Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems

Federico Ciccozzi[1], Davide Di Ruscio[2], Ivano Malavolta[3], Patrizio Pelliccione[4]

[1] School of Innovation, Design and Engineering (IDT), MRTC, Mälardalen University, Västerås, Sweden
[2] Information Engineering, Computer Science and Mathematics Department, University of L'Aquila, L'Aquila, Italy
[3] Gran Sasso Science Institute, L'Aquila, Italy
[4] Department of Computer Science and Engineering, Chalmers University of Technology | University of Gothenburg, Gothenburg, Sweden

Abstract. The near future will be pervaded by robots performing a variety of tasks (e.g., environmental monitoring, patrolling large public areas for security assurance). So far, researchers and practitioners are mainly focusing on hardware/software solutions for specialized and complex tasks; however, despite the accuracy and the advanced capabilities of current solutions, this trend leads to task-specific solutions, difficult to be reused and combined.
In this paper we propose a family of domain-specific languages for specifying missions of multi-robot systems by means of *models* that are (i) independent from the technologies, (ii) ready to be analysed, simulated, and executed, (iii) extensible to new application areas, and (iv) closer to the problem domain, thus democratizing the use of robots to non-technical operators. We show the applicability of the proposed family of languages in two real application domains consisting on the adoption of unmanned multicopters and autonomous underwater vehicles.

1 Introduction

The near future will be pervaded by robots that, moving underwater, on terrain, or flying, will simplify everyday tasks and will open a myriad of new opportunities. Mobile multi-robot systems (MMRS) are special kinds of robotic systems in which robots behave as a team and each single robot accomplishes a well defined task towards the accomplishment of a global mission. On the one side a multi-robot team can accomplish a mission more quickly than a single robot, and on the other side a multi-robot team can accomplish missions requiring particular capabilities that might be impossible or impractical to find on a single robot: a team can make effective use of specialists designed for a single purpose, e.g., scouting an area or picking up objects.

The specification of a mission is difficult when considering a single robot since many details should be taken into account, and it might require technical expertise about the dynamics and the characteristics of the used robot. It becomes even more complex when dealing with missions involving multi-robots. Then, it emerges the need of software engineering approaches and methodologies especially tailored to develop and maintain multi-robot systems.

Model-driven Engineering (MDE) [**?**] is a promising methodology for simplifying the design, implementation and execution of software systems. In MDE, we can notice a shift from third generation programming language code to models expressed in domain-specific modeling languages (DSMLs). In this context, MDE enables the development of MMRSs by means of models defined with concepts that are much less bound to the underlying technology and are closer to the problem domain. This makes the models easier to specify, understand, and maintain, helping the understanding of complex problems and their potential solutions through abstractions [**?**].

In this paper we present a family of DSMLs for specifying civilian missions of MMRSs. The proposed languages are organized in different layers going from languages conceived for the end user, namely those to describe missions and their spatial context, an intermediate language describing the detailed behaviour of each robot (hidden to the user), and the robot language containing the hardware and low-level specification of each type of robot within the team. This paper extends and refines a previous paper [**?**], in which we published first results in the direction of defining a family of DSMLs for specifying MMRSs. In order to show the applicability of the proposed languages in practice, in [**?**] we instantiated the family of languages for the domain of civilian missions of autonomous multicopters. However, the first version of the languages presented in [**?**] was tightly coupled to one specific domain, and consequently we refined them in order to enable their adoption in different application domains. In this paper we show the adoption of the refined languages also in the domain of autonomous underwater vehicles, in addition to the autonomous multicopters one.

Roadmap of the paper. The remainder of the paper is structured as follows: a description of civilian missions is provided in Section 2. The architecture of the proposed family of languages is presented in Section 3. Their instantiation and implementation into two specific application domains is provided in Section 4, namely: the description of their application on autonomous multicopters is provided in Section 4.1, while their instantiation and implementation to the domain of autonomous underwater vehicles is presented in Section 4.2. Section 5 discusses related work, whereas Section 6 concludes the paper and outlines future work.

2 Model-Driven Engineering for Mobile Multi-Robot Systems

The family of languages that we are proposing in this paper are especially conceived for civilian missions. In the literature several civilian missions have been discussed. Skrzypietz [**?**] classifies civilian missions for Unmanned Aircraft Systems (UAS) in six categories:

- *Scientific Research*, such as atmospheric, geological research, forestry, but also studying hurricanes, observing volcano, agriculture and forestry.
- *Disaster Prevention and Management*, like damage assessment after earthquakes, searching for survivors after airplane accidents and disasters.
- *Homeland Security*, such as coastal surveillance, securing large public events.
- *Protection of Critical Infrastructure*, such as monitoring oil and gas pipelines, protecting maritime transportation from piracy, observing traffic flows.
- *Communications*, like broadband communication, telecommunication relays.

– *Environmental Protection*, such as pollution emission, protection of water resources.

According to Washington Post[1], venture investors in the United States poured $40.9 million into drone-related start-ups in the first nine months of 2013, more than double the amount for all of 2012. Moreover, according to the "Unmanned Aerial Vehicle (UAV) Market (2013-2018)" market research, the total global UAV Market (2013-2018) is expected to reach $8,351.1 million by 2018[2]. This is justified by the number of advantages that the use of these devices brings, like:

– *reduced costs*: civilian missions typically requires high costs for personnel which have to be carried on-site, and to the communication overhead required for synchronization purposes of the teams;
– *increased safety*: on-site personnel may be exposed to significant risks (e.g., in case of fire, earthquake, and flood);
– *improved timing and endurance*: monitoring activities are very time consuming. Moreover, these activities are usually stopped during the night, slowing down the execution of the mission.

Even though the benefits coming from the employment of UAVs in the execution of civilian missions are highly relevant, their adoption is compromised by a number of technical difficulties especially for non-technical users, that have typically good experience in the domain of environmental monitoring missions, but a poor (if any) experience in the management of complex systems like UAVs. In particular, currently on-site operators must deeply know all the types of used UAVs in terms of, e.g., behavioural dynamics and hardware capabilities in order to correctly operate with them. Additionally, on-site operators have to simultaneously control a large number of UAVs during the mission execution.

