RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications

Saadia Dhouib[1], Selma Kchir[2], Serge Stinckwich[3,4], Tewfik Ziadi[2], and Mikal Ziane[2]

[1] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
Point Courrier 94, Gif-sur-Yvette, F-91191 France
[2] UMR CNRS 7606 LIP6-MoVe, Université Pierre et Marie Curie, Paris, France
[3] UMR CNRS 6072 GREYC, Université de Caen-Basse Normandie/ENSICAEN, Caen, France
[4] UMI IRD 209 UMMISCO
IFI/Vietnam National University, Hanoi, Vietnam
saadia.dhouib@cea.fr, serge.stinckwich@ird.fr, mikal.ziane@lip6.fr

Abstract. A large number of robotic software have been developed but cannot or can hardly interoperate with each other because of their dependencies on specific hardware or software platform is hard-wired into the code. Consequently, robotic software is hard and expensive to develop because there is little opportunity of reuse and because low-level details must be taken into account in early phases. Moreover, robotic experts can hardly develop their application without programming knowledge or the help of programming experts and robotic software is difficult to adapt to hardware or target-platform changes. In this paper we report on the development of RobotML, a Robotic Modeling Language that eases the design of robotic applications, their simulation and their deployment to multiple target execution platforms.

Keywords: Domain-Specific Languages, Generative Programming, MDE, robotic application.

1 Introduction

Large amounts of robotic software have been developed but cannot or can hardly interoperate with each other because their dependencies on specific hardware or software platform is hard-wired into the code. Thus, changing one or several of the used hardware components in the application implies several time consuming changes in the code. In addition, robotic software is difficult to adapt to target platform changes. Consequently, robotic software is hard and expensive to develop because there is little opportunity of reuse. Moreover, robotic experts can hardly develop their application without programming knowledge or the help of programming experts. This knowledge is not only related to algorithm programming but to the arcanes of the chosen simulation platform and the details of drivers of sensors and actuators.

I. Noda et al. (Eds.): SIMPAR 2012, LNAI 7628, pp. 149–160, 2012.

Defining robotic applications using appropriate notations and abstractions and automatically generating executable code could be a solution to deal with variability problems and to hide the lower level programming details to robotics experts. Domain-specific languages (DSLs) have been introduced, based on thorough understanding of the application domain, to express solutions at the level of abstraction of the problem domain.This assumes that along with the DSL itself a whole toolchain is provided to automate as much as possible the lower-level task and especially code generation [4,10].

In this paper we propose the RobotML DSL and a toolchain to address these issues. In section 2, we outline the requirements for RobotML that is then presented in section 3. Afterwards, we describe the RobotML toolchain, from modeling to code generation in section 4. Then, a case study is outlined in section 5, validating the proposed DSL on a real robotic use case. Section 6 discusses how the proposed RobotML DSL fulfills the requirements presented in Section2. In Section 7, we outline previous work related to our approach and section 8 concludes the paper.

2 Requirements

In this section we specify the requirements of the ideal autonomous robotics DSL. Some of the requirements deal with the DSL itself while some are more related to its implementation and especially to how it will be compiled and run. In this paper we will report on which requirements were achieved and which ones were left for further study:

1. **Ease of use.** Using the DSL should be in the reach not only of programming experts but of robotics experts and ideally of mere robotics users.
2. **Specification of component-based robotic architectures.** Assuming that most robotics software is nowadays component-based, the DSL must allow the specification of component-based architectures of autonomous robotic systems.
3. **Neutrality regarding architectural styles.** The DSL must not impose a specific robotic architectural style (deliberative, reactive, hybrid, ...).
4. **Multiple, heterogeneous target platforms.** As long as the expected components are provided and conform to the architecture, the latter must be executable on robots or on a simulator. In addition, it must be possible to run some components of a given architecture on one platform and other components on another.
5. **Target-platform independence.** Even though target-platform independence is difficult to achieve, the DSL should be as independent as possible of the specificities of the execution platforms.
6. **Smooth evolution of the supported platforms.** Code generation should be as agile as possible so that supporting a new platform can reuse common transformations. Similarly, supporting a new version of a target platform should capitalize as much as possible on the previous implementation.

