Robotic system testing with AMSA framework

Hamza El Baccouri[1] and Goulven Guillou[1] Jean-Philippe Babau[1]

[1] Lab-STICC, CNRS, UMR6285, Univ. Bretagne Occidentale, 29200 Brest, France

(Hamza.elbaccouri, goulven.guillou, Jean-Philippe.Babau@univ-brest.fr)

Abstract.

Nowadays robotic systems are a combination of complex software and hardware components providing sophisticated functionalities. Robotic control systems evolving in an uncertain environment are generally developed case by case for specific deployment platforms. For complex and realistic systems, simulation plays a central role during design by providing testing facilities.

In this paper, we propose to model robotic architectures using the model-driven framework AMSA. In order to facilitate simulation and testing, we propose to incorporate different behaviors in the model through parameterizations. For testing purpose, we define scenarios as a sequence of parameter modifications. From architectural and scenario models, code is generated for the robotic middleware ROS. During simulation, scenarios are used to evaluate different controller behaviors for different contexts.

The approach has been experimented for the MDE challenge: a rover must follow another leader rover. The approach allows the evaluation of control law parameters for different contexts, different behaviors of the sensors, the actuators and the rovers.

Keywords: Software architecture, control, framework, code generation

1 Introduction

A robotic system evolving in an uncertain environment is generally composed of software and hardware components like sensors, actuators, communication ports, drivers, control laws, and so on. In most cases, due to the specificities of the hardware, the environment and the robot mission, development follows a code-centric approach reducing reusability.

Robotics middleware and libraries facilitate implementation by accessing in an abstract manner to machines, networks, sensors and actuators. They also facilitate implementation by providing domain-specific libraries to develop and simulate the controlled system. ROS [4], Player [5], Stage [5] and Gazebo [6] are some of them amongst the most popular. Although these frameworks help to prototype the system, design remains essentially specific to hardware performances, the middleware API and the application domain, focusing on implementation.

Reasoning from an abstract model of the system allows the designer to put aside all the implementation details for the benefit of more global properties of the system like structural, logical and temporal aspects. Model-Driven Engineering (MDE) [1,7] aims to span the conceptual gap between the model and its implementation. Literature proposes lots of MDE-based frameworks to design embedded systems like BRICS model [14], V3CMM [16] and RobotML [17]. They provide facilities to describe robotic architectures. But, considering a given application, the designer still faced a design problem: how to build a correct and efficient system. Like in [15] for aerial robots, we propose to integrate domain-specificities by providing a dedicated architectural style. The approach provides design rules, templates and libraries to facilitate the integration of domain-specific aspects, here control and environment. From models, it is then possible to generate ROS code to target different robotic hardware. The framework is classically based on a component-based paradigm. Scenarios could be tested by configuring, at different times parameter values associated to components.

In this paper, we propose to evaluate the ability of the AMSA framework to set a control law for different contexts. The approach is applied to the leader/follower challenge. Here, the objective of the controller is to maneuver the follower towards a desired position whilst keeping a reference distance. To test the leader-follower controller, we consider different contexts (different sensor, actuator and leader behaviors), showing the facilities provided by AMSA to help on setting control parameters.

The paper is organized as follows: Section 2 presents the AMSA framework and the underlying principles of the ROS code generator. In Section 3, AMSA is applied to the MDE challenge. Section 4 evaluates the approach on parameters setting for different contexts. Finally, Section 5 concludes the paper before presenting future works.

2 The AMSA Framework

The AMSA Framework (Advanced Multihull Simulator for Automation)[13] is a collection of models and tools dedicated to modeling, generation, simulation, evaluation of adaptive control systems architectures evolving in an uncertain environment.

2.1 AMSA Modeling

The framework follows a set of generic principles for architecture modeling and simulation.

AMSA structure The AMSA Framework follows the component-based paradigm [2]. All entities are modeled as *Composite* or *Leaf* components. A *Composite* contains other components. A specific *Composite*, called root, represents the system.

Parameterization A *Leaf* component owns a list of configuration parameters defined by its name, type, description, default, minimal and maximal values. At execution stage, parameters are modified either by a user interface, a scenario definition or by exhaustive exploration.