In order to reduce technical barriers and enabling a wider adoption of UAVs, MDE techniques and tools can be adopted to permit users to employ UAVs even without a detailed knowledge of either the robots to be used or their coordination and planning. Abstraction, automation, and analysis are the peculiar aspects of the MDE methodology. A key tenet of MDE is that it advocates the systematic use of models as first class entities throughout the software development life-cycle. Models represent abstractions of real systems that are analysed and engineering by identifying a coherent set of interrelated concepts precisely captured in metamodels. A model is said to conform to a metamodel, or in other words it is expressed by the concepts encoded in the metamodel. Model transformations are exploited to generate target models out of source ones. Models are analysed by means of queries specifically conceived with respect to the properties to be checked. By means of MDE it is possible to systematically concentrate on different levels of abstractions at which all involved stakeholders can operate for (i) improving the quality of MMRSs in terms of, e.g., safety, reliability and reusability, (ii) reducing the intrinsic variability and complexity of today's MMRSs, and (iii) promoting the reuse of software and hardware components across MMRS.

[1] `http://wapo.st/1bJueLH`
[2] `http://www.marketsandmarkets.com/Market-Reports/unmanned-aerial-vehicles-uav-market-662.html`

In the remaining of the paper we propose a family of modeling languages supporting the specification of civilian missions to be executed by MMRSs. A combination of model transformations, code generation, and formal reasoning can be used to automatically transform the mission specified by means of the proposed languages in low-level instructions provided by the used autonomous robots. In this way, civilian missions can be specified also by end users that have limited IT and robotics knowledge, but that are experts in the domain of civilian missions.

3 The Family of Languages

As a whole, the family of DSMLs we propose aims at supporting the specification of missions of MMRSs, and their execution at run-time. The family of DSMLs is com-

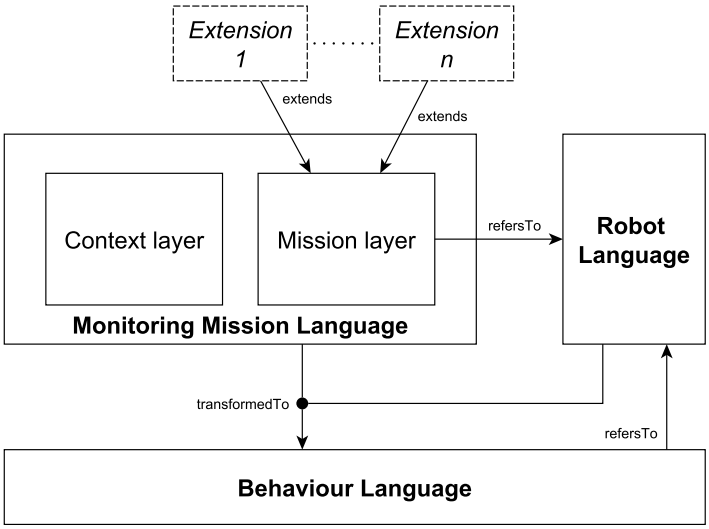


Fig. 1. The family of domain-specific modeling languages

posed of three different languages, presented below, each of them specifically tailored to represent a key aspect of the mission. Table 3 summarises their key features, whereas Figure 1 represents their relationships.

The *Monitoring Mission Language* (MML) is the language especially conceived for domain experts. It is composed of two distinct layers, namely the *Mission layer* and the *Context layer*. The former enables the specification of civilian missions without referring to specific aspects like the technical characteristics of the robots that will be used to execute the missions; basically, it permits to specify missions as sequences of tasks, suitably linked together via task dependencies, forks and joins. The specification of a mission is complemented by the definition of the spatial context in which the mission

Name	Stakeholders	Robot-independent	Extensible
MML	Operator, Platform extender	✓	✓
RL	Robot engineer	-	-
BL	-	✓	-

Table 1. Features of the proposed modelling languages

will be physically executed. Elements of the context can be modeled by means of the context layer of the MML language.

The type and configuration of the robots that will be in charge of realizing the specified mission are described through the *Robot Language* (RL). The *Behaviour Language* (BL) is a language that is hidden to domain experts. It contains a specification of the atomic movements and actions of the robots being considered in the mission. As shown in Fig. 1, both the monitoring mission language and the behaviour language have a reference to the robot language (currently those references are implicitly established by the name of the referenced robot). This is needed because: (i) the mission layer of the monitoring mission language contains a list of all the robots of the swarm, and the type of each of them must be specified, and (ii) the behaviour language contains all the low-level movements and actions of the robots, and so their type must be known in the model in order to correctly instruct the robots at run-time.

By referring to Table 3, the stakeholders of the proposed family of languages are:

1. *Operator*: the in-the-field stakeholder specifying the mission as an MML model. Examples of operators include fire fighters, policemen, or geologists;
2. *Robot engineer*: the engineer that models a specific type of robot (e.g., the Phantom 2 from DJI technology[3]) via an RL model, and develops a corresponding controller for instructing the robot according to the basic operations supported by BL;
3. *Platform extender*: the domain and MDE expert who extends the MML mission layer, if needed, in order to support recurrent tasks and actions in a specific operational context (e.g., the identification of a fire perimeter in case of emergency, accurate and multi-perspective crime-scene inspection, solar panels inspection within a given area, etc.).

The languages shown in Fig. 1 have four key properties as discussed below:

- *Context models reusability:* the mission and context layers of the MML language are kept separated in order to allow mission operators to reuse already existing context models across missions and different organizations. Also, the context layer puts restrictions on the mission layer since they share the same location, thus enabling for a straightforward (automatic) composition of the two.
- *Tasks definitions extensibility:* in order to be as close as possible to the domain of the mission being modelled, the language underlying the MML mission layer has been designed to be extensible. Specifically, a platform extender, who is a domain and MDE expert, can extend the MML mission layer with new kinds of tasks supporting a specific application domain (e.g., agricultural missions, security-oriented

[3] `http://www.dji.com/product/phantom-2`

missions, smart grid monitoring missions, etc.). Those extensions are performed once for each application domain being considered, and can be reused across missions and organizations.
- *Behaviour generation capability:* the languages have been designed to support the automatic generation of BL models from MML and RL models. Consequently, the family of languages enables to (i) obtain BL models that are inherently consistent to their corresponding MML models, and (ii) to mask the complexity of low-level details about the used robots (and their actions) to on-site operators. In light of this, the control code for the robots depends only on the constructs of BL, and thus can be either automatically generated or directly executed by interpreting BL models at run-time (we do not constrain platform developers to any of those two strategies for the sake of flexibility).
- *Analysis-orientation:* issues related to mission correctness, e.g., safety and security, are fundamental for civilian missions for MMRSs. In this context, the proposed languages have been designed to be generic enough for describing this kind of missions from a high-level point of view, and thus mission correctness is not part of the languages themselves. In any case, the proposed languages provide the right level of abstraction for allowing analysis tools to be executed on the models for proving, for example, safety and security properties. We believe that the behaviour language is the best candidate for these kinds of analysis, since it can be easily transformed into a corresponding state machine, a process algebra, a Petri net, etc., thus allowing engineers to reuse already existing analysis tools.

