RobotML for Industrial Robots: Design and Simulation of Manipulation Scenarios

Selma Kchir* Saadia Dhouib† Jérémie Tatibouet† Baptiste Gradoussoff* Max Da Silva Simoes*

* CEA, LIST, Interactive Robotics Laboratory, PC 178 - Digiteo MOULON, Gif-sur-Yvette, F-91191 France
† CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, PC 94, Gif-sur-Yvette, F-91191 France

Abstract—**Robotic systems are a typical example of complex systems. Their design involves a combination of different technologies, requiring a multi-disciplinary approach. This is particularly challenging when a robotic system is required to interact either with humans or other entities within its environment. To tackle this complexity, we propose a design and validation approach based on MDE (Model-Driven Engineering) principles for industrial manipulators. We propose an extension of RobotML for manipulation, a modelling environment based on the Papyrus tool, which was developed specifically for the robotics domain. The extension is aiming to model a complete robotic setting, including protagonists, objects, their properties, the interactions between them, the services provided by the robots, and the actions they can perform. Then we propose to use model execution techniques to validate the design models. We illustrate our approach on a robotic scenario dedicated to the Sybot collaborative robot.**

I. Introduction

Collaborative robotics and computer-aided teleoperation are more and more used in manufacturing to perform tasks that are harmful and hard for humans. These systems are not only hard to develop because of their heterogeneity (control, communication, coordination, software, hardware, etc.) but also their usage imposes considerable levels of safety and real-time constraints.
Therefore, industrial manipulators are faced with several challenges. One of the primary challenges is to simplify the configuration and use of robotic systems. That is, by allowing the design of robotic systems to be accessible not only to programming experts but also to robotics experts and even to actual robotic system users, would, among other things, greatly help customers to more precisely express their requirements. Another major challenge is how to deal with platform variability. The term « platform » here covers a variety of concerns, such as mechanisms, control strategies, middleware, etc. This remains an important issue in robotics systems design and implementation. Yet another key challenge is to simplify the development of robotics applications. Currently, developers spend a lot of time developing and maintaining their applications. A significant amount of time could be saved using automated code generators, which could produce more reliable and more easily maintainable code.
The goal of this work is to provide an engineering environment to design and validate, through model execution, a robotic manipulation scenario as shown in figure 1.

To do so, we propose a tooled approach within a modeling environment dedicated to industrial manipulation scenarios based on Papyrus[1] modeling environment. We provide a Domain Specific Modeling Language (DSML) as an extension of RobotML (Robot Modeling Language) which was initially defined for mobile robotics. Our DSML follows separation of concerns and roles [1], [2] philosophies. It also offers a set of generic actions and components' libraries to guide and help as much as possible the system designer during the design process and until code generation. The variability aspects are handled first through these libraries of generic actions and components which ensure decoupling robotics knowledge from implementational technologies. Besides, we propose to separate concerns related to the different aspects of a robotic system: control, mechanisms, scenario and scene definitions through several dedicated viewpoints. The abstractions (components libraries and generic actions) provided within our DSML have a well-defined semantic so that they are easy to map through code generation to target platforms. Before code generation, these scenarios are validated iteratively by simulation. The simulation is performed using the Papyrus model execution platform: Moka[2].

In this paper, we mainly focus on RobotML extension for manipulation and models validation by simulation.

II. Related Work

Model-Driven Software Development (MDSD) [3] and more specifically Domain Specific Languages (DSL) [4] are now widely used in robotics systems design and development [5]. These latter enable platform independence/isolation, users productivity growth (also easy integration of new team members) and domain experts involvement. In fact, abstractions and notations of DSLs are closely aligned with domain experts terminology. They allow for *very good integration between developers and domain experts: domain experts can easily read, and often write program code, since it is not cluttered with implementation details irrelevant to them* [6]. Most of the existing DSLs for robotics' implementation rely on Model-Driven Engineering (MDE) to define modeling tools proposed in an Integrated Development Environment (IDE). These DSLs

[1] https://eclipse.org/papyrus/ [2] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

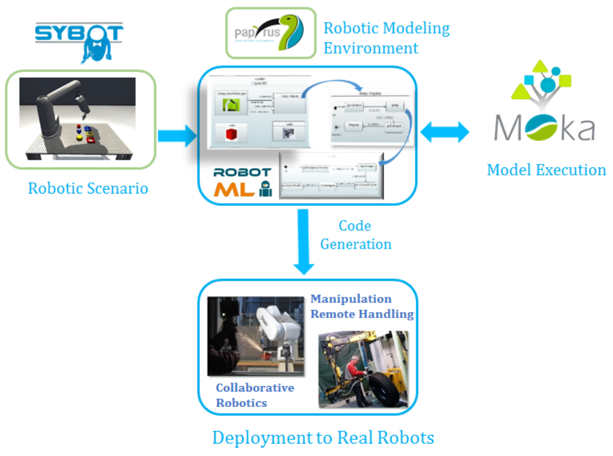

Robotic Modeling Environment
Robotic Scenario
Model Execution
Code Generation
Manipulation Remote Handling
Collaborative Robotics
Deployment to Real Robots

Fig. 1. RobotML for Manipulation and simulation

handle several robotics subdomains (control, communication, perception, architecture, composition, etc.). A complete survey of DSLs for robotics revealed that there is no existing DSL for manipulation [5].

