Journal of Software Engineering for Robotics 7(1), July 2016, 20-35
ISSN: 2035-3928

A Toolset to Address Variability in Mobile Robotics

Tewfik Ziadi[1] Jean-Loup Farges[2] Serge Stinckwich[3,4,5] Mikal Ziane[1,6] Saadia Dhouib[7]
François Marmoiton[8] Nicolas Morette[9] Cyril Novales[9] Selma Kchir[10] Bruno Patin[11]

[1] Sorbonne Universités UPMC, Univ. Paris 06, UMR 7606, LIP6, F-75005, Paris, France
[2] ONERA, DCSD, BP 4025 - F-31055 Toulouse Cedex, France
[3] Université de Caen Normandie, Caen, France
[4] Sorbonne Universités UPMC, Univ. Paris 06, UMI 209, UMMISCO, F-75005, Paris, France
[5] IRD, UMI 209, UMMISCO, IRD France Nord, F-93143, Bondy, France
[6] Sorbonne Paris Cité, Université Paris Descartes, Paris, France
[7] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 94, Gif-sur-Yvette, F-91191 France
[8] Clermont University, Pascal Institute, UMR 6602 du CNRS, F-63171 Aubière, France
[9] PRISME, UPRES 4229, Université d'Orléans, France
[10] CEA-LIST Institute, Interactive Robotics Laboratory, CEA Saclay – PC 178 - Digiteo MOULON F-91191 Gif-sur-Yvette Cedex, France
[11] Dassault Aviation Research and future business, Future aircraft system - 78, Quai Marcel Dassault - Cedex 300 - 92552 Saint Cloud Cedex, France

Abstract—Dealing with variability may be the most serious issue when developing and maintaining robotic systems: it jeopardizes communication between the stakeholders of a robotic system and interoperability between its components. This paper reports on an approach and a toolset to improve both communication and interoperability. The approach relies on an ontology of mobile robotics and on a *Domain-Specific Language* (DSL) to describe robotic systems with missing parts that are problems that can be completed by third-party solutions. The domain model of the DSL is derived from the conceptual framework of the ontology. Models of robotic systems and scenarios can be defined using a graphical modeling environment and code can be generated for open-source as well as commercial middleware and simulators. A key advantage of the approach is that code can be generated for both simulation platforms and real-world robots. The validation of the approach is illustrated with a landmark-search case study.

Index Terms—Ontology, Robotics, Domain Specific Languages, Code Generation, Middleware, Simulation

1 Introduction

Dealing with variability may be the most serious issue when developing and maintaining mobile robotic systems. Not only can robots be made of very different kinds of sensors and actuators but their missions, environments and software architectures may also be quite varied. Consequently, meaningful communication between the stakeholders of a robotic system and interoperability between its components are quite difficult to achieve.

Robotic middleware solutions have been proposed to provide some interoperability but only at a rather low level of abstraction. Several norms and standards have also been established that provide some common definitions but only for specific domains.

Regular paper – *Manuscript received September 02, 2015; revised June 12, 2016.*

- *This work was supported by the French research funding agency ANR (Agence Nationale de la Recherche) from 2009 to 2013.*

What is missing is a common framework to specify robotic systems using high-level abstractions, including their structures, behaviors and communication mechanisms. These abstractions must bear several potentially conflicting qualities. First, to improve communication, they must fit or establish the vocabulary of mobile robotics. Second, to ease maintenance and interoperability, they should be as independent as possible from low-level details including those tied to specific middleware solutions. At the same time, these abstractions must still have clear operational semantics despite their distance from low-level details.

Reconciling these potentially conflicting qualities is a very hard open problem. One possible approach consists in using an ontology for grounding one or several *Domain Specific Languages* (DSLs). Ontologies have become popular to address both communication and interoperability problems through a common set of abstractions. They are however not suited to produce executable systems. DSLs on the other hand have become popular to both give a syntax and a clear operational semantic to a set of concepts of a particular domain.

The French PROTEUS project (Robotic platform to facilitate transfer between industry and academics) which lasted from 2009 to 2013, gathered several French industrial and academic teams to propose a common framework for mobile robotics. This framework experiments the approach discussed above where an ontology is defined for grounding a DSL for mobile robotics. In order to make variability issues manageable it was decided that the case studies should focus on a few, possibly a single, well-specified issues. In order to foster emulation the case studies would provide an environment and a robotic system with one or several ”holes” that had to be filled by external providers. The framework provided by the PROTEUS project is based on the following steps:

- Defining an ontology gathering a set of high-level definitions, interfaces and assumptions that cluster the knowledge provided by users working in the field of robotics in a handful of semantic notions.
- Grounding a DSL, called RobotML, that allows specifying robotics systems with ”holes” (variation points that have to be filled).
- Implementing a set of transformations for code generation from the DSL to several execution or simulation platforms.

One of the innovation issues proposed by the PROTEUS project was to consider realistic case studies as a core resource to define the proposed framework. As will be presented in Section 5, six mobile-robotics case studies were considered. Knowledge was elicited by asking case-studies providers to describe their case study with enough details: high-level objectives, mission, low-level objectives, environment, a priori knowledge, actions, experimental site, hardware and software architectures of the vehicles, simulator, middleware and evaluation metrics. This knowledge is the basis for the development of the ontology.

The DSL was built based on the knowledge elicited by the ontology. The ontology is thus used as input for domain analysis: the first step to making a DSL. As will be discussed below, the use of an ontology helps defining a DSL whose concepts are close to the end-users. However, grounding the DSL with the ontology proved quite difficult to achieve which is not very surprising as ontologies describes static entities while our DSL is meant to describe executable systems.

The PROTEUS toolset provides a set of code generators for different execution or simulation platforms. Several targets were considered including many robots (ReSSAC, WifiBot, and R-Trooper), their middleware (RT-MAPS, OROCOS-RTT, Effibox, and partly ROS) and associated environment simulators (CycabTK and Blender Morse)

The different code generators were implemented using a transformational approach [1], where each DSL model is translated into a robotics platform for which an execution engine already exists.

The structure of the paper is as follows. Section 2 describes the ontology. This DSL is presented in Section 3. The RobotML toolset is presented in Section 4.

The next section presents the case studies whose tests aim at validating the ontology (see section 5.1), the modeling using RobotML (see section 5.2), the code generation capability (see section 5.3) and the fact that variability is actually addressed by the toolset (see section 5.4).

Section 6 presents related work, Section 7 discusses some limitations and the lessons that were learned and Section 8 concludes.

2 An ontology to ground a robotic DSL

The development of a DSL first requires gathering knowledge about the addressed domain in what is referred as the *domain analysis* phase [2]. Many notations and formalisms can be used for domain analysis. In the PROTEUS Project, we relied on ontologies for the following reasons:

- Ontologies are a mature technology with reliable tools and well-defined languages and standards that interoperate with databases and web technologies.
- Tools (reasoners such as Pellet [3], FaCT++ [4] and HermiT [5]) can be used to continuously check the overall correctness of the ontology.

In next sections, we will use the Ontology Web Language (OWL) [6] to describe the PROTEUS Ontology. The main OWL concepts are the following:

- *Classes* define groups of individuals that are together because they share some properties. Classes are often organized in specialization hierarchies. The most general class is named `Thing`.
- *Properties* state relationships about individuals.
- *Individuals* are instances of classes.
- *Axioms* are the logical expressions of the ontology.

Classes, properties and axioms are gathered in *packages*.

The ontology was defined iteratively based on the analysis of the description of the different case studies. An additional design choice was to limit the expressiveness of the ontology to Description Logics (DL). Indeed, without this limitation the tools for checking the consistency of the ontology are inefficient.

Initially, we considered the possibility to let the ontology emerge from use and this would probably help a lot to focus it on the entities that are really useful to the stakeholders of the case studies. Unfortunately, this idea was quickly dropped as using the ontology required that the DSL was operational, at least to build executable code. Moreover, deriving the ontology directly from the needs of the case studies could lead to concepts that are not general enough. It was thus decided to adopt a less iterative approach but rather include different stakeholders in regular meetings.

2.1 Overall structure of the PROTEUS Ontology

An additional design choice was to structure the PROTEUS Ontology as a set of packages. Indeed this choice facilitates collaborative development and maintenance. The division in packages is primarily done by analyzing generic terms for the

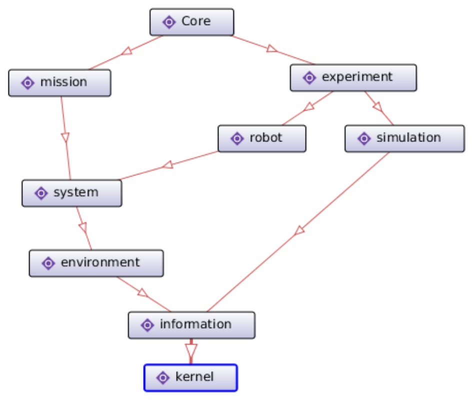

Fig. 1. Import relations of the core ontology packages

considered case studies. In addition and as we will show, more generic packages are also introduced to facilitate collaborative development and maintenance.