The remainder of this section will describe the family of languages. Specifically, for each language belonging to the family we present the corresponding metamodel i.e., the abstract syntax of the language. For what concerns their concrete syntax, MML can have a graphical syntax like, e.g., an overlay on a geographical map representing the various tasks, dependencies and contextual elements (see Section 4.1 for a concrete example). Differently, RL and BL are represented by using an XML-based representation since they will be created and managed by domain experts and they are consumed and produced by other software components.

3.1 Monitoring Mission Language (MML)

In this section we will describe the two layers composing the monitoring mission language.

Mission Layer The mission layer of MML has been conceived by analyzing and extracting those concepts that are involved when specifying monitoring missions. According to the language, a monitoring mission consists of a number of dependent `Tasks` to be executed by a `Team` of `Robots`. Each robot has its own *home* position represented by means of the corresponding `Coordinate` element (see Fig. 2).

In order to reach the needed level of flexibility and expressiveness, in our language a coordinate can be either a `GeoCoordinate` or a `RelativeCoordinate`. Specifically, a geographic coordinate represents a geographic point by means of the well-established latitude and longitude values, according to a given coordinate reference

system (see the *crs* attribute of `Mission`). Also, the *altitude* attribute represents the geodetic height of the coordinate, if needed. The *depth* optional attribute specifies the vertical distance below a surface; it may refer to both ground or water surface. A relative coordinate is specified by the x, y, and z values, and it represents the distance in the three-dimensional space with respect to another coordinate, in meters. In other words, a relative coordinate is defined as a point in the Cartesian space, where the origin is the point identified by the *reference* feature.

The MML mission layer contains three abstract task metaclasses, each of them focusing on a specific kind of geometric entity being considered. More specifically, `PointTask` represents tasks that refer to a specific point of the environment. To this end the *point* reference represents the coordinate of the considered point. `LineTask` represents tasks that refer to a set of points forming a polyline in the environment. Consequently, a line task consists of the *initialPosition* that the considered robot will have at the beginning of the task, and a set of *points* consisting of the points of the polyline. Also, `PolygonTask` represents tasks that refer to specific areas. A polygon task consists of the *initialPosition* reference, and the *shell* referencing the points representing the border of the area that will be involved during the execution of the task.

`ControlTask` is an abstract metaclass that represents the synchronization tasks of the mission. In particular, each task of the considered mission can be executed by one or more robots and can be performed in sequence (see the metaclass `Join`) or in parallel (see the metaclass `Fork`) to other tasks of the modeled mission.

As already introduced, the mission layer (and so its corresponding metamodel) is defined to be extensible. This means that it specifies only general tasks that need to be specialized according to specific needs of the actual civilian mission (see Sect. 2), and to the robots that will be used. The extensibility of MML allows operators to achieve versatility and strong adherence to the environmental monitoring mission domains. More specifically, MML can be extended with additional constructs that are specifically tailored to the considered domain. For instance, if operators are interested to monitoring solar panel installations in a rural environment, MML might be extended with the concept of solar panel groups, thermal image acquisition tasks, and solar panel damage discovery and notification tasks.

Context Layer The specification of monitoring missions includes also the description of the context where they will be executed. By referring to [**?**], this modeling layer concerns spatial and situational contexts. Indeed it represents those portions of geographical areas that have some relevant property, and those elements which can influence the execution of the mission, but that are not part of the mission itself. For example, context models contain information about obstacles, forbidden areas, emergency places where to land in case of emergency. Fig. 3 shows the metamodel of the context layer of MML.

In particular a given monitoring mission will be executed in a `Context` consisting of a number of *forbiddenAreas*, *emergencyAreas*, and *obstacles*. More specifically, a forbidden area represents a specific area within the environment in which robots cannot enter in any way, whereas an emergency area represents a safe area that robots can reach in case of emergency (e.g., for a safe emergency landing of a flying drone). Both forbidden and emergency areas are described by their *shell*, i.e., the ordered set of points