To define our DSL for manipulation, we thought that ideally we could extend an open-source existing DSL whose efficiency have been proven. In the light of the feedback from the literature about best practices of DSL for robotics [5], there are many criteria to take into account while making the choice of defining a DSL for robotics. (1) *Freedom of Choice* or *Freedom from Choice* ? [1] Freedom of Choice consists of supporting many schemes so the DSL users have the responsibility to make their own "right" choices depending on their needs. The risk of such approach is that *there is no guidance with respect to ensuring composability and system level conformance*. In contrast, freedom from choice orientates the user towards the selected features and can guarantee composability and system level conformance. (2) *Separation of Roles* [7]. It is important to define to which user the MDSD tool is dedicated: component builder, end-user, system integrator, robotics community, etc. This can help fixing important objectives and to choose the appropriate abstractions depending on the user background and role. (3) *Separation of Concerns* [2] consists of separating communication, computation, coordination, configuration and composition aspects in the overall software functionality. In [2], the authors consider the 5C's as their most often proven "best practice" in robotics software development because it emerged during their software development experience.

We think that the criteria cited above help to define abstractions but we need to determine the appropriate level of these abstractions. The latters, if too general are unusable and very difficult to map with target platforms for code generation; if too specific, they are not immune from low level details. There are about forty DSLs for robotics [5]. Each of them handles one or several subdomains and follows one or several philosophies among freedom of choice, freedom from choice, separation of concerns and roles.

Some DSLs abstractions rely on domain experts knowledge like RobotML [8] which was defined from a domain ontology [9]. RobotML allows to define architecture, composition, behavior, communication and deployment aspects of a robotic system. It follows separation of concerns and separation of roles philosophies. Composability restricts freedom of choice in order to allow separation of roles between end users and system integrators. Though, RobotML still missing libraries which provide services and API with a well defined semantic so that we know exactly how to implement them or how to map them to existing code. For example, if we specify that a component is a manipulator, we should be able to know its capabilities. If we want to perform a grasping task, we should have a dedicated building blocks to achieve this task.

Our choice has been oriented to RobotML because we participated to its development and we are aware that RobotML abstractions are not sufficient. However, RobotML covers more sub-domains than existing DSLs for robotics [5]. The combination of both freedom of choice and freedom from choice in RobotML allows us to define our own libraries and to give the possibility to users to follow them or to make their own choices.

Let us look more closely to the existing DSLs for robotics and discuss the most known existing approaches. The SmartSoft MDSD called SmartMars [7] have been defined from the knowledge of SmartSoft [10] which is a component-based framework based on communication patterns. SmartMars handle communication and coordination aspects of a robotic system and follows separation of roles and concerns philosophies. MDSD tools support freedom from choice approach which is not suitable to our case because we would like to guide the user without forcing him to completely use our approach.

Unlike RobotML and SmartMars, the BRICS Component Model (BCM) [11] define a very simplified component-based meta-model, with respect to the separation of concerns (the 5C's)[11] and freedom of choice paradigms. However, since the proposed abstractions are highly inspired from the component-based architecture and do not support domain terminology, they remain too general. Therefore, it is difficult

for end users to design their applications because there are no dedicated abstractions to robotics. In this same direction, The 3-View Component Meta-Model (V^{3}CMM) [12] aims to provide designers with an expressive yet simple platform-independent modeling language for component-based application design. V^{3}CMM 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. It is worth noting that although V^{3}CMM has been used mainly in robotics, it does not contain any specificities about this domain.
Consequently, in our view extending RobotML with new abstractions dedicated to manipulation which have well defined semantics is the most suitable solution in our case.

III. RobotML

RobotML [8] is a DSML for mobile robotics which was developed in the context of the PROTEUS[3] (Platform for RObotic modeling and Transformations between End-Users and Scientific communities) project. RobotML development went through several phases: (1) domain analysis, (2) meta-model design, (3) UML profile implementation and (4) the modeling environment development as an extension of Papyrus. RobotML allows the design of mobile robotic systems and their deployment to several middlewares. The domain analysis phase is based on the state of the art of robotic middlewares, simulators and already existing DSML for robotics. This analysis has identified some particular requirements like the independence from the target platforms and the representation of a component-based architecture. The state of the art, combined with the concepts of the domain provided by an ontology of mobile robotics developed in the PROTEUS project, enabled to delimit the field of application (i.e: mobile robotics) and to extract abstractions from the ontology that roboticists need to define their applications. The domain analysis stage led to the definition of RobotML metamodel. RobotML was then implemented as a UML profile (extension of UML metamodel). The graphical modeling environment is available as an open source extension of Papyrus. RobotML addresses separation of concerns through several modeling packages: architecture, control, communication and deployment. It offers also several code generators to robotic middleware.
Lessons learned from RobotML development showed that it is difficult to anticipate a priori abstractions (from the state of the art or ontology). They are often too general so that we do not necessarily know how to implement them. In this work, we went against the top-down approach adopted by RobotML and we focused on specific use cases in industrial manipulation. We performed several code analysis and refactoring to be able to define libraries of components and actions that are at the same time generic and easy to map with a specific code.