Communication Components communicate by exchanging data through *Data* buffers. Data has a single producer (component output) and zero or more consumers (component input). A new data production overwrites the previous value, not consumed after reading operation.

Types and instances *Composite* components are singletons (both a type and an instance). Each *Leaf* component is associated to a *template* component. A *template* (playing the role of a class) defines a set of parameters, data inputs and data outputs. It can be instantiated several times, in order to create different components with the same behavior, but with different frequencies (see after) and parameter default values.

Initialization Each *Leaf* component is initialized separately by calling a mandatory *initialize()* method. Composite components are in charge of calling an initialization method of all their children, so it is possible to initialize a simulator simply by initializing the *root* component

Behavior At execution, each component executes a *doStep()* function. The execution follows the Run-To-Completion paradigm. A *doStep()* function is executed independently for other components, considering data inputs at the beginning and producing data outputs at the end of the function.

Timing and scheduling The *doStep()* function is called periodically at the *frequency* required by the component, expressed in Hertz. The data dependency of components defines a dataflow used to compute a logical order of execution. If a loop is detected in the dataflow, the choice of the scheduling order is left to the user.

Architectural Style For simulation, the framework defines six types of components which correspond to the main constituents of a control loop: the controlled system, its environment, the sensors, the actuators, the control law and system observers. Interactions between types of components are constrained to follow classical control data flow (i.e. a sensor produces data only for the control law or system observers).

Scenarios For simulation purpose, the framework proposes to define testing scenarios as a timed sequence of parameters setting. Two kinds of parameters are here considered. The first ones characterize the environment of the control law (the controlled process and its environment, the sensors and the actuators). Each set of parameter values defines a context. The second kind of parameters characterizes a control law configuration. Each set of parameter values leads to a certain quality of control.

Tooling Three modeling tools are proposed by the framework. First, we describe, through a textual editor (based on Xtext [8]), the *templates*. Then, a graphical editor (based on Sirius [9]) is used to instantiate the components and create the links between them. At the end, a scenario is described through a textual editor (based on Xtext). All these editors are based on Ecore models [10] defining the AMSA meta-model through 4 files (*) (a meta-model for each model and a meta-model for generic concepts). One code generator based on the Acceleo technology [3] is used to generate ROS code by analyzing the whole model (composition of the three models).

2.2 Code generation principles

The main principles of ROS code generation are (more details are given in [11]):

- Software in ROS is organized in packages, where packages might contain ROS nodes and dataset. Using AMSA, a single ROS package is created with the name of the root-component.
- A ROS node is created for each *Leaf* component. Reuse of existing code (filtering and control laws) is based on domain-specific functions calls.
- For communication, a ROS topic (publish/subscribe paradigm) is defined for each *Data*. A ROS parameter is defined for each *Parameter*. We then use the ROS facilities (topic publication and Parameter modification user interface) to modify at execution the behavior of a component. A topic and a parameter are identified by its name and are strongly typed.
- A scenario is generated as a specific ROS node implementing a sequence of parameter modifications for a fixed period of time.
- The language of the generated code is C++.

3 Experimentation and results

3.1 MDE challenge

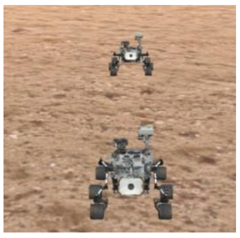

Fig. 1. The leader-Follower challenge

As presented in Figure 1, the proposed approach is illustrated with a system consisting of two rovers: a leader and a follower. The follower must follow the leader while always staying at a safe distance from it. The follower is controlled by controlling the power of the left and right wheels as percentages of max power (-100 to 100). A set of observation commands can be used by the controller that allows obtaining relevant information such as the position of the leader, the position of the follower and the distance between the leader and the follower.

(*) https://github.com/Hbaccouri/AMSA

To reach the given objective, one key challenge is to determine the optimal values of parameters related to control law. Problems are generally related to the unpredictable behavior of the environment of the controller: a sensor may not provide sufficiently accurate measurements, motors are worn, and the leader may have different behaviors. Thus, it is crucial to anticipate and test representative scenarios even an exhaustive check aiming at producing a robust robot. We explore here this problem with two illustrative scenarios:

- First, the nominal scenario with neither error induced by sensors nor power loss, the leader follows a straight trajectory with a constant speed.
- Second, a realistic scenario with the leader following a random trajectory along with reduced motor efficiency and GPS data loss.