7. **Smooth evolution of the DSL.** Ideally it should be possible to change at least some superficial aspects of the DSL without having to build a completely new implementation.
8. **Reasoning.** It should be possible to reason on the architecture especially to check non-functional properties such as real-time constraints or energy consumption.

3 RobotML Domain Specific Language

This work aims at providing a domain specific language (and related tools like editors, model validation, code generators) suitable to specify missions, environments and robot behaviours that will be specified by robotics designers. Independently from the target platform, the DSL will help the robotic system designers to define:

- The system's architecture (i.e. its internal structure). In fact, the DSL will ease the definition of specific robotic architecture (reactive, deliberative, hybrid) and specific components that form the architecture (sensors, actuators, planners, mapping, etc.).
- The communication mechanisms between components (sending/receiving of event notifications and data).
- The behaviour of robotic components that form the system's architecture.

The main design entries of the DSL are the robotic ontology [3] developed in the frame of the French research project Proteus[1] and a state of the art study on languages and tools for robotic systems. The latter has helped to identify the requirements presented in the previous section. The former has helped to build the domain model. Once the domain model is built, we implemented the DSL as a UML profile (i.e. extension of the UML meta-model). Then we developed the graphical modeling tool as an extension of the Papyrus modeling tool[2]. In the following, we justify the choice of using a robotic ontology to build the domain model of the RobotML DSL, and then we present the domain model (i.e. meta-model) of the DSL.

3.1 Rationale of Using an Ontology during the Design of RobotML

Ontologies are mainly used in Artificial Intelligence and Semantic Web communities to represent knowledge, as a set of concepts and relationships of a domain. Several works have already used ontologies for their semantic or structural synergy with DSLs [7,11]. In our case, the main benefit from using the ontology in the design process of the DSL is *reusing a robotics experts knowledge base to enrich the DSL domain model* [6]. From our experience, using an ontology has benefits as well as drawbacks. In fact, reusing an ontology facilitates the DSL designers task by providing a set of concepts that are specific to the

[1] `http://www.anr-proteus.fr/?q=node/111`
[2] `http://www.eclipse.org/modeling/mdt/?project=papyrus`

robotic domain and that can be directly reused to define the DSL meta model. Examples of those concepts include `Robot`, `SensorSystem`, `ActuatorSystem`, `LocalizationSystem`. On the other hand, a lot of concepts are not useful for defining the DSL domain model. For example the concept of `Interaction` is not useful in the DSL, we have instead used the concepts of `Port` and `Connector` to capture the concept of an interaction. So having an ontology as a design entry implies that the DSL designer will filter the ontology concepts that are useful for the DSL domain model. This filtering task is not straightforward, specially when the DSL designer does not master the ontology language and editing tools.

3.2 Domain Model

The domain model of a given DSL defines the concepts of a language and their relationships. In this section, we present the domain model of the RobotML DSL based on the requirements defined in the previous section and the knowledge base provided by the ontology. We have used UML class diagrams to represent the domain model. The packages structuring the RobotML domain model and relationships between them are presented in Fig. 1.

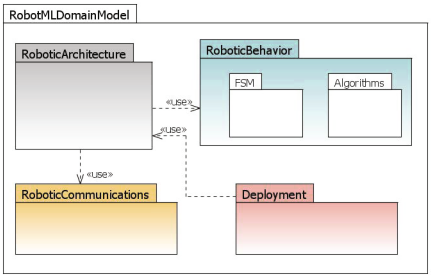

Fig. 1. RobotML Domain Model

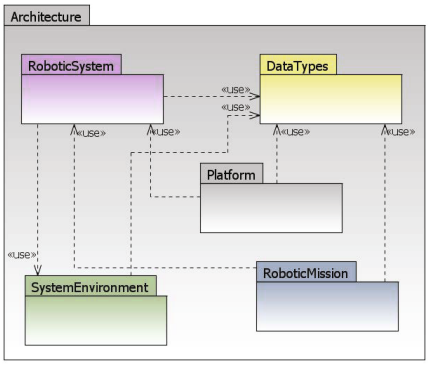

Fig. 2. RoboticArchitecture Package

Architecture. Fig. 2 shows a general view of the RobotML architecture domain model package. This package contains five sub-packages:

1. The *robotic system* package describes the concepts that help define and compose a robotic system. The `System` concept corresponds to the `Component` concept (as found in the component-based approach). The term *system* is more appropriate to describe a robotic component, this is due to the fact that `System` is more meaningful for a robotician than `Component`. Fig. 3 some of the concepts defined in this package. A `System` is composed of properties, ports and connectors. Properties could be either the system's subsystems or attributes (for storing configuration values for example). Ports and connectors allow systems to interact. This package includes also the robotic specific concepts such as sensors and actuators.

2. The *system environment* package defines the concepts composing the environment where robots evolve, since we not only model the robotic system but also its environment (for example for simulation purpose).
3. The *data types* package defines the types of data that will be exchanged between robotic components, between algorithms, etc.
4. The *robotic mission* package describes the concepts that are needed to define an operational mission and which are used by components of the architecture performing it.
5. The *platform* package defines the concept of platform which represents the execution environments for the robotic system (i.e. robotic middleware, robotic simulator).

Communications. Communications between robotic systems formalize data exchange and service calls between them. Communications are refined through the aspect of ports and connectors. A port formalizes an interaction point of a system. `Port` is an abstract concept that is refined through two concepts. On the one hand, we define the concept of `DataFlowPort` which is related to the publish/subscribe model of communication. `DataFlowPort` enables dataflow-oriented communication between systems, where messages that flow across ports represent data items. On the other hand, we define the concept of `ServicePort` that supports a request/reply communication paradigm, where messages that flow across ports represent operation calls.

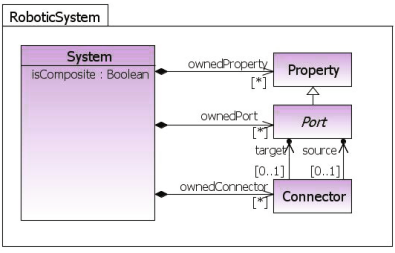

Fig. 3. RoboticSystem Package

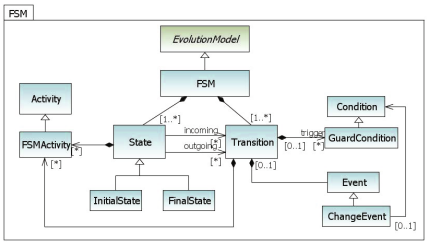

Fig. 4. Details of the FSM Package

Behaviour. The concept of `EvolutionModel` is used to describe the behaviour of the robotic system. An evolution model can be defined using algorithms or finite state machines. Fig. 4 shows the `FSM` concept inheriting from `EvolutionModel` and composed of states and transitions. `States` are composed of `FSMActivities`, meaning that activities can be executed during a `State`. `Transitions` are also composed of `FSMActivities`, meaning that if a transition is fired, an activity can be executed as an effect. An activity is a behaviour that is specified using an algorithm.

Deployment. This package specifies a set of constructs that can be used to define the assignement of a robotic system to a target robotic platform (a middleware or a simulator). The deployment is important because it feeds code generators with the information on which platform the system will be executed.

4 The RobotML Toolchain

RobotML provide a toolchain for robotic development from modeling to software simulation and deployment on real robots. This toolchain, illustrated in Fig. 5, is based on the Eclipse Modeling Project[3] that supports model-driven engineering approach. We have used Papyrus and Acceleo[4] plugins integrated to Eclipse for modeling and code generation.