[3] http://www.anr-proteus.fr/

IV. RobotML for Manipulation

The main goal of this work is to provide a modeling environment for industrial manipulators which will ease their understanding, design and validation with respect to separation of concerns and roles. RobotML for manipulation development lifecycle is composed of the following iterative steps as shown in Figure 2:

1) Analyzing use cases
2) Extending RobotML meta-model
3) Adding new sensing and actions libraries to RobotML
4) Implementing the modeling tools

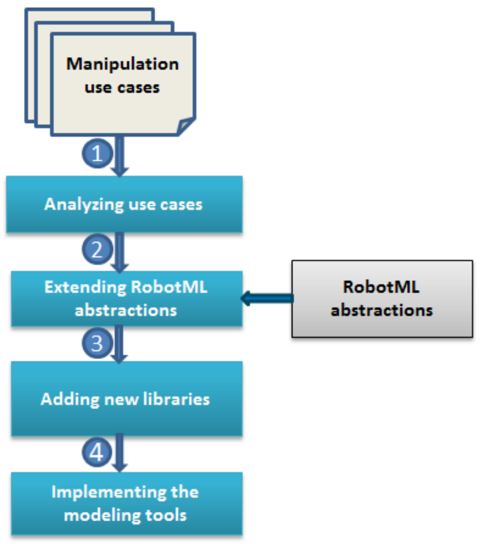


Fig. 2. RobotML for manipulation: development lifecycle

A. Analyzing use cases

The first step of our approach consisted of analyzing use cases for several manipulation tasks like grasping and trajectory following, in order to identify appropriate abstractions specific to manipulation which are configurable and platform-independent so that they hide variability aspects and can be easily reused. Generally speaking, manipulation tasks are achieved by mechanisms which have their own properties (like Degrees of Freedom and reference frames). A mechanism can be poly-articulated and composed of a manipulator, a tool (e.g: gripper, disk grinder, drill, sander, welder) and/or a mobile base. Each of them has specific properties. For instance, a gripper has grasping and actuating force properties. Each mechanism has also a set of actions that it is able to perform. These actions are provided by controllers. Controllers could have several architectures but gather the sense, plan, act paradigms [13]. If we look more closely into controllers architectures, we can roughly distinguish five kinds of components: data acquisition, data treatment, trajectory generators, control/command and drivers. Each of these components has its own configuration for instance periodicity, inputs and outputs.
Mechanisms are situated in a scene containing protagonists

and fixed or mobile objects. Objects and mechanisms interact together to perform the scenario actions.
In each manipulation scenario, the controllers and mechanisms could be generalized if we choose their appropriate associated properties for configuration. Let us now take a look on manipulation tasks. In this paper, we will not make an analysis of all manipulation tasks but we will only discuss a simple use case to illustrate our approach and show how we can identify abstractions. So, we consider a trajectory following task.
In this task, the robot has a target position to reach. This task can be decomposed into three main components that are responsible of trajectory generation, motion control and state observation. The trajectory generator takes the target position as input and generates real-time position instructions sent to motion control. The state observer sends also measures extracted from the robot to the motion control. The latter adjusts the gap between measurements and instructions and directly acts on the physical actuators. If we focus on these components separately, we can find several variabilities. For instance, the trajectory generator component could take into account the state of the robot in its environment. It may include path planning and obstacle detection features. That changes the behavior of the trajectory generator because if it takes into account the perceived information from the environment, the trajectory to follow is not static and is generated dynamically by the path planner. Regarding the motion control component, the variability is mainly algorithmic. The role of this component still unchanged but depending on the used algorithms (e.g:linear control, non-linear control), its performance may change. In all these cases, we can simply define a trajectory following as a *go To Position* function which takes as input its target position and encapsulates variabilities of trajectory generation and motion control. This analysis permit us to define three categories of abstractions:

- Components abstractions which categorizes families of configurable components and encapsulate specific properties while preserving properties of interest.
- Algorithmic abstractions [14] which are linked with the task to develop. They can encapsulate a set of instructions to perform a sub-task or details related to an algorithm variants.
- Non-algorithmic abstractions [14] which are either queries that return information about the environment (e.g: obstacle detection) including the robot itself or high-level actions that the robot can/must perform (grasp, go to position, etc.).

In our view, an abstraction allows to describe a container or a concept with a well-defined semantic so that we know how to implement it after configuring it. Actually, all the elements that play a role in the robot scenario can be abstracted, even existing objects in the robot's environment. These abstractions are platform-independent because they encapsulate specific properties and preserve properties of interest with domain-dedicated notations. Platform-independence and genericity ensure reusability not only at the design level but also they allow code generation to several heterogeneous platforms. Non-algorithmic abstractions are abstractions that can be capitalized because they are independent of the algorithmic details of a task. However, algorithmic abstractions may depend on non-algorithmic abstractions but remain specific to the task to develop.