3.2 AMSA modeling

We now describe the process followed to model the system using the AMSA framework (*). As seen in Figure 2, the system under study is composed of eight *Leaf* components: one for the controlled rover (*Follower*), one for its environment (here the other rover called *Leader*), four for the sensors (*FollowerGPS, FollowerCompass, LeaderGPS*, *MyDistanceMonitor*), one for the actuator (*MyNormalizedPower*), and finally one for the control (*MyFollowerController*).

The position and the heading of the follower (*posX*, *posY and heading*) depend on its behavior (defined by *Follower* considering *normalizedPowerL/R*). Compass, resp. GPS, is a simple sensor communicating the rover heading, resp. the rover position. *MyFollowerController* computes the *Left/Rightpower* commands to correct distance error between the follower and the leader by comparing GPS positions and considering rovers' distance and speed. *MyNormalizedPower* ensures that the emitted power (*normalized_power*) is in the range of -100 to 100. *MyDistanceMonitor* computes the real distance between the two rovers, based on *posX* and *posY* values.

Before the description of components, a template is edited for each *Leaf* component (7 templates here). Then it is possible to instantiate the 8 Leaf components (GPS template is instantiated twice) through the AMSA graphical editor.

As mentioned before, parameters characterize the different contexts. Here, we define three parameters for the sensors, actuators and environment of the controlled process:

- The parameter *lostDataGPS* characterizes the percentage of lost data by the LeaderGPS;
- The parameter *motorCoef* defines the percentage of the power actually considered;
- The parameter *behavior* characterizes the different trajectories followed by the leader.

Other parameters are used to configure the control law which is a proportional regulator inspired by [12]. Hence, two proportional gains noted *Kv* and *Kw* are defined, the former for the linear velocity and the latter for the angular velocity. The linear and

(*) https://github.com/Hbaccouri/AMSA

angular velocities are corrected proportionally to the follower position error. Since we want the follower to be in line and behind the leader, the error is decomposed in two components with respect to the angle between the axis linking the two rovers and the heading of the follower.

The last modeling step is the definition of the scenarios (see Figure 3). We start first with the simple case where the leader goes straight (Behaviors 1 and 3) with no error induced by the GPS and the motor operating correctly. In this context we test four different values of the *Kv* gain (0.05, 0.1, 0.15 and 0.35) aiming at determining the best value by evaluating the quality of control (here the maximal distance between the two rovers).

In the second scenario, we suppose that the leader moves randomly (Behavior 4), the leader GPS provides 90% of the actual *posX/Y* and the actuator *MyNormalizedPower* ensures only 60% of the required power. We test here two values of the *Kv* gain (0.35 and 0.05).