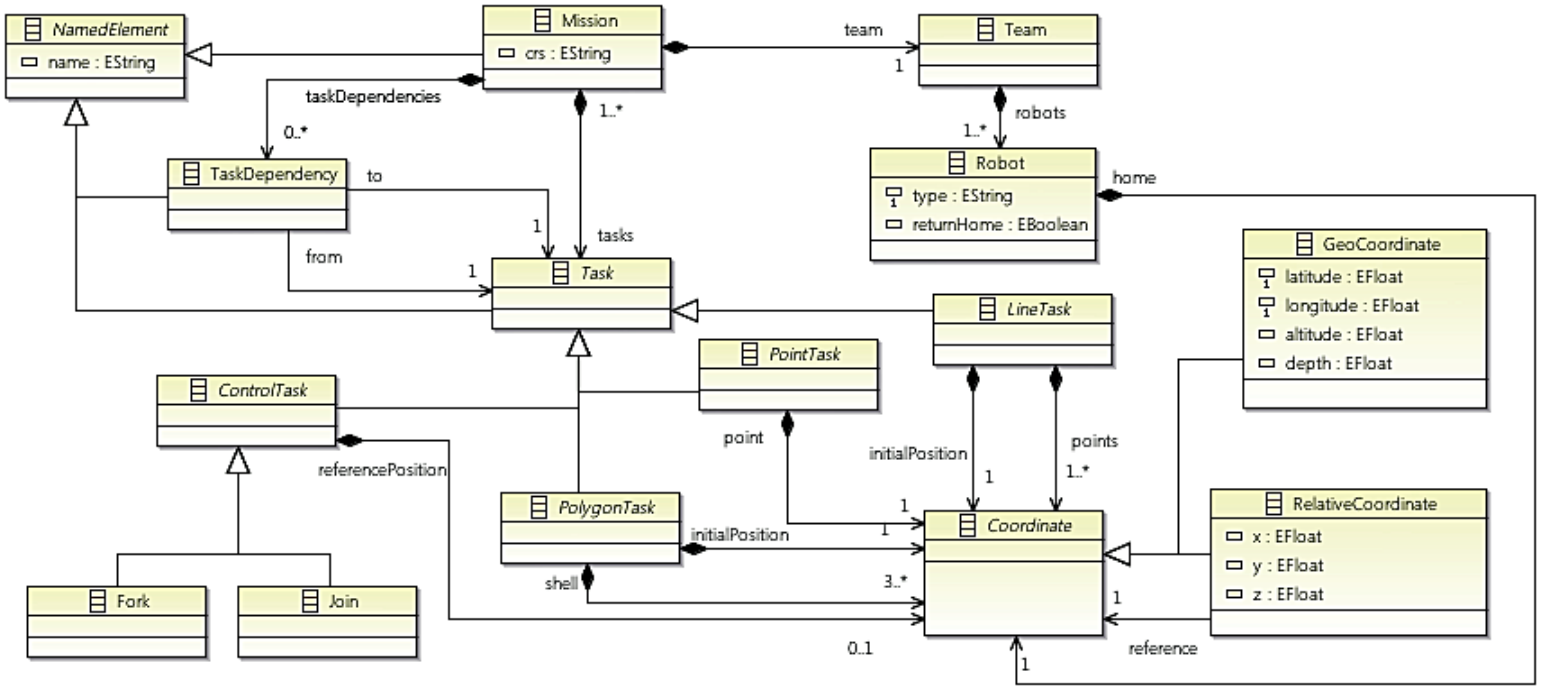


Fig. 2. Monitoring Mission Language - mission layer

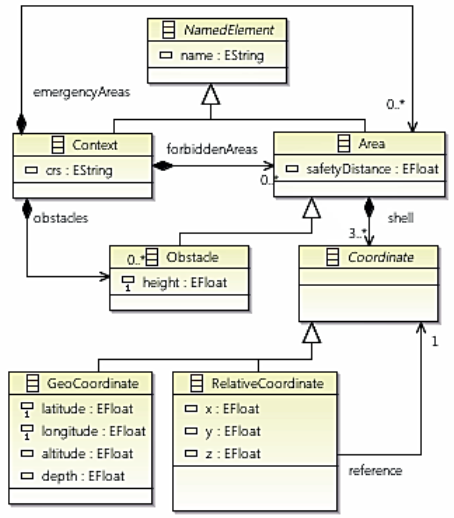


Fig. 3. Monitoring Mission Language - context layer

of the polygon representing their shape; each point of the shell is represented by its corresponding `Coordinate` object, which is the same coordinate object in the mission layer of MML. Moreover, each area has a *safetyDistance* attribute of real type representing the minimum safety distance between the robot and each side of the polygon of the area. More technically, this attribute allows the platform to replace each area in the environment by its Minkowski sum with a sphere with radius *safetyDistance* [**?**]. This operation creates larger obstacles, defined by the shadow traced as the sphere walks a loop around each area while maintaining contact with it [**?**].

The information contained into the context layer of MML plays a key role in the family because it allows potential tools to properly deduce the movements that the robots have to perform in order to both execute the missions specified in MML and satisfy possible environmental constraints (e.g., the presence of a fixed obstacle).

3.2 Robot Language (RL)

The robot language has been conceived to enable the specification of the technical characteristics of each type of robot involved in the missions (see Fig. 4). Clearly, `Robot` is the central concept of the language. The characteristics of the robot that the language permits to specify are the following:

- *name*: unique name of the type of robot being modelled;
- *onBoardObstacleAvoidance*: it permits to specify if the considered robot is endowed with mechanisms able to autonomously avoid obstacles;
- *gyro*, *gps*, *accelerometer*, *magnetometer*, *barometer*, and *pressuremeter*: they are boolean attributes used to specify the available on-board sensors of the robot being modeled;

- *communicationRange*: it is used to specify the maximum range expressed in meters of the supported radio control;
- *dataRate*: it permits to specify the data transmission rate expressed in Kbps between the robot and the control station;
- *radioFrequency*: it permits to specify the radio frequency expressed in MHz used by the robot to communicate with the control station;
- *rangingSystem*: it allows robot engineers to specify which kind of navigation/detection/ranging systems operates in the robot within the following choices: SONAR, RADAR, LIDAR, or SODAR;
- *maxPayload*: the maximum weight of the payload that the robot can carry;
- *maxOperatingPressure*: the maximum allowed pressure in which the robot can be operational;
- *hardwareConfiguration*: it contains a path in the filesystem or an URL referring to a document describing in details the hardware configuration of the robot; this document may contain information about the size, type, and number of the wheels of a ground vehicle, the size, number, and power of propellers of an underwater vehicle, etc;
- *operatingTemperature*: the interval of minimum and maximum temperatures in which the robot can be operational.

Furthermore, the `Size` metaclass is used to specify the size of the robot by means of the *width*, *length*, *height*, and *weight* attributes.

The `Device` metaclass represents additional devices owned by the robot to gather data (e.g., camera, thermal sensors) and to perform actions (e.g, lights, leds, mechanical actuators, sound emitters). Each device can have (i) a set of *properties*, each of them describing some relevant configuration of the device, and (ii) a set of *supportedActions* for describing the actions that can be performed by the device (e.g, taking a photo with a given resolution, acquiring the current carbon dioxide concentration, moving its pneumatic arm to a specified position, etc.). Each supported action has a set of `Parameters` represented by a *key* and a *description* attribute.

The *batteries* of the robot are specified in terms of their *capacity* in milliampere-hour, *voltage* in volts, and optionally *rechargeTime* in seconds. A robot can have no batteries if it is connected to a wired power source, or multiple batteries.