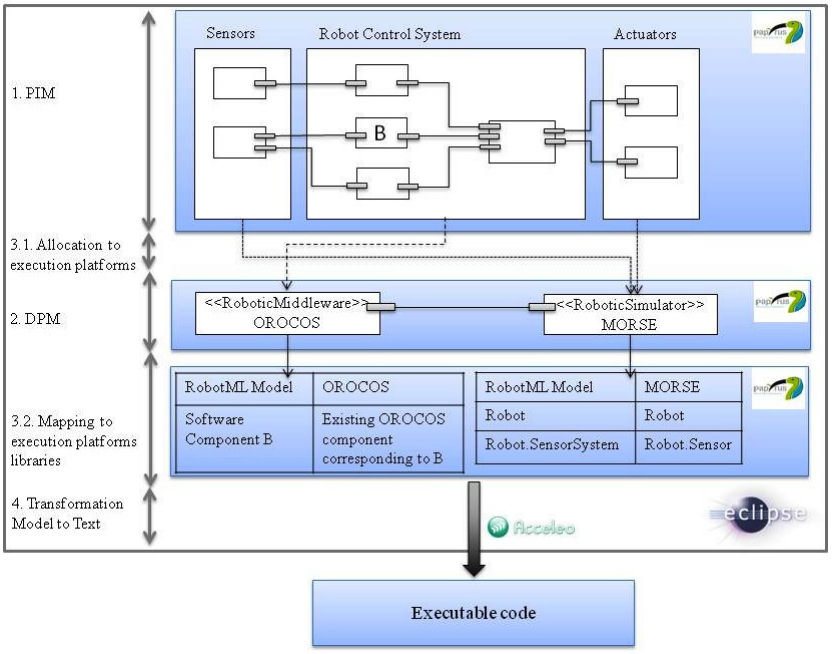

Fig. 5. The RobotML toolchain

The RobotML user starts by designing a model (PIM, Platform Independent Model) of a specific scenario where sensors, actuators and the control system of the robots are represented. The next step is a definition of a deployment platform model (DPM), which consist of a set of robotic middlewares and/or simulators connected with each other to form an execution platform for the PIM. Then,

[3] http://www.eclipse.org/modeling/
[4] http://www.acceleo.org/

after the validation of the model, the user defines a deployment plan where he/she chooses to which robotic middleware (OROCOS-RTT[5], RTMaps[6], Urbi[7] or Arrocam[8]) and to which simulation engine (MORSE[9] or CycabTK[10]) code will be generated. The deployment plan is built in two steps:

- First, the designer allocates the components of the PIM to the modeled execution platform parts. For example, the control system of the robot can be allocated to a robotic middleware and sensors/actuators can be allocated to a robotic simulator. The designer can also specialize the robotic system components by setting some of their properties to match the specificities of runtime platforms.
- Second, the designer has the possibility of reusing existing components already deployed in the component library of the target robotic platform. In fact, a mapping to target platforms component libraries can be established by the user.

Finally, from the deployment plan information, the code is automatically generated. Code generators from the DSL to the aforementioned middlewares and simulators have been developed in the frame of the PROTEUS project.
Overall, the ROBOTML DSL provides a common ground for designing and implementing component-based robotic systems. We illustrate this in the next section with a case study.

5 Case Study: Urban Challenge

In order to validate the proposed DSL on industrial examples, several case studies (called challenges) have been designed by the PROTEUS project partners. In this section, we will focus on the Urban Open-Access Robotic Platform[11] challenge that deals with the problem of intelligent transportation systems (autonomous cabs). In the following, we present the urban challenge model in accordance with the four main parts of the proposed DSL: architecture, communication, behaviour and deployment then code generation.

5.1 Modeling

Using the ROBOTML DSL, we have represented the modules of the challenge in a component-based model.