The identified packages are split into two parts:

- A stable part, that corresponds to the classes, properties and axioms that represent the main elements to specify mobile robotics applications. This part is defined in the *core* package.
- A part that can be refined and enriched by users bringing description of their case study using individuals.

With this design choice case study providers can feed the ontology with new individuals just after the definition of the associated classes by ontology developers. Consequently the validation, with the reasoner, of the ontology against case studies was conducted in parallel with its design. This is important because a complete consistency checking requires classes to have individuals.

The *core* package imports eight packages as illustrated in Figure 1. The ontology does not rely on external packages such as a common sense ontology like for instance Cyc [7], or an upper ontology, as the Suggested Upper Merged Ontology [8]. Indeed, importing an external ontology implies to comply with its language and to use the associated editing and reasoning tools. Moreover, only a small part of the imported content could be relevant for the modeled domain but the reasoner efficiency, in terms of computation time, would be strongly impacted by the total size of imported content. The *information* and *kernel* packages are examples of generic packages that provide useful upper concepts, while, *simulation*, *environment*, *system*, *robot* and *mission* packages gather specific concepts that are extracted from the description of the different case studies.

Categorization criteria were used to define classes in each package: the relevance of their names for case studies owners or DSL developers, the possibility to characterize them by specific properties, disjunctions or points of view and the expectation of individuals for them. However, a consensus was

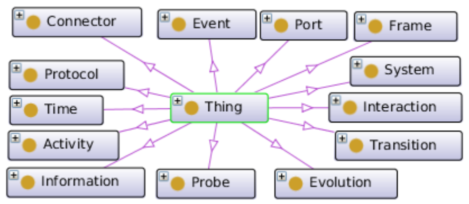

Fig. 2. Part of the taxonomy of the kernel package - Arcs correspond to the has subclass relation

also sought among the case study holders.

2.2 The PROTEUS ontology packages

The **Kernel Package** includes 55 classes and 73 properties. Figure 2 presents the basic classes that inherit directly from the `Thing` root class.

Class `System` represents something that has a boundary and that distinguishes what is inside itself from what is outside and thus possibly interacts with it. This class is disjoint with all other classes presented in Figure 2 except with Class `Probe`. There are two kinds of systems that are defined in the ontology as disjoint subclasses of `System`: `Composite System` and `Atomic Systems`. A property, *aggregates*, whose domain is `CompositeSystem` and range `System` provides the semantics for this disjunction. In addition, the `System` Class is also specialized by three types of systems: `Physical Object`, `Software` and `Model`.

An important design choice was to model robots as `Composite Systems` but not necessary as agents. Indeed, from our point of view, only an autonomous robot is also an agent. This implies defining a more restrictive concept of an agent than the one of classical upper ontologies. For this, a specific kind of `Interaction`, the `Action` is defined and an axiom:

Action(?y) PhysicalObject(?x) triggers(?x, ?y) $\rightarrow$ *Agent(?x)*

indicates that any physical object that triggers an action is an agent.

Because robots are dynamical systems, the Classes `Event`, `Evolution` / `System Dynamics`, `Protocol`, `Transition`, `Activity` and `Time` are also introduced. The semantic for `Evolution / System Dynamics` is given by a property, *animates*, whose domain is `Evolution` / `System Dynamics` and range `System`. Finally, the need of validation and locating physical objects in the robotic systems respectively induces the `Probe` and the `Frame` Classes.

Systems are connected together using the concepts of `Port` and `Connector`. These concepts are usually found in robotics middleware and, even if they are not relevant for the case study owners, they are very relevant for DSL developers. In addition, the `Interaction` and `Information` Classes are defined to allow interactions between the different systems. These interactions are typically exchanges between software or hardware components. Two important properties provide semantics for those exchanges: on the one hand *triggers* with domain `System` and range `Interaction` and on the other hand *impacts* with domain `Interaction` and range `System`. The `Information` Class is specialized in `Data`, i.e. formated information, and `Abstraction`, i.e. meaning.

The **Information** Package defines 120 specializations of the `Data` and `Abstraction` Classes presented above. While specializing, 22 additional properties are defined between data, between abstractions and between data and abstractions. For instance, the property *is coded in(?x, ?y)* is defined between an `Abstraction` and a `Data`. This property is also used between specializations of `Abstraction` and `Data`. For instance, it is used between `Number`, i.e. a number in the mathematical sense of the term, and a `Primitive Data`, i.e. a data type of a computer language, or between an `Algorithm` and a `Source Code`.

The **Environment Package** gathers 24 classes and 17 properties that are used for modeling the environment in which robots move. For instance, `Atmosphere`, `Water body` and `Building` Classes are introduced. In addition, different types of `Surfaces` are defined, for example `Land`, `Floor` and `Water` surfaces. Different kinds of `Forces` are also characterized. This package imports the *information* Package because the description of the environment need specific abstractions.

The **System Package** contains 151 classes and 28 properties used to model specific systems. `Robotics Systems` aggregate `Drivers` and `Hardware` and are protected by `Security Systems` and controlled by `Control Systems`. Four basic specializations of `Robotic System` are modeled: `Sensor System`, `Power System`, `Communication System`, and `Actuator System`. The corresponding specializations of `Driver` and `Hardware` are also defined. These classes are further specialized.

The **Robot Package** includes 15 classes and 7 properties. `Robots` aggregate `Sensor Systems`, `Actuator Systems`, `Power Systems` and `Communication Systems` and are classified following autonomy and mobility points of views. Concerning autonomy, on one hand there are `Unmanned Systems` and on the other hand `Piloted Systems`. Concerning mobility the categorization was performed with respect to classes of the *environment* Package.

The **Mission Package** models the `Mission` performed by robots using 28 classes and 28 properties. `Mission objectives` can be formulated as `Tasks` that can be decomposed in other tasks until reaching atomic tasks called `Operators`. A `Plan`, with instances of operators, is built by a `Planning System` and executed by an `Executive System`. These systems can be included in a `Mission Management System`. The *Mission* Package imports the *System* Package to ground these three systems. Classes are also defined in this package for communication in multi-robots systems and objectives fulfilled by a behavior specified with a `State Machine`.

The **Simulation Package** contains 34 classes and 37 properties that are used for modeling knowledge about simulation. Simulators are categorized in `Hardware Simulators`, `Software Simulators` and `Hybrid Simulators`. Software simulators animate `Simulated Systems` and have `Software probes` for computation of `Metrics`. Animated systems can correspond to `Physical Objects` and to other objects such as a scene camera. `Physics Engine` and `Render Engine` are also defined. The package imports the *information* Package because several types of sets and data are specific to simulation.

The **Experiment Package** contains 18 classes and 14 properties that are used to model people performing experiments. Some of those people, `Problem Providers`, define `Problems` and evaluate `Solutions` to those problems using `Metrics`. Those solutions are designed by other people: `Solution Providers`. The modeling of people is grounded on `Problem` and `Solution` classes. A `Solution` aggregates `Software Modules` and a `Problem` aggregates at least a `Robot` and a `Simulator`. That is the main reason for importing the *Robot* Package and the *Simulation* Package.

3 RobotML

The PROTEUS Ontology provides a vocabulary and a taxonomy that, together, forms a conceptual framework for defining the DSL abstract syntax, called *RobotML domain model*. This abstract syntax was implemented as a UML profile, i.e. an extension of the UML Meta-Model. Then, we developed the graphical concrete syntax as an extension of the Papyrus modeling tool[1].

3.1 From the PROTEUS Ontology to the RobotML domain Model

The concepts of the PROTEUS Ontology are mapped as meta-classes of the Robot domain Model. Among the 87 meta-classes of the RobotML domain Model, 60 are directly obtained from the ontology. This proportion of 69% (60/87), indicates that the definition of the RobotML domain Model is strongly oriented by the ontology. However, 385 concepts among 445 (87%) from the PROTEUS Ontology are not present in the RobotML domain Model. This mainly includes ontological concepts from the three packages *information*, *simulation*, *experiment* and *mission*. Three principal points can justify this:

- The concepts from the *information* package that concern data are ignored. Indeed UML already has `Primitive DataType` and `Structured DataType` concepts. On this basis, a library of ROS data message structures is made available to the users of RobotML.