Robot engineers can specify a specific `MovementCapability` for each amount of weight of the payload carried by the robot. Each movement capability permits to specify the movement characteristics of the robot in terms of the following dimensions:

- *payloadWeight*: the weight of the payload that the robot carries in kilograms;
- *lifetime*: the estimated life time of the robot while carrying the payload with the actual *payloadWeight* weight;
- *maxAcceleration*: the maximum acceleration that the robot can have while carrying the specific payload;
- *speed*: the minimum and maximum speed of the robot while carrying the specific payload;
- *yaw*, *pitch*, *roll*: the movement dynamics of the robot while carrying the specific payload. More in details, these values represent the minimum and maximum angles

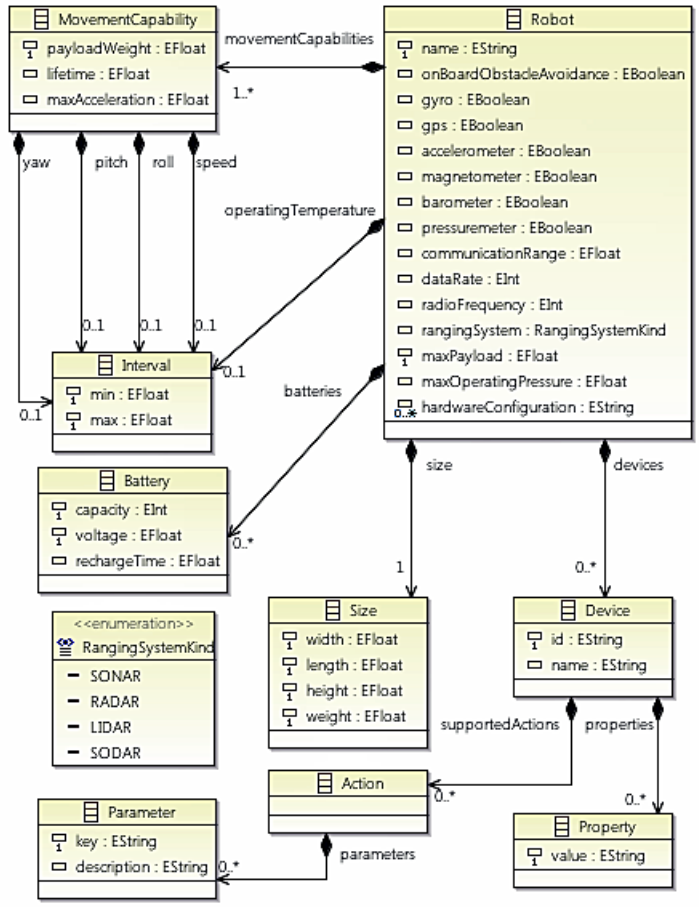


Fig. 4. The Robot Language

of rotation in the three-dimensional space with respect to the center of mass of the robot;

It is necessary to specify the various movement capabilities of the robot depending on the weight of the carried payload since their values are highly dependent on that.

Finally, it is important to note that RL models describe the *type* of each robot involved in the mission, rather than describing each specific instance of robot used; this design choice makes RL models reusable and shared across missions, projects, and organizations.

3.3 Behaviour Language (BL)

The behaviour language permits to specify the chain of atomic movements of each robot in order to perform the missions specified by means of MML specifications. Figure 5

shows the metamodel of the behaviour language, where its main metaclasses are highlighted in light red. A BL model specifies the `Behaviour` of all the robots involved in the mission; a mission can be composed of one or more *robots*, and establishes a coordinate reference system (*crs*) for all the geographic coordinates specified within the mission. Also, an instance of `Behaviour` contains a set of `Targets` that, depending on their specific subclass and involved actions, represent target elements of specific actions, e.g., the geographic coordinates to which a robot must travel to or an object that must be reached by a robot. In the latter case, the description of the object is mission- and robot-specific, and thus we decided to model it as a plain string, which can be, e.g., a URL pointing to an external model describing the object formally, a ULR pointing pointing to a concept within on ontology, an identifier to be used by some robot-specific functionality, etc.

Each `Robot` has an *id* for uniquely identifying it during the mission; the *typeName* attribute refers to the specific type of the robot, as it has been defined in the *name* attribute of a robot type in an RL model. The *travelMode* of a robot establishes the nature of the movements of a robot according to one of the following categories: $<SAFE,\ NORMAL,\ AGGRESSIVE>$; this information can be used by the controlled of a specific robot type to tune the low-level aspects related to the control of the drone, such as its reactiveness, defensiveness of the obstacle avoidance module, etc.

Each robot can have a *home*, i.e., the coordinate where the robot is initially positioned at the beginning of the mission; similarly to the MML language, each coordinate in BL can be either geographic or relative. The home coordinate can be exploited by specific behaviour model generators for implementing an automatic *come-back-home* functionality of a task in the MML model. Also, each robot contains a set of `Slots`, that are used as the primary means of synchronization between different robots involved in the mission; more specifically, a slot is an internal variable of the robot that can be either active or not active. The synchronization mechanism between robots is enacted by suitably activating the slots of specific robots via specific communication primitives (they will be described later in this section).

Each robot can perform a set of `Moves`, which are considered as the atomic movements that a robot can perform during the mission. The BL language supports the following set of movements:

- `Start`: it represents the first movement used to begin any sequence of movements, each robot behaviour can have one and only one start movement;
- `Stop`: it represents the final movement used to end any sequence of movements, each robot behaviour can have one and only one stop movement;
- `HoldPosition`: it permits to specify that the robot must keep its current position for a *duration* number of seconds;
- `HeadTo`: it represents a rotation of the robot with respect to some information inherited from the `Heading` (described later in this section).
- `GoTo`: it represents the movement towards a given *target*, which can be either a specific coordinate or an object. The result of this movement is that the robot will be at *distance* meters from the target. The orientation of the robot while performing this movement is optionally defined by means of a `Heading` object (this feature of the language is important for robots with vision and with video recording capabilities).