[5] http://www.orocos.org/
[6] http://intempora.com/
[7] http://www.urbiforge.org/
[8] http://effistore.effidence.com/
[9] http://www.openrobots.org/wiki/morse/
[10] http://cycabtk.gforge.inria.fr/
[11] http://www.anr-proteus.fr/?q=node/64

Architecture. Fig. 6 illustrates the global architecture of the urban challenge model. The control system consists of controller, trajectory planning, localization and obstacle detection components. It defines the behaviour of the robot during its mission. The control system of the robot takes inputs from sensors (*LIDAR3D, Odometery, RTK_ GPS_ IMU* and *Front Camera*) and sends commands to actuators (*Brake, Steering* and *Motor*) through data flow communications. The `Localization` component calculates the position of the robot and returns the deviation of the robot with respect to the trajectory it must follow. `Trajectory Planning` computes the trajectory to follow from the current position of the robot in comparison with its goal (the goal position is initially specified by the user). `Obstacle Detection` is defined to deal with dynamic changes in the environment. It sends information to `Control` whether a new obstacle is detected. The `Control` component aims at transforming the received data from the components `Localization`, `Trajectory Planning` and `Obstacle Detection` into commands to `Actuators` components, which represent the actuators of the robot. A snapshot of the Papyrus-based modeling environment is shown in Fig. 7. At the right side of the component definition diagram, we can see the customized palette that contains RobotML concepts used for defining the aforementioned robotic components.

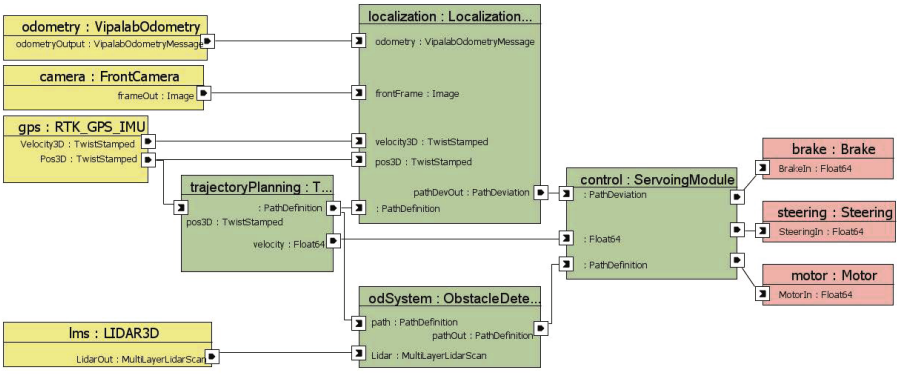

Fig. 6. Simplified urban challenge architecture diagram

Communications. In Fig. 6, components are connected through *Data Flow* ports. Let us take for example the components *Localization* and *RTK_ GPS_ IMU*. The synchronization policy for data exchange (synchronous/asynchronous mode) between these components is specified in addition to the buffer size for data storage. Those information are specified by setting ports attributes in the properties view of the modeling environment (Fig. 7).

Bevahiour. Components (except sensors and actuators) have a behaviour specified by a state machine or an algorithm (see section 3.2). At the right side of the modeling environment (Fig. 7), the state machine diagram of the component `Localization` is shown. The first state is the Kalman filter which handles data sent by sensors and returns an estimation of the pose of the robot. If the position

of the robot changes (guard:positionChanged), the state *ComputePathDeviation* is activated and the computed deviation is returned to the controller component.

Deployment. In the case of the urban challenge, we have modeled an execution platform that contains a robotic middleware, namely OROCOS, communicating with a robotic simulator, namely Morse. The deployment plan contains allocations of sensors (yellow components in Fig 6), actuators (red components in Fig 6) to MORSE and allocations of the control system (green components in Fig 6) to OROCOS.