1. http://www.eclipse.org/papyrus

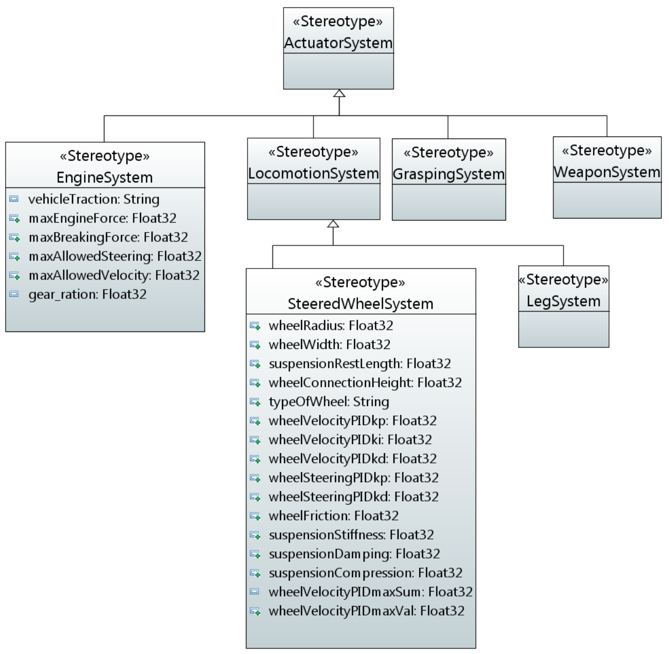
«Stereotype» ActuatorSystem
«Stereotype» EngineSystem
vehicleTraction: String
maxEngineForce: Float32
maxBreakingForce: Float32
maxAllowedSteering: Float32
maxAllowedVelocity: Float32
gear_ration: Float32
«Stereotype» LocomotionSystem
«Stereotype» GraspingSystem
«Stereotype» WeaponSystem
«Stereotype» SteeredWheelSystem
wheelRadius: Float32
wheelWidth: Float32
suspensionRestLength: Float32
wheelConnectionHeight: Float32
typeOfWheel: String
wheelVelocityPIDkp: Float32
wheelVelocityPIDki: Float32
wheelVelocityPIDkd: Float32
wheelSteeringPIDkp: Float32
wheelSteeringPIDkd: Float32
wheelFriction: Float32
suspensionStiffness: Float32
suspensionDamping: Float32
suspensionCompression: Float32
wheelVelocityPIDmaxSum: Float32
wheelVelocityPIDmaxVal: Float32
«Stereotype» LegSystem

Fig. 3. Actuators RobotML DSL concepts coming from the ontology

- Many concepts from the *simulation*, *experiment* and *mission* packages are related to abstract robotics concerns that can not be mapped to software realisation using code generation.
- While RobotML have been implemented as a UML profile, many ontological concepts are redundant with UML behavioral concepts, for instance, `Activity`, `Event` and `Evolution / System Dynamics`. Behavior of the systems have been implemented using directly the UML concept of `State Machine`.

In order to reach an agreed mobile robotic language, it was necessary to limit ourselves to most fundamental concepts such as sensors or actuators. At this step, it was not possible to introduce for instance agreed referential systems and more to the point, additionally to the ROS messages library, other common implemented libraries that would be used throughout the different targets. This issue is not structural and should be solved with additional work.

The RobotML domain Model is structured differently from the PROTEUS ontology packages. In fact, RobotML DSL is intended to be used to design robotic systems structure and behavior in order to simulate them or to deploy them to middleware implementations ported on real robots. Clearly the RobotML DSL use is different from the ontology use which is primarily to collect and organize knowledge about mobile robotic systems. That's why the structure of the RobotML domain Model reflects more the future use of the DSL than the structure of the ontology from which it was derived. The RobotML domain Model is structured around four main packages:

- The **Architecture Package** contains the concepts that help defining and composing a robotic system and the environment where it evolves. It includes concepts related to a wide range of hardware components (sensors, actuators, joints, chassis, mechanical linkage, etc) and software components (drivers, closed/open-loop control system, etc). Figure 3 shows concepts related to actuators coming from the PROTEUS Ontology.
- The **Behavior Package** contains the concepts for describing the behavior of robotic systems based on algorithms or finite state machines. To represent algorithms using RobotML DSL, we chose to use the ALF language (Action Language for Foundational UML)[2].
- The **Communication Package**: Communications are defined through the concepts 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 defined the concept of `Data Flow Port` which is related to the publish/subscribe model of communication. `Data Flow Port` enables dataflow-oriented communication between systems, where messages that flow across ports represent data items. On the other hand, we defined the concept of `Service Port` that supports a request/reply communication paradigm, where messages that flow across ports represent operation calls.
- **Deployment Package** specifies a set of constructs that can be used to define the assignment of a robotic system to a target robotic platform that can be a middleware platform or a simulator. The deployment is important because it feeds generators with the information on which platform the system will be executed.

3.2 The graphical modeling environment

After defining the domain model of the RobotML DSL, we chose to implement the DSL as a UML profile because UML contains generic metaclasses such as `Class` and `Port` that can be extended with specific concepts from the robotic domain having their own semantics. This semantics is consistent with the one provided by UML generic metaclasses. Afterwards, we developed the graphical modeling environment (see figure 4) as an extension of Papyrus: the open-source Eclipse based UML modeling tool. This tool provides customization features which enables us to develop the RobotML modeling environment in an agile and iterative approach.

4 Code Generation

As defined by DSL engineering methodology [1], there are two main approaches to build executable programs from

2. http://www.omg.org/spec/ALF/

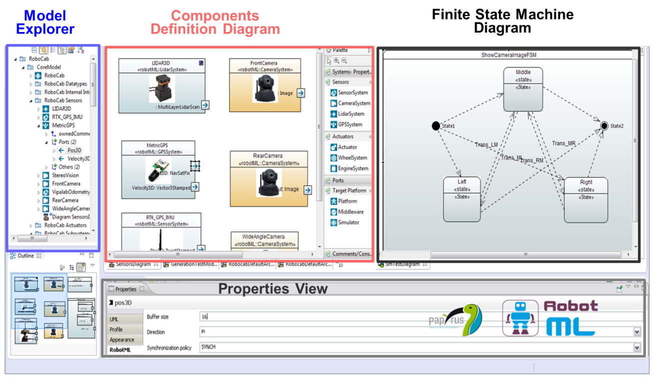

Fig. 4. RobotML graphical modeling environment

DSL models: transformation and interpretation. The former translates a DSL model into a language for which an execution engine on a given target platform already exists. In the latter case, we need to build a new execution engine which loads the DSL models and executes it directly. In the mobile robotics domain, we believe that the transformation approach is more adapted for code generation. This can be justified by two points:

- Building a new execution engine for robotics models is very complex, in particularly where DSL programs are specified as models, as in the case of the RobotML DSL.
- There are a diversity of robotic platforms and middleware that propose a set of abstractions and mechanisms to execute robotics programs. Therefore, targeting these platforms using code generation can help reusing their abstractions and mechanisms.

In the context of the RobotML DSL, code generation concerns translating RobotML models to the different targets platforms. This includes open-source and commercial middleware platforms (OROCOS-RTT [9], RTMaps[3][10] and Effibox[4] [11]) and two open-source simulator platforms (Blender Morse [12] and CycabTK[5]). The reader not familiar with one or several middleware implementations can find additional information in the Appendix.

To implement code generation from RobotML models, a set of requirements were identified:

- The link between the simulator and the middleware and between the middleware and the target platform must be easily configured: the switch from the simulator to the target platform must require only a configuration of sensors and actuators.
- Code generated for different target platforms must interoperate with each other.

In this section, we present RobotML code generators for several robotics simulators and middleware platforms.

3. http://intempora.com/products/rtmaps-software/overview.html
4. http://effistore.effidence.com/
5. http://cycabtk.gforge.inria.fr/

TABLE 1
Correspondence between on the one hand UML or RobotML and on the other hand OROCOS-RTT

UML or RobotML	OROCOS-RTT
Robotic System	Task Context
Attribute from UML	Attribute
Primitive DataType from UML	C++ types
Structured DataType from UML	Data structure
Interface from UML	Abstract class
Interface Operation from UML	Virtual Method
Operation from UML	Operation
Parameter	Parameter
Data Flow Port direction {In}	Input Port
Data Flow Port direction {Out}	Output Port
Service Port {Provided}	`this->provides (Operation)`
Service Port {Required}	`this->requires (OperationCaller)`

4.1 General principles

During code generation, RobotML models are translated into text artifacts (source files, configuration files, state machines code, etc.)

As mentioned in the previous section, a deployment plan is defined at the modeling level to assign a robotic system to a targeted robotic platform. In the deployment plan, the hardware components and the robot's modeled environment may be assigned to a simulator while software components are assigned to a middleware platform. That deployment plan constitutes the input point of RobotML code generators.

The code generator of each platform, either middleware or simulator, relies on its own transformation rules to translate its assigned elements into the model and their properties into their corresponding code using Acceleo tool[6]. Acceleo allows to define model-to-text transformations and supports modeling standards like UML, XMI, etc. The syntax implemented by Acceleo is template-based where each template uses:

1) queries to take information from the input models and
2) conditions and loops to combine them and generate code.