B. *Extending RobotML*

As mentioned previously in sectionIII, RobotML existing abstractions are aiming at modeling mobile robots and not specifically industrial manipulators. In addition, they do not provide libraries to manipulation, perception and action.
RobotML metaclasses are related to software, hardware, environment, systems (i.e: components), communication between them and behavior (state machines). We have chosen to cluster new manipulation abstractions as specializations of these existing meta-classes.
We introduced the `Mechanism` metaclass as a specialization of the RobotML `Hardware` metaclass illustrated by Figure 3. A mechanism can be a `Tool`, a `MobileBase` or a `Manipulator`. All of them inherit the `Mechanism`'s metaclass properties like DoF, Position, etc. For instance, *Position* is a data type composed of x, y and z coordinates relative to the reference *Frame*.

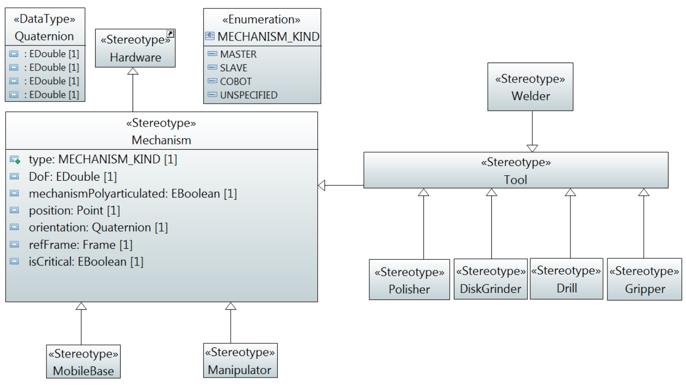


Fig. 3. Extract of mechanisms metaclasses

Figure 4 indicates that the RobotML `Software` metaclass has been extended with a generalization of control components. Each of them has configurable properties like periodicity and provides/receives buffered synchronous/asynchronous data.

Regading the scene part, we defined the `Object` metaclass as extension of the RobotML `PhysicalObject` metaclass (Figure 5). It represents elements situated in the robot's scene and sets up their properties. For instance, the approach position of an object is an important information for grasping.

C. *Adding new libraries*

In addition to metaclasses which describe components, we defined also algorithmic and non-algorithmic abstractions that we included as libraries. Figure 6 shows a set of interfaces which propose actions associated to each mechanism depending on its capabilities. This is useful to establish qualitative