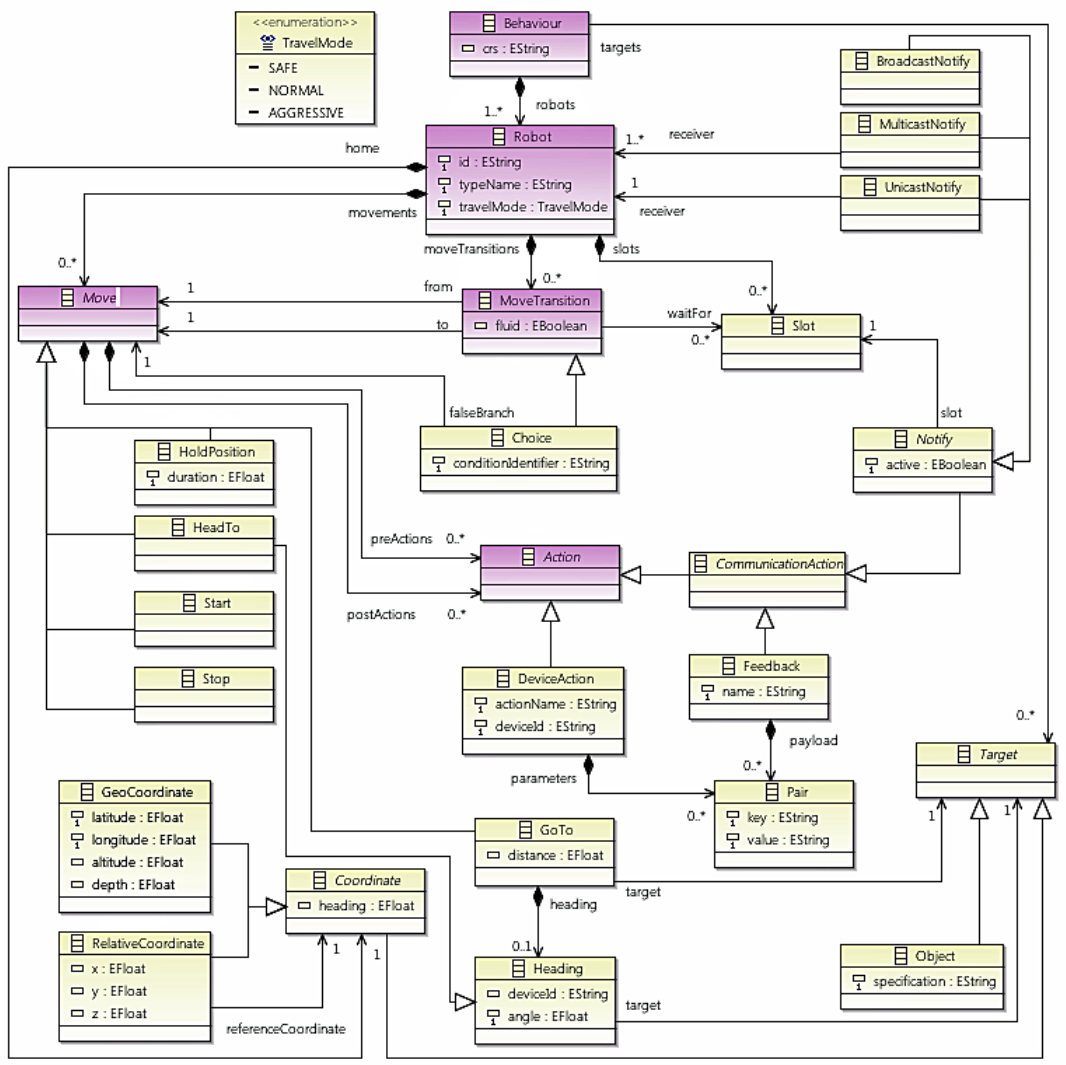


Fig. 5. The Behaviour Language

The `Heading` metaclass represents the information needed to specify the rotation of the robot. In this context, the movement of the robot is performed so that the device identified by the *deviceId* attribute is oriented towards a given *target*, that can be either a specific coordinate or an object. The value of the *deviceId* attribute refers to the *id* attribute of a `Device` in an RL model. If the *deviceId* attribute is not specified, then it is assumed that the controller of the robot has the concept of robot front. The *angle* attribute defines the angle in degrees that the device must have with respect to the target (e.g., if the angle is zero then the device is directly pointing towards the target, if the angle is 90 degrees then the device is keeping the target to its right side, etc.).

For each robot, a set of `MoveTransition` objects suitably represents the chain of all the movements that a robot must perform during the mission. So, if the robot contains a move transition $m_a \rightarrow m_b$, then after performing the m_a move, the robot will perform the m_b move; all move transitions together represent the full chain of movements of the robot during the whole mission. Moreover, if the *fluid* attribute is specified as *true* than the robot will execute the moves seamlessly, without any interruption. The triggering of a move transition can be delayed by means of a waiting mechanism. Basically, the move transition is executed only if all the slots referenced by the *waitFor* reference are active; a slot can be activated by a dedicated `Notify` communication action performed by a robot during the mission. Also, `Choice` is a special kind of move transition that represents a branch within the chain of moves of a robot. More in details, if the execution of the function represented by the *conditionIdentifier* returns $true$, than the execution of the moves chain goes on by following the move linked by the *to* reference, otherwise the execution of the moves chain goes on by following the move linked by the *falseBranch* reference. The implementation of the *conditionIdentifier* function is left to the developers of the robot controllers, the only constraint posed by the BL language is that it must return a boolean value, so that it can be correctly included in the chain of moves of the mission of the robots.

It is important to remark that a robot mission is not composed of movements only. Robots must also perform actions, such as sensing data from a CO_2 sensor, acquiring images from a mounted camera, starting to record a video, etc. In our behaviour language, a robot can perform an (ordered) set of `Actions` before and after each movement (see the *preActions* and *postActions* references in Figure 5). The actions supported in our BL metamodel are:

- `DeviceAction`: permits to specify a control action using a device mounted on the robot. To this end, the name of the action, the device that will perform the action, and its parameters are specified by means of the features *actionName*, *deviceId*, and *parameters*, respectively. These features refer to a device, an action, and its parameters supported by the robot, as defined in its corresponding RL model. Parameters are represented as key-value `Pairs`;
- `CommunicationAction`: it is used to specify the communication among the robots involved in the execution of the considered mission. In this respect, the possible actions that can be performed are the following:
  - `Feedback` represents a feedback message that a robot can send back to its controller (possibly running in a base station); the *payload* of the feedback message is composed of a set of key-value `Pairs`;

- `Notify` is a superclass representing all the possible notification actions that a robot can perform, they are: `UnicastNotify`, `MulticastNotify`, and `BroadcastNotify`. The target of a notify actions is a `Slot` of a robot, and the boolean *active* attribute defines if the slot must be activated or not;

In the next sections the languages previously presented are applied in two different application domains. In particular, the DSMLs will be applied to manage autonomous multicopters (see Section 4.1), and underwater vehicles (see Section 4.2).

