simulink/

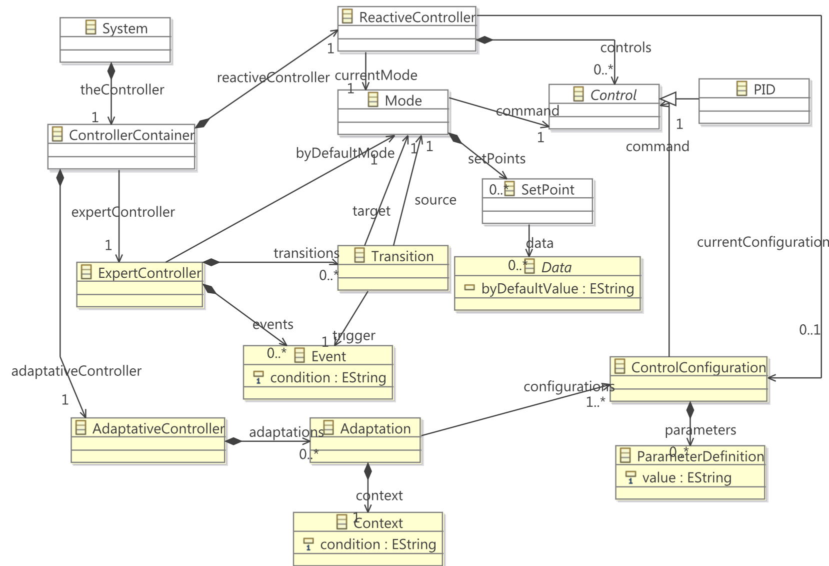


Figure 2: IMOCA view.

`Target` with its `Actuators` and `Sensors` is a platform-specific model of Inputs/Outputs. `Control` uses `Data` (a virtual view of the environment) to compute `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 (DeAntoni and Babau, 2005) and this respects the separation of concerns principle and allows the independent development of specific sensors and actuators technologies. This independence is important in the context of embedded systems due to platforms evolutions.

The `Controller` itself is composed of three sub-controllers. The `ReactiveController` (see figure 2) 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 `Context`. Finally, an `AdaptativeController` adjusts different parameters of the control law (typically the P, I and D of a PID regulator) with respect to a look-up table in which appear all the possible configurations of the control laws. Based on this three collaborating sub-controllers, the `Controller` allows to answer the 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`.

IMOCA is useful for targeting different application domains like the control of underwater and terrestrial vehicles, sailing boats, flying drones or embedded scientific stations with various sensors because the system is always controlled by the `ReactiveController` at a frequency close to that of the sensors, the control law is chosen by the `ExpertController` which reacts to its environment by switching from a law to another more adapted law and the parameters of these laws can be adjusted according to the environment features. These applications may use various platforms like arduino boards, ARMadeus boards, Lego NXT bricks or mini PC with various firmwares and operating systems. If IMOCA serves for modeling of the control architecture of such applications, it is not designed for generating code and for facing the platforms heterogeneity. For this purpose another model, called ImocaGen, is dedicated to consider these aspects.

3.2 ImocaGen: Model for Code Generation

The generated code depends on platforms, operating systems, programming languages and communication technologies. In order to avoid implementation details in IMOCA, the meta-model ImocaGen includes, in addition of IMOCA itself, a model of technical aspects.

In the ImocaGen meta-model (see figure 3) the

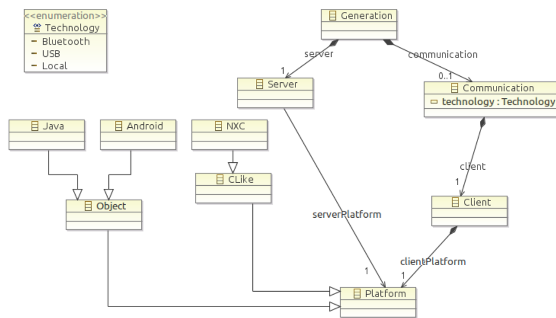
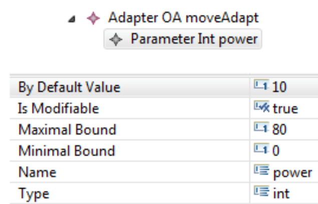


Figure 3: ImocaGen meta-model.