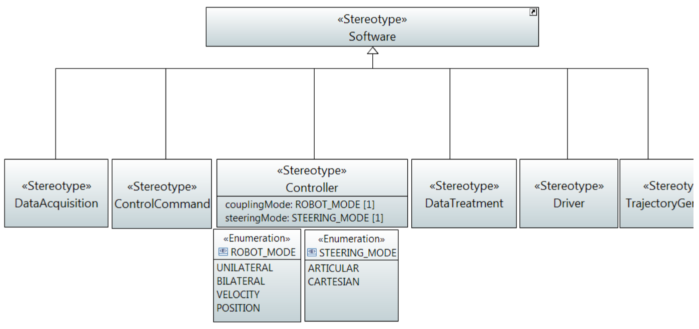


Fig. 4. Extract of control metaclasses

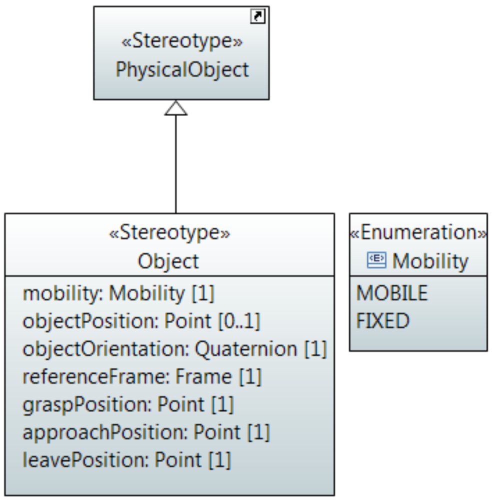


Fig. 5. Object metaclass

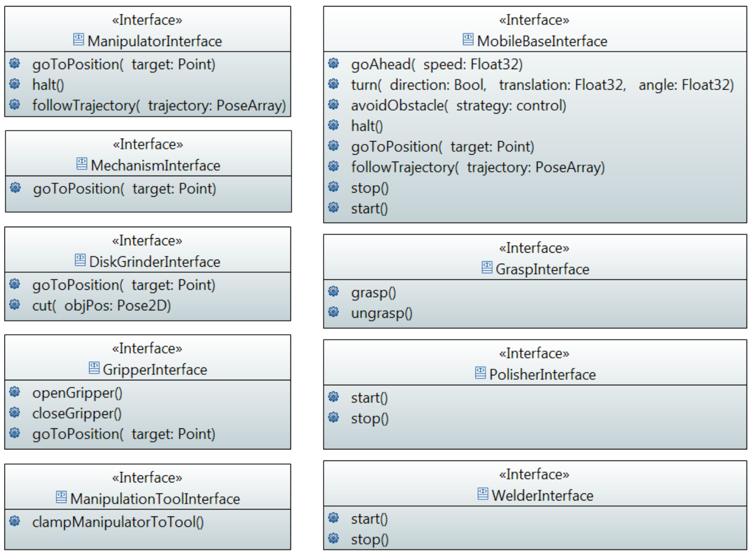


Fig. 6. Extract of algorithmic and non-algorithmic libraries

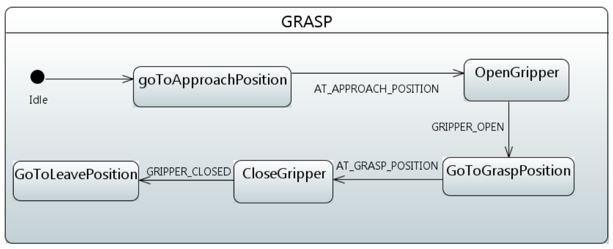


Fig. 7. Extract of algorithmic library: Grasp state machine

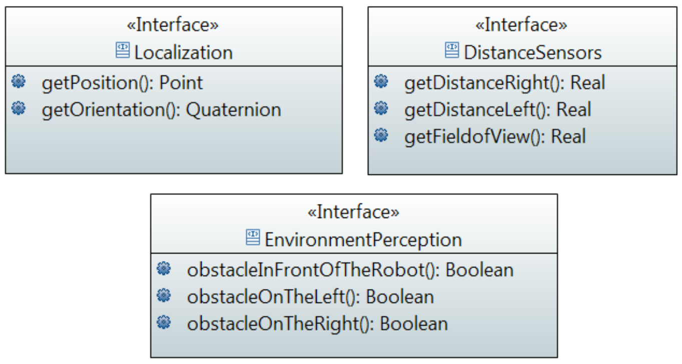


Fig. 8. Extract of non-algorithmic library

verification so that we anticipate what are the capabilities of the robot. Some capabilities could also result from the combination of several mechanisms. For example, a manipulator can only go to a position among all reachable positions around it; a manipulator with a mobile base can go to a position situated among all reachable positions of the mobile base and the manipulator; a manipulator with a gripper and a mobile base can grasp an object in a position among reachable positions of the mobile base and the manipulator.

Figure 7 shows an algorithmic abstraction represented with a state machine corresponding to the grasping task. In this state machine, states correspond to manipulation actions. Transitions are triggered by events at each end of state (at_position, gripper_open, etc.).

As mentioned previously, in addition to mechanisms actions, non-algorithmic abstractions correspond also to the information that we can extract from the environment through sensors. These abstractions are grouped within the sensors categories. For instance (Figure 8), `DistanceSensors` can be associated to infrared, laser or sonars sensors; `Localization` to GPS or odometers and `EnvironmentPerception` to the queries we can send to both contact or distance sensors.

D. Implementing the modeling tools

The RobotML for manipulation graphical editor has been implemented as an extension of Papyrus. We defined several viewpoints where we can drag and drop components associated to the different aspects of the system:

- The mechanisms definition and architecture diagrams allow the configuration, communication and composition of mechanisms. In the configuration diagram, each mechanism has automatically associated ports typed with interfaces which propose a set of actions that it is able to perform.

- The controller definition and architecture diagrams allow the definition of configuration, communication, computation and composition parts of control components. Each of them when dropped into the diagram, has automatically associated data input and output ports.
- The scenario definition diagram ensures the coordination between components. It allows the definition of hierarchical state machines where states correspond to the actions of mechanisms which are sent through commands to the controller depending on conditions on the perceived data handled by the controller. We can define several scenarios which communicate with the controller.
- The scene definition diagram gather all the elements that constitute a scene (i.e: objects, scenario, controller and mechanisms) and allows their composition.

Through this distribution of diagrams we answer the need of: (1) separation of concerns because these diagrams allow composition, configuration, coordination, computation and communication. (2) separation of roles because each user can focus on the part of the system with regards to his needs and expertise.

V. Simulation with Moka

Papyrus provides UML models execution by means of a module called Moka which aims at providing a generic environment for model execution [15]. Moka natively includes an execution engine complying with fUML (Foundational UML) , by implementing the interpreter described in this specification. This implementation supports the execution of UML subset defined in fUML specification[4]. It basically provides execution of active classes behaviors, represented by UML activities. Moka also supports PSCS[5], an interesting basis to ensure propagation of data and control between a set of connected components in a model. Finally Moka integrated recently a support for PSSM (Precise Semantics of UML State Machines)[6] (Figure 9). In this paper, we took advantage of PSSM support in Moka to validate manipulation scenarios.

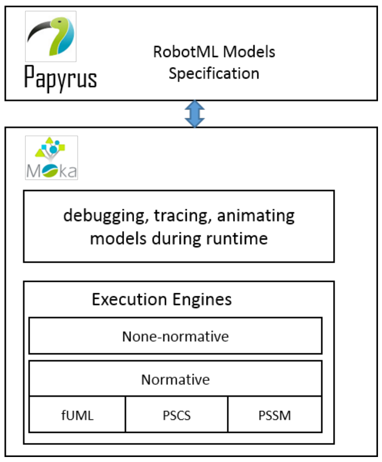

Papyrus
RobotML Models Specification
Moka
debugging, tracing, animating models during runtime
Execution Engines
None-normative
Normative
fUML
PSCS
PSSM

Fig. 9. Moka: Execution Engine for RobotML and UML models

VI. Simulation of a use case Scenario with a Sybot

Our use case was realized based on the Sybot[7] robot behavior. The Sybot is available in the RIF (Robotic Innovation Facilities) Paris-Saclay created in the frame of the ECHORD++ european project[8].