In order to ensure interoperability between generated code for different target platforms, a common data exchange protocol was introduced. Since the bus communication notion is not defined in the RobotML domain Model and considering that RobotML target platforms have been integrated as ROS [13] stacks, we have taken into account ROS topics during code generation.

4.2 Generation to middleware

Transformation rules link concepts, from UML and from the packages architecture, behavior and communication of the RobotML domain model, to middleware concepts. Table 1

6. http://www.eclipse.org/acceleo/

shows for instance, an extract of the mapping between on one hand UML and RobotML concepts and on the other hand OROCOS-RTT concepts.

The OROCOS-RTT code generator takes as input a RobotML model and provides:

- OROCOS-RTT components,
- Interfaces which define a set of abstract operations,
- Data Structures representing user defined data types,
- A OROCOS-RTT Program Script file configuring communication between components,
- RTT-LUA components which load the state machines describing the behavior of components,
- LUA state machines implemented using the rFSM package where states, transitions, events, effects and guards are explicitly specified.

The OROCOS-RTT code generator aims at providing an executable application. To do so, a set of artifacts are also generated including a makefile, a Cmakelists.txt where the components to be loaded are specified and a manifest.xml file which specifies which libraries have to be imported.

Some limitations exist when code generation is done to a target middleware platform. The most important ones are:

- In OROCOS-RTT and RTMaps, only `Atomic Systems` can embed executable code. `Composite Systems` do not have their own implementation. When starting the execution of an application, we considered that the whole diagram would be flattened which means all the `Composite Systems` would be removed and replaced by their content, and this recursively. In the end, only `Atomic Systems` would remain on the diagram.
- In RTMaps, RobotML `Service Ports` are not supported and will probably never be.
- In Effibox, which is not a component-based middleware platform, an application is a monolithic C++ code responding to dated events. Translating RobotML models into Effibox code consists in generating a skeleton of a directly compilable application including the subscription code to the various selected sensors and the implementation of modeled state machines. Concerning the communication aspect, Effibox interfaces are used.

4.3 Encountered difficulties

Different code generators were defined considering robots (ReSSAC, WifiBot, R-Trooper, etc.), their middleware platforms (RT-MAPS, OROCOS-RTT, Effibox, and partly ROS,) and associated environment simulators (CycabTK, Blender Morse). It was extremely difficult to create a stable basis simultaneously for all these components. Their necessary packaging led to an unforeseen workload. In particular the generation for Blender Morse is not available mainly because the input format of the Blender Morse version supported by Blender Morse developers changed during the course of the project. It was partly managed but there was nevertheless a lack of support. Compatibility of new versions of middleware or simulator with their previous input formats should be ensured by their developers in order to facilitate development of higher level tools.

5 Validation with case studies

As mentioned before, the following case studies were used to validate the PROTEUS toolset:

- Air scenario: An UAV inspects the environment in order to find the best place to land,
- Military unmanned aerial vehicles scenario: unmanned combat aerial vehicles perform a deep strike mission,
- Indoor scenario: a small ground robot climbs stairs in order to assess a building,
- Air ground scenario: a ground robot and an helicopter drone patrol in an area searching for intruders and then follow the first intruder detected,
- Landmark search scenario: this case study is used in this paper to illustrate the toolset validation process and is detailed in this section.
- Urban scenario: a taxi robot transports customers.

In this section, we illustrate the validation process with the case study concerning *landmark search*.

This case study deals with structured outdoor exploration and object searching. A Nexter Robotics Wifibot[7] robot has to move autonomously in an unknown environment in order to find a landmark.

This is a differential 4-wheel-drive robot: The two wheels of each side move at the same speed. Its dimensions are 30x35x30 cm^3 for about 5kg.

It is equipped with 4 infrared proximetric sensors, a WiFi communication board, a Hokuyo laser range finder (scanning over 270° every 0.25° and up to 30m, at a frequency of 40Hz), a Unibrain FireWire camera, a Haicom GPS, a VectorNav IMU and odometers.

The environment is structured with non-homogeneous ground (concrete, macadam, grass, etc.) that has only one level and no slope above 5%. There is no trap, no hole, no stairs, no water, nor wireframe obstacles. Geometrical obstacles clutter the area (concrete, wood, steel, plastic, etc.). Three successive scenarios were considered for this case study:

- **Explore and Find**: The robot is put in the input/output area and the run begins. The robot must avoid obstacles and find the landmark in minimal time. A computer monitors the robot by WiFi, without any action. It only records and prints all the data required to compute the evaluation metrics (including the map). It also prints the messages coming from the robot. No additional communication is allowed. The scenario ends when the robot stops at less

7. http://wifibot.com/

TABLE 2
Case studies used to validate the ontology

Case Study	Number of individuals for ontology validation
Urban	30
Unmanned aerial	33
Indoor	15
Air ground	47
Landmark search	55

TABLE 3
Validation coverage, as inferred by the reasoner, of basic classes and of direct sub-classes of the class `System`

Basic class	Number of individuals
`Activity`	6
`Event`	1
`Event` and `Interaction`	1
`Evolution`	4
`Frame`	2
`Information`	19
`Interaction`	27
`Probe` and `System`	8
`System`	111
`Transition`	1
Direct sub-class of the class `System`	
`Physical Object`	39
`Composite System`	26
`Software`	19
`Composite System` and `Physical Object`	12
`Software` and `Model`	11
`Software` and `Atomic System`	4
`Software` and `Model` and `Atomic System`	4
`Physical Object` and `Atomic System`	2
`Model`	1
`Atomic System`	1

than 5m from the landmark and sends a picture of it along with a ”mission terminated” message.

- **Explore, Find and Come back**. This scenario is similar to the previous one, but the robot has to come back to its input area after sending a picture of the landmark.
- **Explore, Share/Organize, Find and Come back**. In this variation of the previous scenario, two robots cooperate. It ends when the last robot comes back to the input area. This scenario encourages data sharing between the robots to minimize time.

5.1 Ontology validation

The validation objectives for the ontology are to check its consistency and completeness regarding the case studies. In order to fill those objectives, individuals are created and linked using properties in order to describe the different parts of each case study. As shown in Table 2 180 individuals of 5 among 6 case studies are used for validating the ontology.

The consistency is checked by running the reasoner after the introduction of each new individual. When an inconsistency is found by the reasoner an analysis is performed in order to decide if it is the result of an erroneous use of the ontology or if there is a consistency problem in the ontology.

The completeness is checked by assessing the specificity of the classes of the ontology that can be assigned to each new individual. If those classes are not specific enough, additional specializations are needed in the ontology.

With respect to the taxonomy presented in Figure 2, the individuals populate the ontology as presented in top of Table 3. It is noticeable that no individuals of the classes `Connector`, `Port` and `Protocol` are used for the description of the case studies. This reflects on the one hand that even if those concepts are obvious for architecture description language developers they are not naturally used for the description of robotic case studies and on the other hand that those concepts were added to the ontology lately. Another remark is that polymorphism is used on the one hand for `Event` and `Interaction` and on the other hand for `Probe` and `System`. For the second case, may be the ontology could be improved by stating that `Probe` is not a basic class but some specific kind or use of a `Sensor System`. However, for the first case it is clear that it is a correct use of polymorphism because internal events are not interactions and continuous interactions are not events. The last remark this that more that one half of individuals belong to the class `System`. This validate the basic design choice to center the ontology around this concept and to consider robots as `Composite Systems`.

The validation results indicate that the presence of disjoint classes together with polymorphism induces some consistency risks. This is the case for the `System` Class of the ontology. Indeed, there are different views for systems, for instance:

- A functional view with Classes `Motion Planning System`, `Platform Management System`, etc.
- An implementation view with Classes `Software`, `Physical Object`, etc.
- A compositional view with classes `Atomic System` and `Composite System`,

and in some views there are disjunctions. For instance a `Software` cannot be a `Physical Object`. The individuals created for all the case studies validate the concepts linked to the different specializations corresponding to the different views of the class `System` because their introduction in the ontology does not produce an inconsistency. The bottom of Table 3 presents the coverage of the validation for the classes inheriting directly from the Class `System`. It can be observed that, despite the fact that most individuals belong to a single class at that level of the class hierarchy, polymorphism is used for modeling case studies. The main usage is for considering physical objects either composite or atomic. Another relevant usage is for indicating some software that it is the model of itself. Another remark is the relatively low number of inferred

TABLE 4
Individuals members of the class Software (individual name given in the first column) that are also members of another class (class name given in the first line); a yes indicates that the individual belongs to the class

Individual	Motion Planning System	Platform Management System	Mission Management System	Security System
Navigator	no	yes	no	no
Path Planning	yes	no	no	no
Global Planning	no	no	yes	no
Proximity Map.	no	no	no	yes
Localization	no	yes	no	no
Local Mapping	yes	no	no	no

individuals for the class `Atomic System`. Working with an open-world assumption, it is possible for the reasoner to infer individuals for the class `Composite System` because those individuals aggregate other individuals but it is impossible for the reasoner to infer individual for the class `Atomic System` from absence of aggregation. Going deeper into the hierarchy of classes, table 4 presents for the illustrating case study some individuals from the `Software` class that are also specific kind of systems in a functional view. Those individuals contribute to the validation of the final version of the ontology because they produced inconsistencies only in its earlier versions.