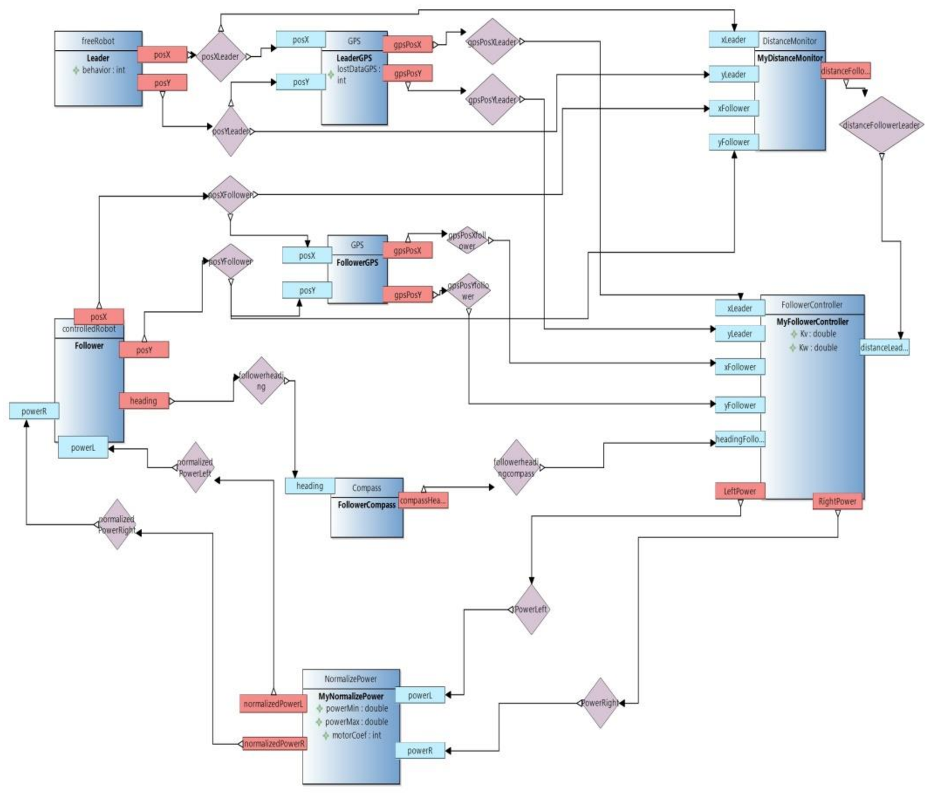

Fig. 2. AMSA modeling of the system under study via a Sirius diagram

```
Scenario scenario1 {
ParamEvent at 0 on scenarioNode {
    scenarioNode.motorCoef = 100;
    scenarioNode.lostDataGPS = 0;
     scenarioNode.behavior = 1;
    scenarioNode.Kv = 0.35;
    scenarioNode.Kw = 0.005; }
ParamEvent at 200 on scenarioNode {
     reset();
     scenarioNode.behavior = 3;
     scenarioNode.Kv = 0.15; }
ParamEvent at 400 on scenarioNode {
    reset();
    scenarioNode.behavior = 1;
    scenarioNode.Kv = 0.1; }
ParamEvent at 600 on scenarioNode {
    reset();
    scenarioNode.behavior = 3;
    scenarioNode.Kv = 0.05; } }
```

```
Scenario scenario2 {
ParamEvent at 0 on scenarioNode {
    scenarioNode.motorCoef = 60;
    scenarioNode.lostDataGPS = 10;
    scenarioNode.behavior = 4;
     scenarioNode.Kv = 0.35;
    scenarioNode.Kw = 0.005; }
 ParamEvent at 100 on scenarioNode {
     reset();
     scenarioNode.Kv = 0.05;  } }
```

Fig. 3. AMSA scenarios to test the Leader/Follower controller

3.3 ROS generation

Once the model is instantiated through the graphical editor, Acceleo scripts generate the C++ code associated to each ROS node. The generation leads to 9 C++ files [(*)] (a ROS node file for each Leaf component node and one for the scenario). A launch file and commands for the CMakeLists.txt file are also generated. As presented in Figure 4, the ROS architecture follows the AMSA architecture.

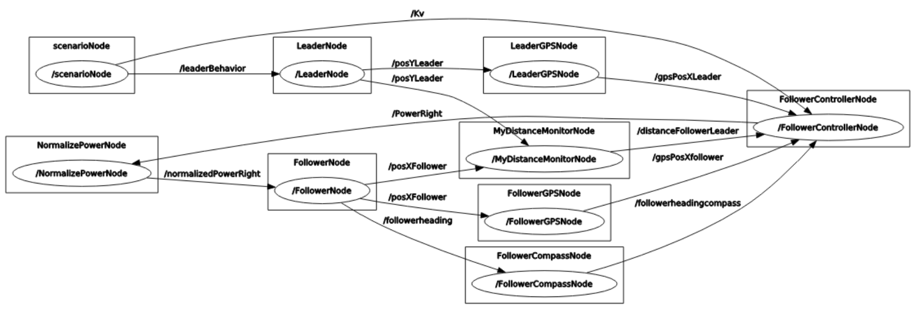

Fig. 4. ROS graph at runtime: nodes as ovals, topics as squares

(*) https://github.com/Hbaccouri/AMSA

4 AMSA evaluation

4.1 ROS simulation

The purpose of this section is to illustrate how to evaluate a set of parameter values for the controller. *MyFollowerController* is the main node since it implements the control algorithm and applies control effort aiming at preserving a reference distance between the leader and the follower. The Qt Guide User Interface dedicated to ROS is used in order to assist the user to evaluate the quality of control, here the distance between the leader and the follower.