A. The Sybot robot

The Sybot is a collaborative robot arm which has two main modes (see Figure 10): a learning mode where it directly collaborates with the operator to learn how to perform a task. In this mode, the robot records all the information about the trajectory that must be followed and the actions that must be performed. The second mode is an automatic mode which is triggered by the operator at the end of the first mode. When the robot is in this mode it follows the recorded trajectory and performs the recorded actions. When a human operator collides with the robot arm, it stops its current task. We would like to simulate this behavior of the automatic mode to observe how the Sybot will behave in different situations.

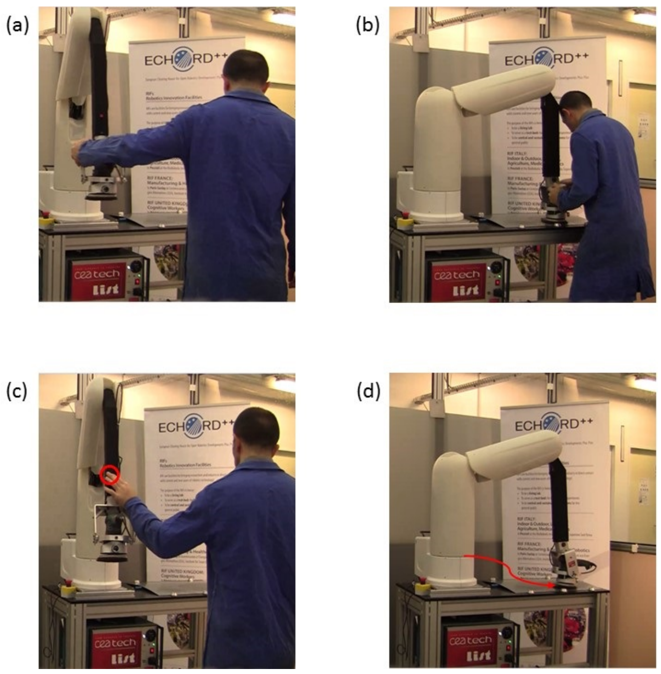


Fig. 10. (a) learning mode: the operator teaches the Sybot the trajectory to follow (b) learning mode: the operator performs the movement so that the Sybot records it (c) the operator puts the arm in its initial position and triggers the automatic mode (d) automatic mode: the Sybot follows the trajectory and performs the movement automatically

B. The Sybot automatic mode simulation

Within our use case, we are interested on the simulation of the automatic mode of the Sybot. To do so, we consider

[4] http://www.omg.org/spec/FUML/1.2.1/
[5] http://www.omg.org/spec/PSCS/1.0/ [6] http://doc.omg.org/ad/2015-3-2
[7] http://sybot-industries.com/ [8] http://echord.eu/the-paris-saclay-rif/

a scenario where the robot behavior is described in a state machine in terms of states (actions) and transitions triggered by conditions on the environment information. The actions are sent from the scenario to the controller which executes them. Information about the environment are hidden in a perception component which is contained in the controller. In this case, we will not focus on environment perception and data computation. They are handled by components dedicated to data acquisition and treatment inside the controller. We only consider the information *obstacle detected* when an obstacle is detected but the way it is done is not our focus at the moment. Our concern is to simulate the behavior of the controller and its reaction to the commands sent by the scenario. For this reason, we have chosen several possible behaviors of the controller to see how the scenario handles them.

When the Sybot executes its trajectory following (or *goToPosition* function) in the automatic mode, if an obstacle is detected, it stops. The architecture of our simulation example is given by Figure 11. The scenario sends commands to the controller through signals. The controller executes these commands and sends back the results to the scenario.

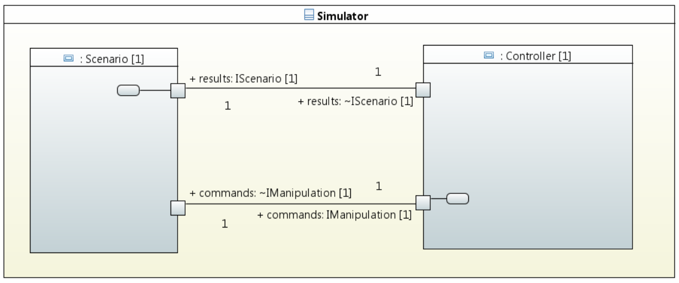

Simulator
: Scenario [1]
: Controller [1]
+ results: IScenario [1]
1
1
+ results: ~IScenario [1]
+ commands: ~IManipulation [1]
1
1
+ commands: IManipulation [1]