4 Leveraging the DSML family for different robots

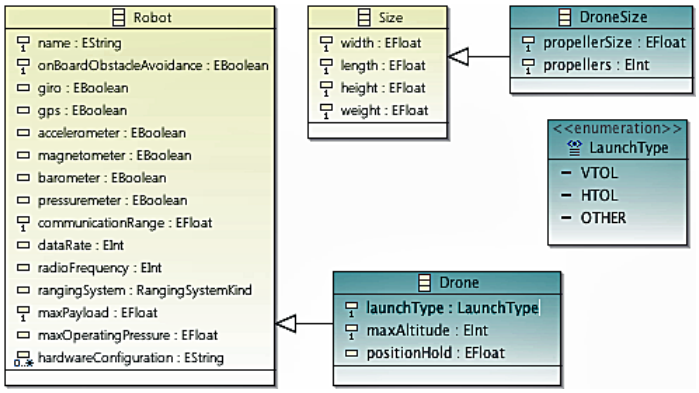
In order to assess the feasibility of our approach we leverage the DSML family for supporting two different robotic domains, namely: autonomous multicopters and autonomous underwater vehicles. Furthermore, we evaluate each language of the DSML family and we analyse it in terms of its expressiveness with respect to the peculiarities of each considered domain.

In order to avoid possible biases during the extension of the DSML family, we applied a combination of top-down and bottom-up reasoning processes when using the languages of the family in concrete scenarios. More specifically, when considering autonomous multicopters, we applied a top-down process in which we developed the software platform for executing the missions according to the extended DSML languages; conversely, when considering autonomous underwater vehicles we applied a bottom-up process in which we considered an already existing controller of an underwater system, and then we mapped the concepts in our DSML family to its main capabilities. In the following we give the details about how we applied the DSML family in the autonomous multicopters (see Section 4.1) and autonomous underwater vehicles domains (see Section 4.2).

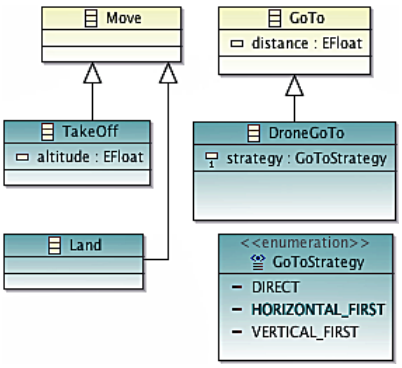
4.1 Leveraging the DSML family for autonomous multicopters

In a project in collaboration with Telecom Italia we are developing an open-source platform for the specification and execution of environmental monitoring missions. The platform is called FLYAQ [**?**] and it allows non-technical operators to straightforwardly define monitoring missions of swarms of flying drones at a high level of abstraction, thus masking the complexity of the low-level and flight dynamics-related information of the drones. More specifically, we employ multicopters, that are a special kind of unmanned aerial vehicles that take the form of multirotor helicopters that are lifted and propelled by four or more rotors [**?**]. In this section we present the instantiation of the languages proposed in Section 3 to support the specification of missions to be executed by swarms of autonomous multicopters.

Extension of the DSMLs family Interestingly, we did not need to adapt the Monitoring Mission language of the DSML family since it is still a good fit for the new application domain without any extension. Indeed, for what concerns the mission layer, from an abstract point of view a swarm of autonomous multicopters can be considered as a



(a) Extension of the robot language



(b) Extension of the behaviour language

Fig. 6. Extension of the languages of the DSMLs family for representing missions of multicopters

swarm of robots performing some tasks for fulfilling the goal of a global mission. Tasks refer to polygons, polylines, or points within a given mission environment. They may have dependencies among them (synchronization between start and end of different tasks) and in this case they are controlled by fork and join operators. Even the context layer has not been extended since its concepts are still satisfactory in the current state of the FLYAQ project. Indeed, when reasoning about the context of a mission performed by multicopters the main issues are about: the presence of obstacles (represented by the `Obstacle` metaclass in the context language metamodel), emergency landing areas (represented by the *emergencyAreas* reference), and no-fly zones where no multicopter can fly over (represented by the *forbiddenAreas* reference).

We needed to extend **Robot** language with additional concepts, specific to the nature of the managed robots (i.e., flying multicopters). Fig. 6(a) shows a fragment of the robot language metamodel focussing on the metaclasses that have been added, they are:

– `Drone` extends the `Robot` metaclass of the mission behaviour language with a set of additional attributes:
  - *launchType* represents the information about how the flying drone can be launched. The value of this attribute can be one among Vertical Take-Off and Landing (*VTOL*), Horizontal Take-Off and Landing (*HTOL*), or any other (*OTHER*);
  - *maxAltitude* represents the maximum altitude that the drone can reach when performing a mission;
  - *positionHold* represents the maximum wind speed (in meter per second) supported by the drone to maintain a given geographical location.

  The attributes specified in the `Drone` metaclass are specific to the aerial vehicles domain, and have been necessary for representing specific concepts related to this domain (e.g., *maxAltitude* and *launchType*).
– `DroneSize` extends the `Size` metaclass of the robot modelling language with two attributes for representing (i) the number of propellers of the drone, and (ii) the size of the propellers of the drone in millimeters. This two additional attributes are necessary since there is a great variability of flying drones with respect to the number and size of their propellers [**?**], and flying drones behave very differently depending on those two properties. For example, the popular AscTec Firefly[4] drone has six propellers and, by citing its official data sheet, *the redundant propulsion system enables a controlled flight even with only 5 functioning motors and actively compensates for failure*; it is difficult to imagine a drone with only four propellers providing this peculiar safety function.