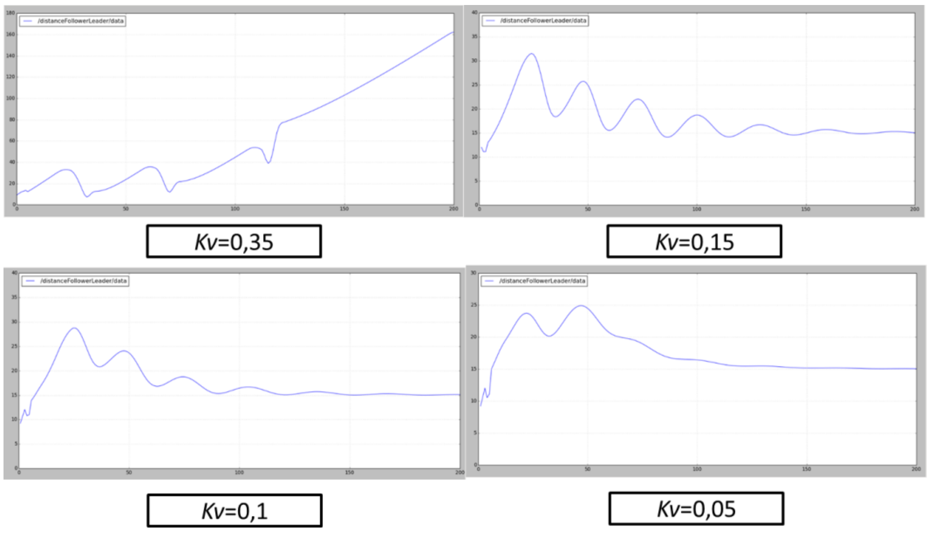

Fig. 5. Scenario 1 results: Leader-Follower distance with *Kv*=0.35, *Kv*=0.15, *Kv*=0.1 and *Kv*=0.05 (X-axis represents time in seconds)

Figure 5 illustrates the nominal scenario where there is neither error induced by GPS nor power loss. In this context, we tested 4 different gain values. First, we start with the leader going straight ahead with a gain value *Kv=0.35*. In this case the follower could not catch up with the leader. Using *Kv=0.15*, we note that the maximal distance between the two rovers was 32m and it takes almost 150s to reach the reference distance. Oscillations have also noted. By reducing the gain value to *Kv*=0.1, the maximal distance has not exceeded 29m with less oscillations. A gain value of *Kv*=0.05 is here a best value for this nominal context, since that the maximal distance observed was 25 and it took the follower only 120s to be within a safe distance.

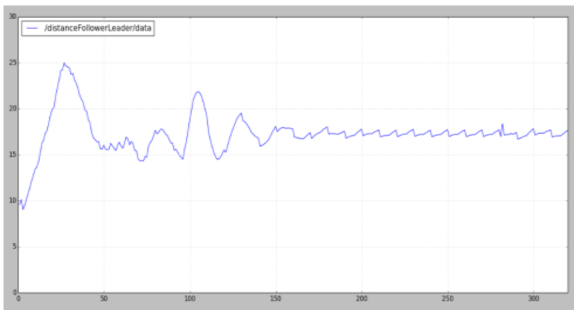

Fig. 6. Scenario 2 results: Leader-Follower distance with *Kv*=0.35, *Kv*=0.05

Figure 6 shows the result for the second scenario. We note that a gain value of *Kv*=0.35 (on the left side of the Figure) is a better value here despite the random trajectory of the leader, leader GPS data loss and a less power efficiency of the motor. Apart from the beginning, where the follower has not yet received the current position of the leader, the maximal distance between the two rovers oscillates around the reference distance fixed here at 15. Although, we note also some local peaks and maximal distance not exceeding 22. At t=100, we test *Kv* =0.05 (on the right side of the Figure). Obviously the best value for the nominal scenario is not well suited here. The maximal distance between the rovers is practically stable at 18, more than expected (15 here). To conclude, an optimal set of parameter values for control law depends on the context of the controller.

4.2 Discussion

As illustrated previously, the use of the AMSA framework has been beneficial on several aspects. The main advantage of the proposed approach consists on using simple models defined at a high level of abstraction. The introduction of models hides technological details and the architecture style helps the designer in combining the different elements. Code generation improves the code generation productivity by limiting the number of lines to program to domain-specific parts. The ROS generated code can be used for simulation and may be embedded on a ROS-compatible robot.