Fig. 11. The simulation architecture

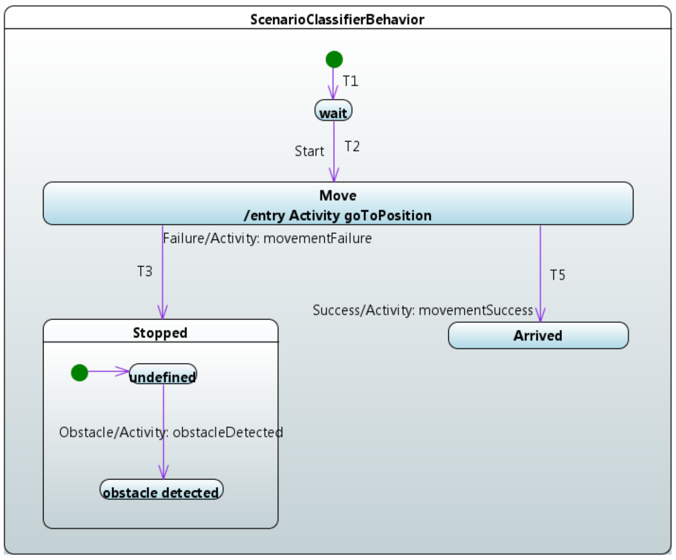

ScenarioClassifierBehavior
T1
wait
Start
T2
Move
/entry Activity goToPosition
Failure/Activity: movementFailure
T3
T5
Success/Activity: movementSuccess
Stopped
Arrived
undefined
Obstacle/Activity: obstacleDetected
obstacle detected

Fig. 12. Sybot scenario: automatic mode

As mentioned previously, the scenario is described with a state machine (see Figure 12) where states correspond to the actions that the mechanism must perform. In this scenario, the robot first waits for the trigger event activated by the operator (signal *start*). Then, it starts moving to the recorded target position. If it receives from the controller the signal *movementFailure* it switches to the state `Stopped`. In this state, we only know that the robot cannot move (state `undefined`) unless the signal *obstacleDetected* is received. If the robot receives a *movementSuccess* signal while moving, it switches to the state `Arrived`.

The commands signals sent by the scenario are handled by the controller (see Figure 13). The controller accepts the signal *Move* and sends the coordinates of the target point to the function *goToPosition*. The latter after its execution sends *Failure* or *Success* signals. We mentioned previously that we would like to simulate different behaviors of the controller. The variation point in our case is the *goToPosition* function implementation which is realized with Alf (Action Language For Foundational UML) [9].

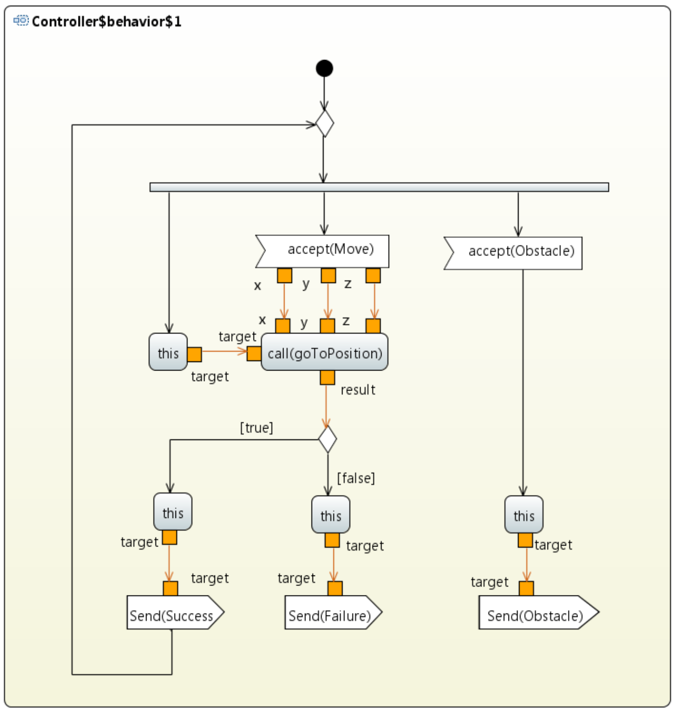
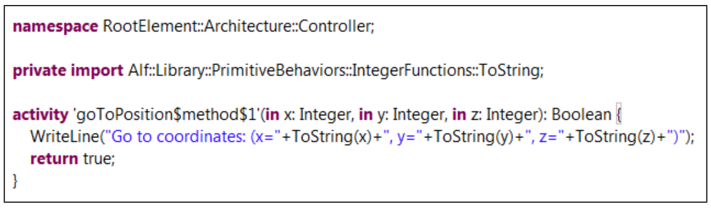


Fig. 13. The controller behavior diagram

Figures 14 and 15 show two implementations of the *goToPosition* method. The first one simulates that the function terminates correctly by returning true. The second one simulates that the robot encounters an obstacle on its trajectory. In this implementation, the controller sends the signal *obstacle* to itself. This signal triggers the action `accept(obstacle)` in the controller behavior as already shown in Figure 13. This action leads the controller to generate a signal *obstacle* which will be sent to the scenario. The latter while receiving

[9] http://www.omg.org/spec/ALF/

this event, switches from the state `undefined` to the state `obstacleDetected`.