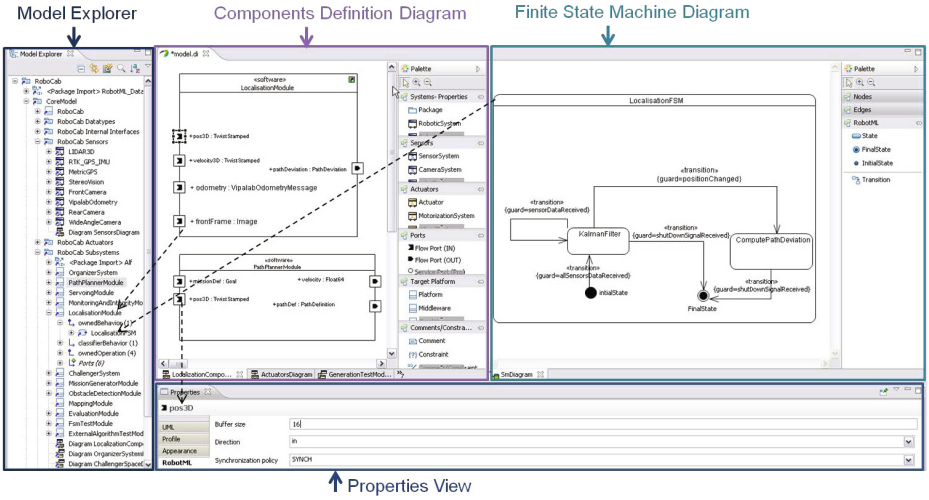

Fig. 7. The RobotML modeling environment

5.2 Code Generation

The RobotML toolchain integrates several code generators defined by Proteus partners to several robotics middlewares and simulators engines (cf. section 4). The process of translating a model to code is to perform Model to Text (M2T) transformations from the DSL to text artifacts (source files, configuration files, etc.) needed to create an executable application. Starting from the model of the urban challenge and the deployment plan defined in the model, users can generate code for several platforms.

6 Discussion

RobotML The main objective of the RobotML DSL was to propose to the robotics community a domain specific language which facilitates the development of robotics applications. In this context a set of requirements was identified in Section 2. In this paper, the RobotML DSL presented an answer to achieve these requirements. In this section we report on which of these requirements were achieved by the RobotML DSL and which ones were left for further work.

1. **Ease of use** Thanks to RobotML, DSL users can model the components of their missions without mastering programming languages of robotics platforms. In fact, platforms details are hidden to the DSL users and concepts used in RobotML are based on robotics ontology.
2. **Specification of component-based robotic architectures** As presented in Section 3, the RobotML DSL includes the architecture part which gathers the concepts of *System*, *Port*, *Connector*. The latters allow the specification of component-based robotic architectures. For instance, Fig. (6) shows an example of such architecture.
3. **Neutrality regarding architectural styles** Most robotic systems software architectures are based on components, the only architecture taken into account in RobotML is component-based. Consequently, any robotic architecture (hybrid, deliberative, etc) can be specified using an approriate combination of components. For example, the used architecture in the scenario of Fig. 6 is hybrid.
4. **Multiple, heterogenous target platforms** Thanks to the use of ROS, components deployed into robotics platforms can easily communicate with components deployed into simulators or directly with real robots. Thus, the code generated from RobotML can be executable on both real and simulated robots.
5. **Target-platform independence** Using RobotML and thanks to the abstract concepts of the DSL, the architecture model of the robotic application is independent from the target platform (PIM).
6. **Reasoning** RobotMLenable roboticist to model some non-functional properties of the system notably timing properties for software components (period, deadline, WCET, priority). Such timing properties will feed schedulabilaty analysis tools e.g. Cheddar [9]. Another types of reasoning could be considered for robotic systems specified with RobotML given the extension of the language to integrate the adequate non functional properties modeling.

The requirements *smooth evolution of the supported platforms* and *smooth evolution of the DSL* were left for future work.

7 Related Work

At the moment, there are not that many proposals to use MDE (Model Driven Engineering) in the context of robotic systems. One of the first initiatives to promote this approach was Blanc et al. [2] who applied MDE to develop control ssoftware for an AIBO robot.

SmartSoft [8] combines a service-oriented component-based with a Model-Driven Software Development approaches. The SmartSoft[12] component model relies on a small and fixed set of communication patterns, e.g., request/response and publish/subscribe, that define the semantics of the interface (externally visible ports) of components. The component model is represented by a meta-model called

[12] `http://smart-robotics.sf.net/`

SmartMARS (Modeling and Analysis of Robotic Systems). Like RobotML, a concrete implementation of this meta-model has been done as a UML profile and the proposed Eclipse-based SmartMDSD model-driven software development toolchain is based on Papyrus as well. But unlike RobotML, the SmartSoft approach does not provide the possibility to describe components behaviour at the same abstraction level than components specification level.