Moreover the differences between classes are the result of their properties and of axioms involving those properties. For instance the difference between an `Atomic System` and a `Composite System` is that the latter aggregates other systems. Those axioms can also be a source of inconsistencies. The aggregation property is validated by generating hierarchies of individuals relevant for the case studies and checking consistency with the reasoner. For the landmark search case study examples of individuals used to test the aggregation property are:

- The odometry system that aggregates an odometer driver and four odometers.
- The left engine system that aggregates the left engine drive and two wheel motors.

An important result with respect to the aggregation property, is that common sense indicates that it could be transitive and that a `Composite System` aggregates at least two `Systems`. However, this kind of statement does not comply with DL restrictions and made the reasoner unable to analyze the ontology. Thus, for the validated version of the ontology either the transitivity or the cardinality constraint has to be removed. The choice has been to keep the cardinality constraint on aggregation property.

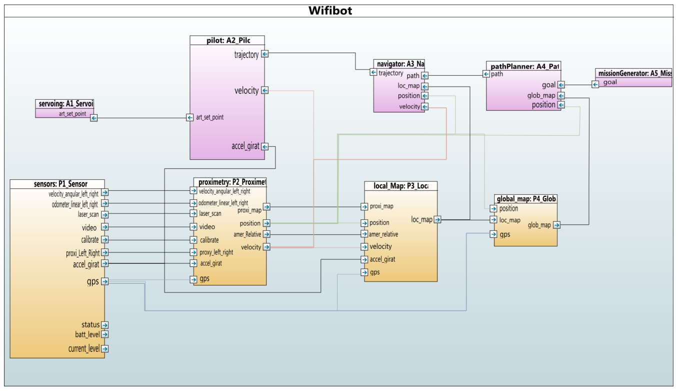

Fig. 5. Wifibot control architecture modeled with RobotML

5.2 RobotML modeling validation

The validation objectives for RobotML are to check its consistency, completeness and usability for code generation. In order to fill those objectives, models are designed for urban, air, and landmark search case studies.

For the landmark search case study, the control architecture of the Wifibot has been modeled. It is a multi level architecture including the actuators and sensors of the robot as well as the reactive and deliberative algorithms which control the robot. This model contributes to assess that, from a control architecture point of view, RobotML is quite complete.

In order to model this architecture with RobotML (see Figure 5), we have divided the functionalities in two parts: perception part (systems drawn as orange boxes) corresponding to functionalities related to sensor data processing and maps building, and action part (systems drawn as purple boxes) corresponding to functionalities related to decision making and control of the robot's actuators. This decomposition is a choice of the robotic architecture designers in order to fit their own needs, but it is not imposed. This shows some flexibility of the DSL.

In the perception part (orange boxes), P1_Sensors system models the links with the odometer, range finder, camera, GPS and the IMU of the Wifibot. The P2_Proximity system which draws proximity maps of the surrounding areas near the robot is shown in Figure 6. P3_Local and P4_Global model the algorithms in charge with the building of local and global maps respectively. Typically this part of the model represents the transformation of the data acquired by the sensors into more complex representation modes such as local and global maps. This transformation is performed integrating and crossing the various sensor information in space and time.

In the action part (purple boxes), we have the A1_Servoings system which models the link between the actuators of the robot and the software part of the robot. The A2_Pilot system generates set points in order to control the robot wheels velocities. A3_Navigation system, shown in Figure 7, generates trajectories for the A2 system, taking into account

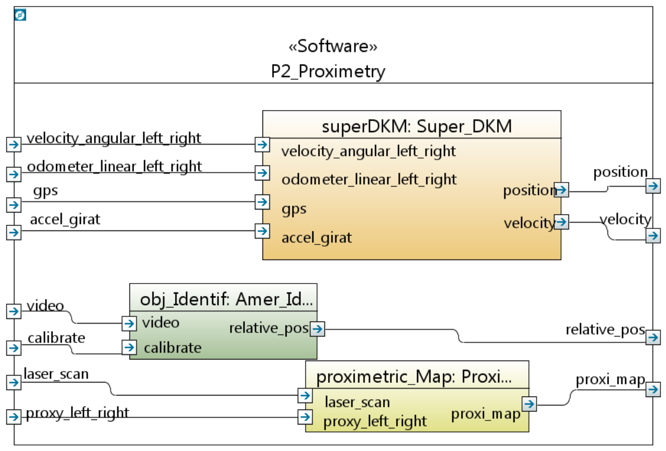

Fig. 6. The P2_Proximity box from the perception part of the Wifibot architecture

the constraints of the environment (given by P3 system) and following a global path provided by the A4_PathPlanner system. Lastly the A5_MissionGenerator system controls the current mission of the robot (explore, find or come back). This part of the model corresponds to the decision part of the architecture, upper level (A4 and A5) corresponding to more deliberative part whereas lower levels (A1 and A2) model the reactive parts of the robot (such as emergency avoiding maneuvers), A3 being the key transition part between reactive and deliberative behavior of the architecture.

Besides the landmark search model, two other case studies were modeled using RobotML. Table 5 shows the number of RobotML modeling artifacts for the three case studies. More specifically, we list:

- the number of packages used to structure each model,
- the number of components (sensors, actuators, software systems, etc) that were defined to model the whole case study,
- the number of user defined data types that were necessary to model the data flow communications between the components, it is worthy to note that RobotML language offers a library of data types that was inspired from ROS common messages,
- the number of graphical diagrams that were created for each case study.

The consistency verification of RobotML models was performed through validation rules implementing the semantics of the RobotML Metamodel. Examples of validation rules are:

- A DataFlow Port has to be typed by a DataType,
- A Service Port has to be typed by an Interface,
- An input port has to be connected to only one output port,
- a component can not contain itself (no recursion).

Concerning completeness, modeling of the PROTEUS case

TABLE 5
Case studies modeling with RobotML: number of modeling artifacts

Use Case	packages	components	user defined data types	diagrams
Landmark search	8	29	7	11
Air	1	16	9	12
Urban	14	41	16	18

studies has shown that RobotML contains the necessary concepts to model autonomous and mobile robotic applications. Moreover, all components of the use cases models have direct interpretation in terms of middleware, indicating the ability to generate code on the basis of a RobotML model. For the usability validation of RobotML, the graphical modeling diagrams (i.e, structural and behavioral diagrams) defined in the toolset were sufficient to model all the use cases.

5.3 Code generation validation

The validation objective for code generation is to check that the generated code is consistent with the RobotML model. Addition or omission with respect to the model shall be minimized. Validation of code generation has been conducted on 3 case studies: Air, urban and landmark search scenarios.

For the landmark search case study, automatic generation from RobotML on 2 middleware (RTMaps[8] and Effibox) has been performed and tests have been conducted in order to validate these generations, both on simulators and on the real robot. In order to perform these tests, we used simplified models of the architecture of the Wifibot compared to the one presented in Figure 5. The simplified model for the real robot includes systems A1, A2, P1 and P2. It allows obstacle avoidance behaviors. The simplified model for simulation includes also systems A3 and P3. It allows local map building and robot navigation. In this section we describe more precisely what was generated for each middleware.

For Effibox, each systemresults of the model which is allocated to Effibox in the deployment plan of RobotML creates source files in the Effibox generated files repository. An awp configuration executable is also created. The user can then launch this executable to configure the ports of the sensors of the application and define for each sensor if the sensor is a real one or a virtual sensor on a simulator. The generated source files are a skeleton of the robot architecture, with protected areas where the user can add its own robotics algorithms.