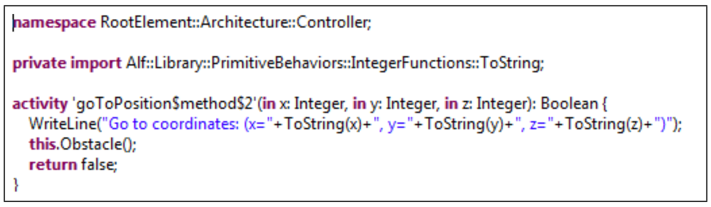


```
namespace RootElement::Architecture::Controller;

private import Alf::Library::PrimitiveBehaviors::IntegerFunctions::ToString;

activity 'goToPosition$method$1'(in x: Integer, in y: Integer, in z: Integer): Boolean {
   WriteLine("Go to coordinates: (x="+ToString(x)+", y="+ToString(y)+", z="+ToString(z)+")");
   return true;
}
```

Fig. 14. *goToPosition* ALF implementation: first case



```
namespace RootElement::Architecture::Controller;

private import Alf::Library::PrimitiveBehaviors::IntegerFunctions::ToString;

activity 'goToPosition$method$2'(in x: Integer, in y: Integer, in z: Integer): Boolean {
   WriteLine("Go to coordinates: (x="+ToString(x)+", y="+ToString(y)+", z="+ToString(z)+")");
   this.Obstacle();
   return false;
}
```

Fig. 15. *goToPosition* ALF implementation: second case

The results of both simulations have shown that the scenario switches correctly between states with regards to the signals sent to and received from the controller.

VII. Conclusion

Best practices in the literature have defined directions to follow if we want to define an easy-to-use DSL which addresses the challenges identified in robotics systems development. This paper introduces an extension of RobotML for industrial manipulators to design and simulate manipulation scenarios. The design phase allows the modeling of several aspects of a configurable robotic system: mechanisms, controller, scenario and scene. Sensing and actuation libraries help to deduce the capabilities of the modeled mechanisms and to guide as much as possible the user during design time. We also presented an example of model simulation before application deployment using Moka which allowed to simulate and validate the controller behavior. Simulation with Moka was basically dedicated to class diagrams, this work constituted a first step in model execution within composite diagrams. Several code generators have been developed within our DSML to deploy the designed system to several frameworks like OPC-UA [16] and OROCOS-RTT. This was not the focus of this paper but it is important to mention that we are able to generate components behavior because of the use of pre-compiled libraries and easy to map functions.
Several perspectives are considered for this work. First of all, we intend to enrich our DSML with control architectures and configurable building blocks for robotics tasks in the context of P-RC2 [10] project. Having a complete control architecture will allow us to simulate the behavior of the whole controller. It would be also interesting to map the simulation scenario with a real robot so that we can visualize the execution in real-time. This is useful in the case where we need to record the state of the robot and the production cell in a specific context.

[10] http://www.p-rc2.com/

References

[1] M. Lutz, D. Stampfer, A. Lotz, and C. Schlegel, "Service robot control architectures for flexible and robust real-world task execution: Best practices and patterns," in *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 2014, pp. 1295–1306.

[2] D. Vanthienen and H. Bruyninckx, "The 5C -based architectural Composition Pattern : lessons learned from re-developing the iTaSC framework for constraint-based robot programming," *Journal of Software Engineering for Robotics*, vol. 5, no. May, pp. 17–35, 2014.

[3] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[4] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000.

[5] A. Nordmann, N. Hochgeschwender, and S. Wrede, *Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20-23, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. A Survey on Domain-Specific Languages in Robotics, pp. 195–206.

[6] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[7] C. Schlegel, A. Steck, and A. Lotz, "Model-driven software development in robotics: Communication patterns as key for a robotics component model," *Introduction to Modern Robotics*, 2012.

[8] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds., 2012, vol. 7628, pp. 149–160.

[9] S. Dhouib, N. Du Lac, J.-L. Farges, S. Gerard, M. Hemaissia-Jeannin, J. Lahera-Perez, S. Millet, B. Patin, and S. Stinckwich, "Control architecture concepts and properties of an ontology devoted to exchanges in mobile robotics," in *6th National Conference on Control Architectures of Robots*, 2011.

[10] C. Schlegel, "Communication Patterns as Key Towards Component-Based Robotics," *International Journal of Advanced Robotic Systems*, no. 1, pp. 49–54, Mar. 2006.

[11] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The brics component model: A model-based development paradigm for complex robotics software systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1758–1764.

[12] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, "V^{3}CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010.

[13] D. Kortenkamp and R. G. Simmons, "Robotic systems architectures and programming," in *Springer Handbook of Robotics*, 2008, pp. 187–206.

[14] S. Kchir, T. Ziadi, M. Ziane, and S. Stinckwich, "A Top-Down Approach to Managing Variability in Robotics Algorithms." in *Fourth International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2013)*. Tokyo, Japan: DSLRob13, Nov. 2013, p. 6. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00995131

[15] G. Sahar, T. Jérémie, C. Arnaud, D. Saadia, and G. Sébastien, "Executable modeling with fuml and alf in papyrus: Tooling and experiments," in *1st International Workshop on Executable Modeling*, Nov. 2015.

[16] S. Azaiez, M. Boc, L. Cudennec, M. Da Silva Simoes, J. Haupert, S. Kchir, X. Klinge, W. Labidi, K. Nahhal, J. Pfrommer, M. Schleipen, C. Schulz, and T. Tortech, "Towards flexibility in future industrial manufacturing:a global framework for self-organization of production cells," in *to appear in the proceedings of the 2nd International Workshop on Recent Advances on Machine-to-Machine Communication (RAMCOM 2016)*. Elsevier Science, 2016.