By Default Value	10
Is Modifiable	true
Maximal Bound	80
Minimal Bound	0
Name	power
Type	int

Figure 4: Modifiable parameters.

control system is represented by the server (the embedded system) and the client (the remote controller). If a client exists, a communication protocol is defined with the help of different technologies like Bluetooth or USB. Platforms are mainly taken into account through the supported programming languages. Two main families are defined, C-like languages and object oriented languages, the former contains NXC (for Not eXactly C) a high-level C-like programming language for the Lego Mindstorms NXT and the latter contains Android which is not a programming language. For the generator, the interesting information about Android is the usage of the virtual machine Dalvik which allows to run Java code.

At architectural level, for a parameter, the property `IsModifiable` allows to declare it reconfigurable. If an IMOCA parameter is reconfigurable, ImocaGen defines a set of reconfiguration features, used for code generation, like minimal and maximal bounds (see figure 4). If at least one reconfigurable parameter exists, a client interface is generated to allow the tuning of the values of reconfigurable parameters. In the embedded code, a server task receives the changes and modifies online the corresponding parameters. This interface is especially useful during prototyping stage. Thus, the behavior of the system can be tested without stopping or recompiling the application (Navas et al., 2013).

4 CODE GENERATION

IMOCA and ImocaGen are designed with the help of Ecore the core meta-model of EMF (Eclipse Modeling Framework). It is important to notice that `Imoca.ecore` is a part of `MyImocaGen.ecore`. So `MyAppli.Imoca` is used to initialize `MyAppli.ImocaGen` (see figure 5).

From this model, Acceleo modules (`ImocaGen.mtl`) are used to generate code. Acceleo is a code generator from the Eclipse Foundation based on the model-to-text (M2T) transformation. The generator produces both embedded C-like code and Java code and more generally all the files structure as well as the code which describes the behavior of the application. The code generator is modular, based on a code generation library called `LibImoca.mtl`. The generated code allows to integrate two kinds of legacy code: platform dependant code (`ImocaLib.nxc`) and specific domain code (`UserLib.nxc`).

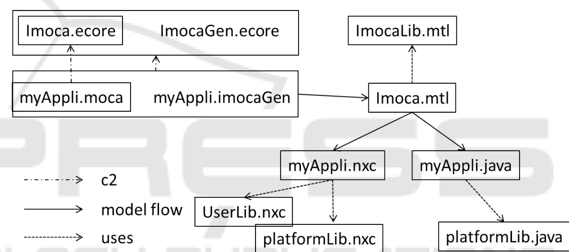


Figure 5: Code generation process.

4.1 Platforms and Languages Integration

In embedded system domain, the C language is used classically for efficiency. Although the targeted platforms support various programming languages thanks to various IDE and compilers, most of them offer the possibility of using a C-like language. However, versions of C are slightly different from one platform to another. Therefore the generator is forced to use a subset of C, the common part of different C variants for targeting different platforms. For the simulator written in Java, the generator exploits the fact that Java syntax and C syntax are close for factorizing as much as possible their common part.

The generator is composed of three software layers. The first layer is platform specific. It is in charge of generating the files. Indeed, the set of files may vary between the chosen programming languages. A C project contains header and implementation files whereas a Java project is organized in packages and

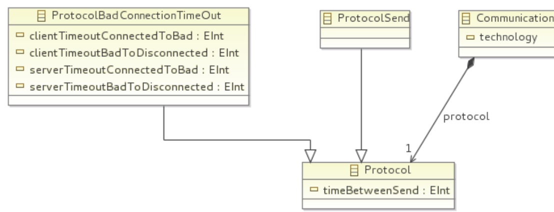


Figure 6: Communication protocol.

classes. For NXC only implementation files are generated. In C, data is a variable whereas, in Java, a data is viewed as an object, an attribute with setters and getters.

The second layer is a generic layer for generating all that is common to C and Java. The generated algorithms (expert and adaptive controllers) are independent of platforms and, therefore, are written in a basic C, without pointers and specific C. This code describes the structure and the behavior of the application.

The last layer is platform specific and is in charge of the syntax of the languages (declaration, function calls, ...). In particular, it tries as much as possible to factorize the common parts of imperative code and object oriented code (through a parameter with specifies the type of the language) by exploiting some similarities between C and Java.

4.2 Remote Control Integration

In the targeted application domain, interaction with the embedded system is done through a wireless communication. The code generation provides a graphical user interface (GUI) and a communication protocol, depending on the `Client` declaration. In the ImocaGen meta-model (see figure 6) a `Communication` contains a `Protocol` which defines all the technical characteristics. The protocol defines how to establish the communication, how to stop it and how to ensure the data correctness. It also defines frames, messages format and parameters about the maximal size of a packet and timeouts (to detect communication loss). The timeout aspect is modeled by the `ProtocolBadConnectionTimeout` class in the ImocaGen metamodel.