For RTMaps, a diagram corresponding to the modeled architecture is automatically generated when the deployment plan is launched in the RobotML interface. In this diagram one component (called package) by sub-system of the RobotML model is generated, and all those component are automatically

8. https://intempora.com/products/rtmaps.html

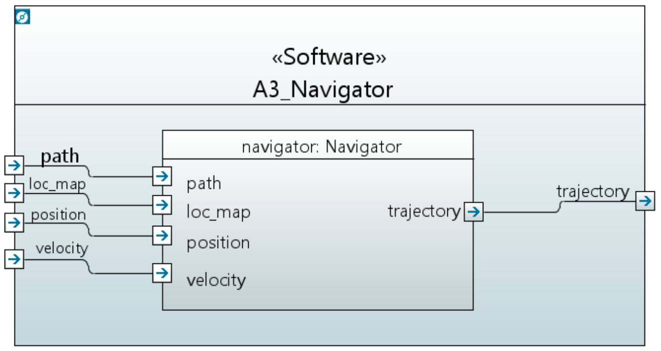

Fig. 7. The A3_Navigator box from the action part of the Wifibot architecture

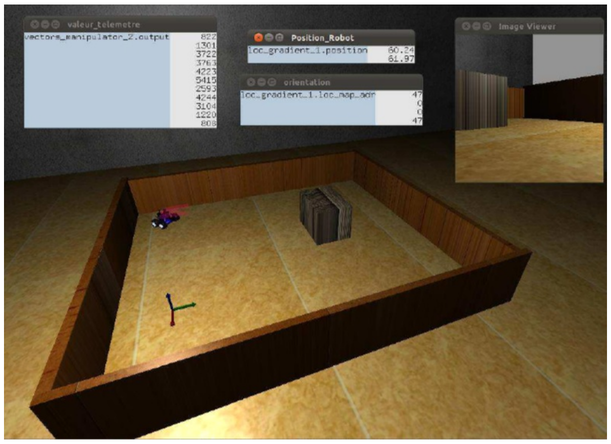

Fig. 8. Navigation simulation of the Wifibot by Blender Morse simulation engine

linked on the diagram with the right datatypes according to the RobotML model. Moreover macro components are created for the sensors and actuators on the model, in order to automatically integrate the conversion component between the datatypes used by RTMaps and those used by ROS (ROS datatypes being used as a bridge between middleware datatypes and and the 2 simulators). Once the packages are completed with the desired algorithms, the diagram can be turned on to launch the application.

5.4 Validating the ability to address variability

The main validation achievement was the capability to build effective running solutions for the different case studies:

- The urban case study was demonstrated with a Vipalab robot using the PAVIN platform;
- The air case study provided its users the capabilities to demonstrate capabilities in a real flying robot namely the ReSSAC in the Caylus environment;
- The indoor case study provided a Cameleon Robot to demonstrate capabilities to maneuverer in stairs;
- The landmark search case study provides education with a way to introduce robotics and complex systems.

Besides this ability to address various case studies, the ability to address variability inside each case study has been demonstrated. For instance for the illustrative case study, 3D modeling design of the Wifibot and navigation environment has been done during the project. These models have been designed with the 3D drawing software Blender and have been integrated with 2 simulators of the project: Blender Morse and CycabTK.

For the Blender Morse simulator (see figure 8), Python files are used to load the environment, the robot, the sensors with their respective positions and orientations in the scene. The simulation is launched with a wifibot.py file where a range finder, a gyroscope, a camera and actuators can be associated with the robot. Concerning the control part of the robot, an actuator can be added taking into account the linear and angular velocities of the robot by two 3-component vectors (Vx, Vy, Vz) (wx, wy, wz). For the communication with RTMaps, we use a component ROSTopicSubscriber to make the bridge between the information sent on the ROS node and the middleware.

For CycabTK, the modeling of the robot and of the environment is the same as those used on Blender Morse. The only difference is that the mgEngine simulator take decomposed Blender files as an entry. The simulator launches the simulation by taking into account the elements contained in the repositories (scripts, textures, meshes, materials). The component allowing communication with ROS and then RTMaps is the same as the one used for Blender Morse.

The interaction, between on the one hand the Blender Morse and CycabTK simulators and on the other hand the RTMaps and Effibox middleware, works as intended, using ROS as a bridge to share the data. We have been able to test the same navigation algorithm on each of the simulators.

We have tested and validated simulations between the pairs of following middleware; RTMaps and Effibox, and simulation engines; Blender Morse and CycabTK. All those simulations are generated from a single RobotML model presented in Figure 5. This model describes a control loop with a two sensors: laser range finders and odometers.

As shown in Figure 7, the information from those sensors is taken into account by a simple navigation algorithm in order to generate a trajectory for the robot.

Simulator generators are less mature than the middleware ones. They are working correctly only if RobotML components have their matching components in the simulators. This type of generation has not been used for the landmark search scenario. However the generation of the RobotML model to 2 middleware solutions has been validated, and cross validation between two middleware platforms and two simulators, on generated files coming from the same model has been successful.

6 Related Work

Managing variability is identified by the robotics community as one of the main challenges for the development of robotic systems [14].

Robotic middleware solutions [15] have been proposed to provide some interoperability throughout the introduction of some abstract sensors and actuators. For instance, ROS [13] introduces generic messages datatypes shared by common sensors or actuators. By using these messages, a robotic application does not depend on a particular specification of sensors or actuators but only depends on this introduced abstraction. However and with the emergence of many middleware solutions (at least, fifteen middleware solutions are presented in [15]), the proposed abstractions become very specific to each middleware. Consequently, it is more and more difficult for roboticists to maintain their robotics systems and to manage the variability throughout the different kinds of middleware.

RobotML does not introduce new middleware and tries to remain neutral regarding the choice of middleware. From the RobotML DSL it is possible to generate code to specific middleware. The drawback is that one needs to provide a new code generator every time new middleware is used. Moreover, code generators should be updated when changes in middleware occur. As we have already reported, we also did some experiments on how to integrate different middleware solutions at the same time by using a common ROS as a common message bus. But this integration is only done at the syntactic level and more work is needed to have a deeper semantic integration.

Robotics ontologies can be seen as another way to manage the variability among the robotic community.

As highlighted in [16], various robotic ontologies have been designed with different purposes. They range from the testing of decisional autonomy of robots [17], to the description of control architectures [18] [19] [20] of the robot navigation and action in different environments [21] [22] or specific robotics applications such as search-and-rescue missions [23] or mobile manipulation and service robots[9]. More recently, a core ontology has been developed for providing a common ground for further ontology development [24] [25] [26]. Moreover, Brugali [27], [28] proposed to use feature models to specify the concepts of robotics. In addition to concepts of the domain, feature models allow also specifying the variability at the domain level. As underlined by [29], there are some similarities between ontologies and features models. However, feature models is more useful to specify the variability in the perspective to implement software product lines.

The main originality of RobotML ontology, with respect to the existing robotic ontologies, is first at all that it is built around the concept of system (in the sense of system theory or DEVS [30]).

Compared to existing ontologies, the RobotML ontology was built quite differently.

For example, in [24] a consensus is searched among a quite large group of specialists keeping a high level of abstraction, an upper ontology is used and the language used (SUO-KIF) implements second order logic. No use of reasoner and individuals to validate the result is reported.

In our case, the ontology was written from scratch, a relatively small group of experts was involved but they were explicitly asked to provide case studies to design and validate the ontology. The language was restricted to implementation of DL allowing a systematic consistency check using a reasoner.

Just to illustrate some differences between [24] and our work: in the first one, a robot is an agent and a device that has robot parts. In our case, a robot is a composite system but only an autonomous robot is also an agent. [24] distinguish five levels of autonomy from fully autonomous robot to automated robot. Oddly, those classes are not subclasses of robot but relations between robots and processes. Moreover those relations are differentiated one from the others only by their names bringing no semantic at all. It seems that our ontology can lead to clearer and more explicit models for important robotic concepts than in [24].

Finally, Domain-Specific Modelling Languages (DSMLs) are also an alternative way to manage the variability and complexity of robotics systems. A complete and a recent survey on existing DSMLs for robotics in presented by Nordmann et *al.* [31]. All these DSMLs address differents needs in variability (see for example the VML and SmartTCL DSML defined in the SmartSoft project [32]), but tend to allow more flexible transformation and code generation, especially in the context of many different robotics middleware solutions.

The RobotML DSL was built iteratively after an ontology, without any specific middleware in mind. Because of that, the RobotML DSML is very system-oriented and define a robotic application as as set of interconnected hierarchical components.

7 Discussion

Some obvious limitations to code generation are induced by middleware capabilities: RobotML models may have a hierarchy of systems and different types of ports that are not supported by some middleware platforms. The consequences for the user are:

- a poor readability of the generated code when the middleware does not support the hierarchy or is not based on components,
- the necessity to take into account in the model the capabilities of the target middleware or to choose the target middleware according to the capabilities required by the model.

9. http://www.robot-standards.eu/

While both RobotML and the ontology can be substantially improved, lessons have been learned:

- An ontology should ideally be developed iteratively with frequent and numerous tests of its ability to grasp operational needs. Such a process would however be extremely long and expensive. The ontology developed here addresses only mobile robotic. The extension to other domains such as medical or manufacturing robotics is a challenge that requires a large number of resources. Moreover, cleaning the ontology by deleting useless concepts would require observing for a long time the additions in the part of the ontology that can be modified.
- The robotic ontology should ideally be available on the web for addition of individuals by users of the robotic community contributing with robotic problems or solutions. The simple solution provided here raised intrusion issues. A more secure web publication method for OWL files shall be studied.
- Grounding the DSL with the ontology proved quite difficult to achieve. This is not surprising as ontologies describes static entities while our DSL is meant to describe executable systems. For instance, for each entity of the ontology there was a choice to discard it, to put it into the DSL or into a support library.
- Keeping or not keeping in the ontology the concepts that are not directly part of the DSL is an open question. On the one hand those concepts are not useful in the perspective of code generation, on the other hand they seem useful to understand the concepts manipulated by the DSL. Criteria for cleaning the ontology taking into account the resulting DSL should be defined.
- Code generation from the DSL to the execution or simulation platforms are fragile to changes of the DSL, the target platform or the ontology. Obvious redundancies between the code generation transformations have been captured by common low-level transformations but this is too superficial so that not enough effort is factored out.
- A tool should be added to the toolset in order to allow the synchronization of a model written using the DSL with its corresponding set of individuals in the ontology. A user models a new robotic system by editing an OWL file and by using the RobotML graphical modeling environment. Ideally, the copy of common elements from the ontology to RobotML or from RobotML to the ontology should be automated.