The development of context-aware applications is always a challenging and complex task. There is a growing need for more flexible adaptive systems able to operate in dynamic environments coping with unanticipated situations. An optimal behavior for a context may be inappropriate for another. To deal with this issue, AMSA allows defining different simulation scenarios to determine the optimal value for each scenario. To do so and instead of dealing with different versions of the code, we propose to just add parameters to components. There are two types of parameters. Parameters related to the controlled component, its environment, the sensors and actuators represent the context. Parameters associated to the controller are used to evaluate different behaviors of the control law. To conclude, AMSA allow testing multiple scenarios by changing parameter values in our quest for the adaptability of the control law to respond to context changes.

5 Conclusion

In this paper, we present an approach to generate ROS simulation code from a model of the architecture based on the AMSA framework. The case of the MDE challenge based on a leader/follower problem has been used for experiments. It illustrates the interest of having parameterization, scenario modeling features in order to evaluate different control configuration regarding different environmental contexts.

For future works, we will include adaptive behavior to the robot by a dynamic reconfiguration of the parameters at any time depending of the context.

6 References

1. A. Ramaswamy, B. Monsuez, and A. Tapus, “Model-driven software development approaches in robotics Research” International Workshop on Modeling in Software Engineering, 43-48, Hyderabad, India, 2014.
2. H. Bruyninckx, M. Klotzbucher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, “The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems” ACM Symposium on Applied Computing, SAC, 1758-1764, New York, USA, 2013.
3. https://www.eclipse.org/acceleo/
4. M. Quigley, B. Gerkey, K.Conley, J. Faust, T.Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, “Ros: an open-source robot operating system” IEEE International Conference on Robotics and Automation, ICRA, Kobe, Japan, 2009.
5. B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems” International Conference on Advanced Robotics, vol. 1, 317–323, Coimbra, Portugal, 2003.
6. N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," International Conference on Intelligent Robots and Systems, IROS, vol.3, 2149-2154, Sendai, Japan, 2004.
7. C. Schlegel, T. Hassler, A. Lotz and A. Steck, "Robotic software systems: From code-driven to model-driven designs," International Conference on Advanced Robotics, 1-8, Munich, Germany, 2009.
8. https://www.eclipse.org/Xtext/
9. https://www.eclipse.org/sirius/
10. http://www.eclipse.org/ecoretools/
11. H. EL Baccouri, E. Lavigne, G. Guillou and JP. Babau, “ROS code generation from AMSA framework for robotic systems testing”. National Conference on Software and Hardware Architectures for Robots Control, Saint-Tropez, France, 2018.
12. Choi IS., Choi JS. “Leader-Follower Formation Control Using PID Controller” International Conference on Intelligent Robotics and Applications, 625-634, Montreal, Canada, 2012.
13. E. Lavigne, G. Guillou and JP. Babau, “AVS, a model-based racing sailboat simulator: application to wind integration”. IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control, CESCIT, 88-94, Faro, Portugal, 2018.
14. M. Klotzbuecher, N. Hochgeschwender, L. Gherardi, H. Bruyninckx, G. Kraetzschmar, D. Brugali, A. Shakhimardanov, J. Paulus, M. Reckhaus, H. Garcia, et al, “The brics component model: A model-based development paradigm for complex robotics software systems” ACM Symposium on Applied Computing, SAC, Coimbra, Portugal, 2013.
15. JL Sanchez-Lopez, R. A. Suarez Fernandez, H. Bavle, C. Sampedro, M.Molina, J.Pestana, and P. Campoy, ”AEROSTACK: An Architecture and Open-Source Software Framework for Aerial Robotics”. International Conference on Unmanned Aircraft Systems, 332-341, Arlington, USA, 2016.
16. D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, “V3cmm: A 3-view component meta-model for model-driven robotic software development”, Journal of Software Engineering for Robotics, vol.1 3–17, 2010.
17. S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, “Robotml, a domain-specific language to design, simulate and deploy robotic applications”. Simulation, Modeling, and Programming for Autonomous Robots, vol.7628, 149–160, 2012.