The code generation consists on instantiating a generic protocol. The client sends frames (connection and requests) to the server periodically. Then it waits for a response. If the client, resp. the server has nothing to send, resp. to answer, it has to transmit a null message. Then, as presented in figure 7, timeout is used to detect connection loss. The connection state information is available for both client and server sides. The proposed protocol is well suited

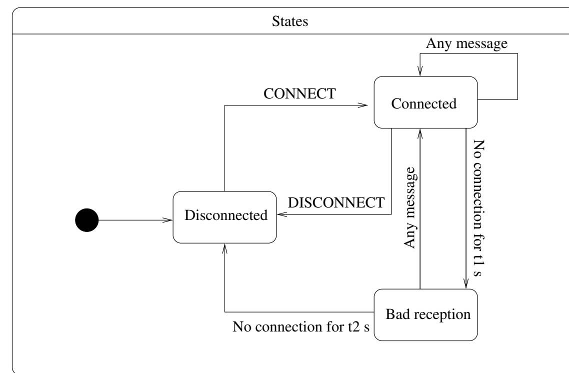


Figure 7: The three states of the protocol BadConnectionTimeout.

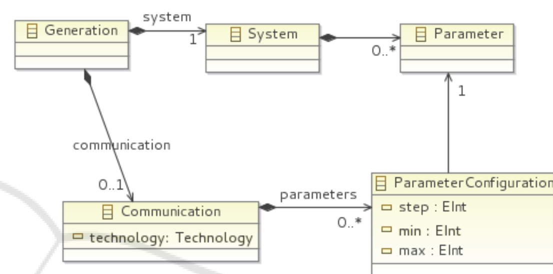


Figure 8: Remote control and parameters reconfiguration.

to a system which evolves in a natural environment where several events may disturbed communications. Indeed, if necessary, ImocaGen may be extended to model other kind of communication policies.

For each reconfigurable parameter, a `ParameterConfiguration` (see figure 8) details all the necessary elements for defining the interaction policy: minimal, maximal and the step between two consecutive values. The generation of a dedicated widget is automatically performed for each reconfigurable parameter. When a parameter modification is performed on remote side, it initiates a client/server communication. Technically, an identifier (ID) is generated for each reconfigurable parameter and allows to do the link between the client and the server.

As for protocol, the remote client code is in fact a parameterization of an existing generic code. For example the `InitServer` primitive has two parameters, the first one defines the reception timeout and the second one the disconnection timeout. The function `CreateMessage` takes an ID as parameter for identifying the request. The creation of a widget is based on a name, an ID, a type (int, float ...), a minimal, a maximal and a step value.

From structural point of view, the generic code is based on provided libraries fo each platform. The library interface is a facade to set the application de-

pendent characteristics, i.e. the name and the number of widgets for the GUI. The implementation of the library is based on two layers. The first layer implements the behavior and is generic. The second layer is platform dependent, encapsulating technologies details. GUI description, send/receive message primitives belong to this layer.

4.3 Domain-specific Code Integration

The generated code is a wrapper for user code. The user code integration finalizes the application. Although Acceleo proposes specific mechanisms for user code integration, in ImocaGen, the user code integration mechanism is based on the delegation pattern (see figure 9) for a better independence between user code and generated code. In the classification proposed in (Greifenberg et al., 2015), this type of mechanism is merely called delegation.

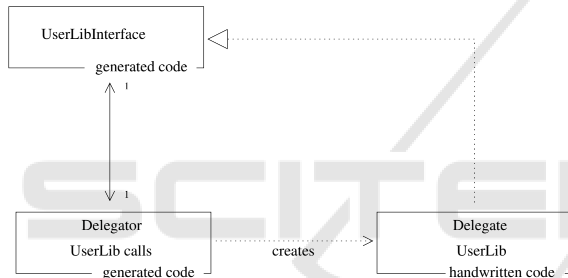


Figure 9: Delegation pattern.

A delegator performs specific function calls in a normalized way. More precisely, a delegator invokes methods which are declared in an interface implemented by a delegate. Then the delegator delegates the task to a delegate. The method calls are generated by the code generator. The interface `UserLibInterface` gives all the signatures of functions and is also generated by the generator. Finally, the user only has to provide an implementation of the interface in the `UserLib` file while respecting the given signatures. This approach, based on the component-based paradigm, ensures a high-level of encapsulation and a clear separation between generated and user code.