Finally, maybe the most interesting lesson to draw from the PROTEUS attempt is that while high level abstractions that are produced in a top-down manner may be quite useful if they are standardized, they are quite difficult to define. The reason is that a consensus is extremely difficult to reach between experts.

A bottom-up approach should thus also be explored which would produce intermediate-level abstractions from concrete projects. Making each project's assumptions explicit and organizing them into a constantly evolving ecosystem may lead to more easily individually reusable abstractions. The main difference with a DL compliant ontology would be that no single viewpoint is assumed. The drawback of such an approach is that consistency when combining several abstractions would be more difficult to reach.

8 CONCLUSION

The PROTEUS project was an ambitious attempt to address one of the most difficult issues of robotics: its huge variability. The idea was to define both an ontology – a set of very high-level abstractions on top of those provided by middleware solutions – and a DSL to easily define robotics systems.

Six case studies were used by French robotic teams to ground and to validate this approach. One of them has been described and used to illustrate the validation in this paper. The results are contrasted yet encouraging. Code generation was only completely achieved for the OROCOS-RTT and RTMaps targets. However, using the DSL for the other middleware proved to be useful to define the architecture of the system. The complete RobotML toolset (ontology, DSL and code generators) is available as an open-source software on GitHub, https://github.com/orgs/RobotML/, and Eclipse Foundation git repositories, http://git.eclipse.org/c/papyrus/org.eclipse.papyrus-robotml.git.

As far as the coordination is concerned the way to routine modeling activity in the robotic field was reached. Nevertheless, the work presented in this paper has paved the way towards this direction. The toolset created needs to be consolidated and adopted at large by the community through several types of actions:

- the technical platform needs to be industrialized, the RobotML components need to be packaged and associated support should be provided;
- the built ontology is missing high level robotics decision making. This needs to be addressed;
- the ontology defined needs to be standardized or normalized in order to penetrate the different communities.

Performing those actions should lead to an improved design practice in the robotic community.

APPENDIX

The purpose of this appendix is to provide additional informations about target middleware that are considered for code generation. Indeed, the PROTEUS toolset implemented three code generators respectively for OROCOS-RTT, RTMaps, and Effibox.

A common function for the middleware is the abstraction of the operating system and the management of, on the one hand, communication between elements, corresponding to `Software` and `Atomic Systems` classes of the DSL and defined or configured by the user, and, on the other hand, dynamic of those elements. This function is performed respectively by the *real-time toolkit* and the *deployment component* for OROCOS-RTT, an *execution engine* for RTMaps and the *heart module* for Effibox.

The communication between `Atomic Systems` can be specified either by *ports*, for OROCOS-RTT `Atomic Systems` exchanging data, events or services and for RTMaps `Atomic Systems` exchanging only data, or by a *subscription* mechanism, for Effibox `Atomic Systems` exchanging only data. To exchange data, *data ports* should be connected to each other. For consistent *data types*, an *output port* can be connected to several *input ports* and an *input port* can be connected to a single *output port*. The communication graph between `Atomic Systems` can be either limited only by consistency of *data types* or *interfaces* or can depend also on types of `Atomic Systems`. For OROCOS-RTT and RTMaps `Atomic Systems` are *components* while for Effibox an `Atomic System` is either a *driver module*, responsible for the communication with an external entity like a sensor, a software, a computer, etc. or a *brik module*, a piece of software written by the user and corresponding to an application algorithm. There is a particular *driver module* for each communication bus (IEEE, CAN network, Ethernet network, Serial ports, etc.). For Effibox communication is only allowed between *driver modules* and *brik modules* leading to specific communication graphs.

For the three middleware, `Atomic Systems` can be developed or chosen in libraries. However, for Effibox the libraries include only driver modules.

OROCOS-RTT and RTMaps *components* have *properties*, i.e. variables that can be read from a file and therefore can be used to store values such as configuration parameters or persistent data, allowing to configure them. They also have *actions* that can be triggered. For OROCOS-RTT an *action* can be implemented as a function, as a script or as a state machine while for RTMaps it results in a specific function call. In terms of control of *component* activities, OROCOS-RTT have four available policies: *Non periodic*, *periodic*, *sequential* and *slave*. For RTMaps, each *component* executes in its own thread a loop in which it waits for and retrieve data on its inputs, then process it and finally output the results on its outputs. In terms of *data ports* there are some differences:

- For OROCOS-RTT *data ports* can be configured to use a FIFO buffer or not. Also, they can be write-only, read-only or read-write and are accessed in real-time in a thread-safe fashion that uses a mutual exclusion procedure. *Data ports* can also be configured to trigger the activity of a *component* or to execute a function upon reception of data.
- For RTMaps each output has a circular buffer, containing samples of data produced. Connected inputs register on this buffer and can read data. Several input behaviors are available. The FIFO behavior returns the oldest data that has not been processed or waits for a new sample. The last or next behavior retrieves the most recent unprocessed sample or waits for a new sample. The wait for next behavior waits for new data. The sampling behavior returns immediately the most recent sample available. The never skipping behavior is like the FIFO one but the downstream component blocks an upstream one in case the buffer is filled.

A specificity of OROCOS-RTT is to have in each *component* an *execution engine block*. It is executed according to the activity of the *component* and implements the processing of the scripts, commands, events and state machines associated to the *component*.

RTMaps and Effibox date accurately each sensor data gathered. This permits to the user to replay all the recorded sensor data at the desired speed. In order to replay exactly the sensor data in the same way that they were recorded (order of sensor data reception and interval time between two sensor data). For Effibox the date of each sensor data corresponds to the reception date of the data by Effibox during recording. RTMaps time stamping associates two times with each data: The timestamp that is definitively set by the most upstream producer and the time of issue that is the date at which the sample was output from the last *component* it went through.

RTMaps has specific features for distributed applications: clocks of instances are synchronized using either an IP based master-slave scheme or an external time source in a component for each instance. Exchange of data between instances is done using specific components.

References

[1] 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. [Online]. Available: http://www.dslbook.org 1, 4

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/352029.352035 2

[3] B. Parsia and E. Sirin, “Pellet: An OWL DL reasoner,” in *Third International Semantic Web Conference-Poster*, vol. 18, 2004. 2

[4] D. Tsarkov and I. Horrocks, “FaCT++ description logic reasoner: System description,” in *Automated reasoning*. Springer, 2006, pp. 292–297. 2

[5] R. Shearer, B. Motik, and I. Horrocks, “HermiT: A highly-efficient OWL reasoner.” in *OWLED*, vol. 432, 2008, p. 91. 2

[6] D. L. McGuinness, F. Van Harmelen *et al.*, “OWL web ontology language overview,” *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004. 2

[7] C. Matuszek, J. Cabral, M. J. Witbrock, and J. DeOliveira, “An introduction to the syntax and content of Cyc,” in *AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*. Citeseer, 2006, pp. 44–49. 2.1

[8] I. Niles and A. Pease, “Towards a standard upper ontology,” in *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*. ACM, 2001, pp. 2–9. 2.1

[9] H. Bruyninckx, “Open robot control software: the OROCOS project,” in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, May 2001, pp. 2523–2528 vol.3. 4

[10] N. du Lac, C. Delaunay, and G. Michel, “RTMaps - real time, multisensor, advanced prototyping software,” in *3rd National Conference on Control Architectures of Robots*, 2008. 4

[11] C. Tessier, C. Cariou, C. Debain, F. Chausse, R. Chapuis, and C. Rousset, "A real-time, multi-sensor architecture for fusion of delayed observations: application to vehicle localization," in *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE*, Sept 2006, pp. 1316–1321. 4

[12] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, and S. Stinckwich, "Simulating complex robotic scenarios with MORSE," in *SIMPAR*, 2012, pp. 197–208. 4

[13] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009. 4.1, 6

[14] A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, and C. Schlegel, "Managing run-time variability in robotics software by modeling functional and non-functional behavior," in *Enterprise, Business-Process and Information Systems Modeling - 14th International Conference, BPMDS 2013, 18th International Conference, EMMSAD 2013, Held at CAiSE 2013, Valencia, Spain, June 17-18, 2013. Proceedings*. Springer, 2013, pp. 441–455. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38484-4_31 6