BRICS (Best Practice in Robotics) is an ongoing FP7 EU funded project[13] that also promotes the model-driven engineering (MDE) approaches in order to solve robotic software engineering issues. Among all the activities done in the project, a features-based model toolchain was proposed in order to reflect variabilities in robotic systems [5] and a meta-model called BCM was defined for describing a minimal component model[14] suitable for code generation for multiple targets (ROS, OROCOS-RTT). Unlike RobotML, BRICS meta-model does not enable specification of composability, component's behaviour and robotic specific components.

The 3-View Component Meta-Model (V^3CMM) [1] relies on the use of the OMG Meta-Object Facility (MOF) instead of using UML. It aims to provide designers with an expressive yet simple platform-independent modeling language for component-based application design. V^3CMM is aimed at allowing designers (1) to model high-level reusable components, including both their structural and behavioural facets (modeling for reuse); (2) to build complex platform-independent designs up from the previous components (modeling by reuse); and (3) to automatically translate these high-level designs into lower level models or into different implementations, isolating functionality from platform details. Compare to RobotML, it is worth noting that although V^3CMM has been used mainly in robotics, it does not contain any specificities about this domain.

8 Conclusion

In this paper we have reported on the RobotML domain-specific language and toolchain. RobotML is easier to use than the targeted robotic execution or simulation platforms because low level details have been hidden behind easier to manage abstractions. Not all of these abstractions are directly related to robotics and some do relate to defining component-based architecture but a sizeable amount of low-level programming knowledge has been put into the code-generation transformations. Early usage reports suggest that even though the overall development time of a robotic application using RobotML has not significantly decreased, using RobotML induces the following advantages:

- more time is spent on design than on dealing with low level details,
- the architecture is made explicit,
- switching to a new target platform is much easier.

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

[14] `http://www.best-of-robotics.org/pages/publications/BRICS_Deliverable_D4.1appendix.pdf`

The RobotML DSL and toolchain have been designed in the context of the ANR Proteus projet. This project funded by the French research agency, gather 14 academic and industrial partners. The RobotML toolchain will be available with an open-source licence in the future.

References

1. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Alvarez, B.: V^{3}CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Journal of Software Engineering for Robotics 1(1), 3–17 (2010)
2. Blanc, X., Delatour, J., Ziadi, T.: Benefits of the MDE approach for the development of embedded and robotic systems. In: Proceedings of the 2nd National Workshop on "Control Architectures of Robots: from Models to Execution on Distributed Control Architectures", CAR 2007 (2007)
3. Dhouib, S., Du Lac, N., Farges, J.L., Gerard, S., Hemaissia-Jeannin, M., Lahera-Perez, J., Millet, S., Patin, B., Stinckwich, S.: Control architecture concepts and properties of an ontology devoted to exchanges in mobile robotics. In: 6th National Conference on Control Architectures of Robots (2011)
4. Gerard, S., Babau, J.P., Champeau, J.: Model Driven Engineering for Distributed Real-Time Embedded Systems. Wiley-IEEE Press (2005)
5. Gherardi, L., Brugali, D.: An Eclipse-based Feature Models Toolchain. In: Proc. of the 6th Workshop of the Italian Eclipse Community (Eclipse-IT 2011) (2011)
6. Lortal, G., Dhouib, S., Gérard, S.: Integrating Ontological Domain Knowledge into a Robotic DSL. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 401–414. Springer, Heidelberg (2011)
7. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.-M.: Weaving Variability into Domain Metamodels. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 690–705. Springer, Heidelberg (2009)
8. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: Communication patterns as key for a robotics component model. Introduction to Modern Robotics (2012)
9. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In: Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada, SIGAda 2004, pp. 1–8. ACM, New York (2004)
10. Steck, A., Lotz, A., Schlegel, C.: Model-driven engineering and run-time model-usage in service robotics. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, GPCE 2011, pp. 73–82. ACM, New York (2011)
11. Walter, T., Ebert, J.: Combining DSLs and Ontologies Using Metamodel Integration. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 148–169. Springer, Heidelberg (2009)