5 APPLICATION

In this section, we present a basic and illustrative application in the NXT world: an arm equipped with a gyrometer and a compass, controlled by a rotate motor (see figure 10). In an horizontal plane, the arm has

Figure 10: The NXT brick and the compass needle.

to indicate always the north even if the device moves: the arm plays the role of a compass needle.

For controlling the arm we use a simple PID regulator:

$$\theta = P(heading - 180) + I \sum e + D.gyr$$

where θ is the angle command of the motor, *heading* the value provided by the compass, *gyr* the value of the gyrometer and e the error between *heading* and the desired direction. The error accumulation is calculated over the last ten seconds. The setpoint is 180 in order to avoid the jump between 359 and 0, but if the compass side of the arm points out the south, the other side shows the north.

Finding out good values for P, I and D is generally a long and tedious task, especially in a disturbed environment because physical models are generally far to the reality and numerous phenomena are difficult to simulate and forecast. With ImocaGen, a GUI is generated with a widget for each of these parameters and a widget for the motor (to set its power). Using the GUI helps on tuning the system.

For this application three kinds of communication (local, USB and Bluetooth) have been tested by using ImocaGen. For local communication, a simulator code is generated in Java. The simulator involves a widget for each sensor in order to set the corresponding value. Four sliders are generated, one for each reconfigurable parameter: P, I, D and the power of the motor. The simulator allows to watch the response of the system to given inputs. It helps to test and debug the application behavior. The user interface platform library is based on the SWT library. Using the USB communication allows to test the application on the real embedded system (generation of NXC code). The client platform is a PC and the generated code for the GUI is the same as for local (Java code) except that it does not simulate sensors. A widget is generated for each reconfigurable parameter. For the Bluetooth communication, the client platform is an Android tablet. On the tablet, when the client is

launched, a configuration file is needed for the GUI initialization. Therefore the generator generates only a file `ConfigClient.java` to store into the Android application. Finally, thanks to the tuning tool, the parameters of the control application can be fixed and an embedded code can be generated.

6 CONCLUSION AND FUTURE WORKS

The architecture model IMOCA is designed to process control systems in disturbed environments whereas ImocaGen, which includes IMOCA, is a model of code generation for multiple platforms. ImocaGen allows to generate both embedded C-like code and tuning and control tools in Java. It also integrates a communication model for managing the communications between the control application and the tools. The tools are useful to test the behavior of the system and to allow system evaluation.

This work has several extensions. Other platforms and domains are targeted thanks to the development of specific libraries, as well as the evolution of the ImocaGen meta-model. The quality and the efficiency of the generated code is another challenge because of the embedded platforms constraints. Integrating real-time aspects is also an ongoing work.

REFERENCES

Alonso, D., Vicente-chicote, C., Ortiz, F., Pastor, J., and Alvarez, B. (2010). V^3cmm: a 3-view component meta-model for model-driven robotic software development. *Journal of Software Engineering for Robotics*, 1(1):3–17.

Collett, T. H., MacDonald, B. A., and Gerkey, B. P. (2005). Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia.

DeAntoni, J. and Babau, J.-P. (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*, pages 257–264, Montreal. IEEE Computer Society.

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *Third international conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'12)*, Lecture Notes In Computer Sciences, pages 149–160, Tsukuba, Japan. Springer-Verlag.

Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Nazari, P. M. S., Müller, K., Pérez, A. N., Plotnikov, D., Reiss, D., Roth, A., Rumpe, B., Schindler, M., and Wortmann, A. (2015). A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. *CoRR*, abs/1509.04498.

Guillou, G. and Babau, J. (2013). IMOCA : une architecture base de modes de fonctionnement pour une application de contrle dans un environnement incertain. In *7ème Conférence francophone sur les architectures logicielles*, Toulouse France.

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

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

Ringert, J. O., Rumpe, B., and Wortmann, A. (2013). MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials : IEEE International Conference on Robotics and Automation (ICRA) : Karlsruhe, Germany*, page 3 S.

Simon, D., Espiau, B., Kapellos, K., and Pissard-Gibollet, R. (1997). ORCCAD: Software Engineering for Real-time Robotics. A Technical Insight. *Robotica*, 15(1):111–115.

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

Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G. (2002). Miro - middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497.

SCIENCE AND TECHNOLOGY PUBLICATIONS