[15] N. Mohamed, J. Al-jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *In IEEE International Conference on Robotics, Automation, and Mechatronics (RAM 2008)*, 2008, pp. 736–742. 6

[16] S. Dhouib, N. Du Lac, J.-L. Farges, S. Gerard, M. Hemaissia-Jeannin, J. Lahera-Perez, S. Millet, B. Patin, S. Stinckwich *et al.*, "Control architecture concepts and properties of an ontology devoted to exchanges in mobile robotics," in *6th National Conference on Control Architectures of Robots*, 2011. [Online]. Available: http://www.lirmm.fr/gtcar/webcar/CAR2011/files/2011/05/submission15.pdf 6

[17] P. Deplanques, "Vers le test de l'autonomie des robots: une ontologie de la robotique," Ph.D. dissertation, 1996. 6

[18] F. M. Casas and L. A. García, "OCOA: An Open, Modular, Ontology Based Autonomous Robotic Agent Architecture," in *Artificial Intelligence: Methodology, Systems, and Applications*. Springer, 2002, pp. 173–182. 6

[19] W. Hwang, J. Park, H. Suh, H. Kim, and I. H. Suh, "Ontology-based framework of robot context modeling and reasoning for object recognition," in *Fuzzy Systems and Knowledge Discovery*. Springer, 2006, pp. 596–606. 6

[20] M. Tenorth and M. Beetz, "KnowRob—knowledge processing for autonomous personal robots," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. IEEE, 2009, pp. 4261–4266. 6

[21] A. Chella, M. Cossentino, R. Pirrone, and A. Ruisi, "Modeling ontologies for robotic environments," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. ACM, 2002, pp. 77–80. 6

[22] J. Bateman and S. Farrar, "Modelling models of robot navigation using formal spatial ontology," in *Spatial Cognition IV. Reasoning, Action, Interaction*. Springer, 2005, pp. 366–389. 6

[23] C. Schlenoff and E. Messina, "A robot ontology for urban search and rescue," in *Proceedings of the 2005 ACM workshop on Research in knowledge representation for autonomous systems*. ACM, 2005, pp. 27–34. 6

[24] "IEEE Standard Ontologies for Robotics and Automation." [Online]. Available: http://dx.doi.org/10.1109/ieeestd.2015.7084073 6

[25] C. Schlenoff, E. Prestes, R. Madhavan, P. Goncalves, H. Li, S. Balakirsky, T. Kramer, and E. Miguelanez, "An IEEE standard ontology for robotics and automation," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 1337–1342. 6

[26] E. Prestes, J. L. Carbonera, S. R. Fiorini, V. A. Jorge, M. Abel, R. Madhavan, A. Locoro, P. Goncalves, M. E. Barreto, M. Habib *et al.*, "Towards a core ontology for robotics and automation," *Robotics and Autonomous Systems*, vol. 61, no. 11, pp. 1193–1204, 2013. 6

[27] L. Gherardi and D. Brugali, "Modeling and Reusing Robotic Software Architectures: the HyperFlex toolchain," in *IEEE International Conference on Robotics and Automation (ICRA 2014)*. Hong Kong, China: IEEE, May 31 - June 5 2014. 6

[28] D. Brugali and M. Valota, "Software Variability Composition and Abstraction in Robot Control Systems," in *Proc. of the 16th International Conference on Computational Science and Applications (ICCSA 2016)*, O. Gervasi and et al., Eds. Beijing, China: Springer, July 4-7 2016, pp. 358–373. 6

[29] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg, "Feature models are views on ontologies," in *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*. IEEE Computer Society, 2006, pp. 41–51. [Online]. Available: http://dx.doi.org/10.1109/SPLINE.2006.1691576 6

[30] B. P. Zeigler, "Devs representation of dynamical systems: Event-based intelligent control," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 72–80, 1989. 6

[31] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal of Software Engineering for Robotics*, 2016. 6

[32] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. F. Inglés-Romero, and C. Vicente-Chicote, "Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot," *Journal IT - Information Technology: Methods and Applications of Informatics and Information Technology*, vol. 98, pp. 57–85, 2015. 6

Jean-Loup Farges was born in Clamart, France, in 1956. In 1978 he received a degree in electric engineering from ENSEEIHT, Toulouse, France, and a M.S. degree in automatic control from Paul Sabatier University in Toulouse. He received his Ph.D. from ISAE, Toulouse, in 1983. He has since been working as a Research Scientist for Onera, the French Aerospace Lab. Initially, his main area of interest was road traffic control. Since 2003 his research focuses on planning and execution control for autonomous aerospace vehicles and on optimization in air traffic control.

Serge Stinckwich is an associate professor (maître de conférences) at Université de Caen Normandy and a researcher at UMMISCO, a joint research unit team from IRD and Université Pierre and Marie Curie. He received a Ph.D. from University of Savoy in 1994. His main research area of interest is modelling and simulation of complex system with domain-specific languages and tools, applied to rescue robotics, crisis management and epidemiology.

Mikal Ziane is an associate professor (maître de conférences HDR) at Université Paris Descartes and a researcher at Laboratoire d'Informatique de Paris 6 (LIP6), Université Pierre et Marie Curie (UPMC). He received a Ph.D. from UPMC in 1992. His main research topic is building knowledgeable assistant tools. This requires identifying programming knowledge and finding efficient and convenient ways to include it into assistant tools. Part of this knowledge can be represented using program transformations, transformation strategies or constraints (e.g. coupling constraints).

Selma Kchir graduated in 2010 from the University of Montpellier 2 with a master's degree in Software Engineering. She then started a Ph.D. in software engineering for robotics at Université Pierre et Marie Curie (UPMC). Her thesis focused on facilitating the development of robotics applications through software engineering technologies. She received her Ph.D. in 2014 and joined the Interactive Robotics Laboratory of CEA-LIST. Her current research focuses on software architecture and design of robotics applications for industrial, healthcare and hazardous environments (humanoid robotics, manipulation, mobile robotics, aerial robotics).

Bruno Patin was born in Saint Quentin, France in 1961. In 1983, he achieved a master degree in theoretical physics from Paul Sabatier university (Toulouse, France) and, in 1986, a degree in Electronic from ENSERG (Grenoble, France). From 1987 to 1998, he was in charge in Dassault Aviation of creating and managing the on ground electromagnetic test facilities. Since 1998, he is working on all aspects linked to autonomy of the unmanned platform for the same company. It has led him to promote a thesis and an European project in the field of visual servoing as well as to lead the ANR PROTEUS project that helped to create the knowledge underlying this paper. He is now involved in many committees trying to break the regulatory barrier that limit the use of drones in France and Europe.

Cyril Novales received an Engineer diploma in Electronics from ISIM Institute (Montpellier) in 1989, and a Ph.D. degree in 1994 in robotics from the University of Montpellier II (France). From 1995 to 1997, he worked on autonomous vehicles in INRIA of Grenoble. In 1997, he joined the University of Orleans (France), where he is associate professor in the laboratory PRISME (previously LVR). Since this date, he has been involved in the design and the development of mobile robots and medical robots. He worked on teleoperated robotics project (OTELO, TERESA), and more recently on PROSIT and PROTEUS national projects, focusing on architectures for tele-operated or/and autonomous robots.

Tewfik Ziadi is an associate professor (*maître de conférences*) at Université Pierre et Marie Curie (UPMC) and a researcher at Laboratoire d'Informatique de Paris 6 (LIP6). He received a Ph.D. from University of Rennnes 1 in 2005. His main research area of interest is related to software product lines engineering. This includes variability management and the extraction of variability from existing legacy artefacts.

Nicolas Morette holds a post-doc position in the Laboratoire PRISME of the University of Orleans (France). He received a Ph.D. in robotics of the University of Orleans in 2009. He worked on the PROTEUS project in the same laboratory (robotic development platform) and on medical robotics projects (needle guiding for local anesthesia and teleoperated echography). His main research focus is on navigation for mobile robots, trajectory generation using direct kinematics model and predictive control approaches.

Saadia Dhouib received her Ph.D. degree from University of South Brittany (France) in 2010 and graduated before from the National School of Computer Sciences (ENSI-Tunis) as a Computer Science engineer. She is currently a research engineer at the CEA LIST. She is working on modeling and analysis of complex systems. She was involved in several research projects: PROTEUS a project on robotic modeling and transformations for end-users and scientific communities, EP-IT a French research project on modeling and simulation of smart grids, and she is currently involved in the French research project ROMEO2 on humanoid robots.

François Marmoiton received his Ph.D. degree from the University of Clermont Ferrand II in 2000. From 2000 to 2002, he worked on image compression as associate professor in the laboratory IRCOM-SIC of Poitiers. Since 2002, he is research engineer at CNRS in Institut Pascal laboratory (Clermont Ferrand University). He works on the developement of mobile robots like driverless shuttles, based on multisensors perception systems.