Concerning the **Behaviour** language of the proposed DSML family, we needed to slightly extend it with some additional concepts, they are shown in Fig. 6(b). Firstly, we extended the set of available movements with the `TakeOff` and `Land` metaclasses, in order to directly represent these two movement types during the design of the mission. Also, we extended the `GoTo` metaclass for specifying the kind of trajectory that the multicopter must following when it needs to reach a certain point. The available trajectory kinds represent how the multicopter can reach a given point in the 3D space, they are:

– *DIRECT*, the multicopter follows a straight line between its current position and the target point;
– *HORIZONTAL_FIRST*, firstly the multicopter moves horizontally until it goes below the target point, and then it adjusts its altitude so that it reaches the target point;
– *VERTICAL_FIRST*, firstly the multicopter moves vertically until it reaches the same altitude of the target point, and then it moves horizontally until it reaches the target.

We added the above mentioned concepts to the Behaviour language in order to provide better flexibility when reasoning about the movements of the multicopters in the environment. For example, the `GoTo` movement could not be extended at all (e.g., all the multicopters always move directly to the target point), however we argue that this solution could have been too restrictive for performing real missions in practice.

[4] `http://www.asctec.de/uav-applications/research/products/asctec-firefly/`

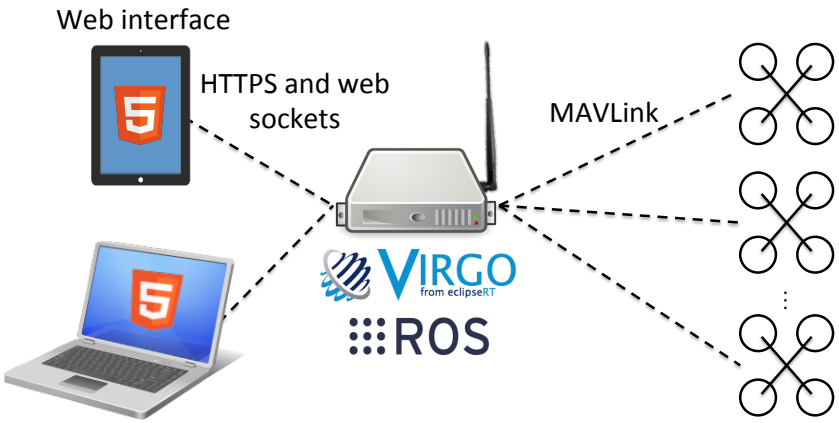


Fig. 7. Overview of the FLYAQ platform

As a matter of fact, we have been actually quite surprised when noticing that we did not have to extend at all the mission monitoring language, neither in its mission nor context layers. This result is positive since those two modeling languages have proven to be expressive enough for the needs of a concrete project. However, when reasoning at a lower level of abstraction some kind of adaptation is needed (e.g., for specifying the propeller size of a multicopter); still, the extensions performed in the Robot and Behaviour languages are not massive and, most interestingly, they did not disrupted the structure of the languages being extended.

Platform Implementation In this section we provide a description of how we developed the FLYAQ platform by taking advantage of the languages presented in the previous sections. Figure 7 gives an overview of the FLYAQ platform. On-site operators design the mission, store it, and monitor the status of ongoing missions via a standard web browser connected with the platform through a secure HTTP connection. Such a design decision enables operators to (re-)use any kind of device, such as tablets, and laptops, which are capable to run standard web browsers. Quadrotors are instructed and controlled by the platform via MAVLink communication so that radio modems can retain control up to eight miles.

More in details, the FLYAQ platform offers a web-based **graphical interface** to specify missions in the ground station at a high-level of abstraction and integrated with Open Street Map. As a matter of fact the web interface of FLYAQ is a domain-specific editor for both the mission (see Section 3.1) and context (see Section 3.1) layers of the MML language. The graphical interface of FLYAQ is implemented using HTML5, JavaScript, CSS3, and web sockets for managing real-time communications with the quadrotors (i.e., for getting the telemetry feedback from each quadrotor in the field).

The FLYAQ platform has an internal **engine** that leverages *model transformations* and *formal reasoning* to automatically transform a mission specified using the Mission

modeling language into low-level steps conforming to the Behaviour modeling language (see Section 3.3). The transformation step takes into account also a model of the context and the models of the drones that will concretely execute the mission in the field. The FLYAQ internal engine is implemented using Eclipse Virgo[5], the Eclipse Modeling Framework (EMF[6]), Java, and exposes a Rest API to the FLYAQ web interface.

A **layer of controllers** abstracts the types of the specific multicopters to all the other components of the platform; this is fundamental because it makes the platform totally agnostic of the multicopters used for executing the mission (abstraction is one of the strongest points of using MDE techniques). The layer managing the controllers is implemented using Java and ROS [**?**] and its extension rosbridge[7], a middleware communication framework specifically tailored for real-time communication with robots. Each controller can be implemented by using any kind of programming language. This is possible by exploiting the ROS communication middleware.

4.2 Leveraging the DSMLs family for autonomous underwater vehicles

At Mälardalen University, in collaboration with the Swedish Foundation for Strategic Research[8] and ABB Corporate Research, we are running the research project "Ralf3–Software for Embedded High Performance Architecture"[9]. The main demonstrator is represented by an autonomous underwater vehicle (AUV) [**?**] able to carry out unsupervised missions thanks to the exploitation of a powerful stereo vision system. As for the flying drones case, a code-centric specification of underwater missions is rather complex and requires multi-dimensional knowledge of domain, context, and navigation-related dynamics. In order to tackle such a complexity and thus allow the definition of missions at a higher level of abstraction hiding the details not directly related to the missions themselves, we want to leverage the DSMLs family and provide the needed extensions specific for underwater robots. In this section we present the extension of the languages proposed in Section 3 to support the specification of AUVs and related underwater missions.

Extension of the DSMLs family In this section we show how the languages composing the DSML family have been extended to conceive the domain of AUVs for the definition of underwater monitoring missions. In this context, the first step consisted into analysing the expressiveness of the languages composing the DSML family in order to identify whether, and eventually where, extensions were needed. In order to do so, we reverse-engineered our existing AUV, that participated to the AUVSI Foundation and ONR's International RoboSub Competition[10] in both 2013 and 2014, trying to model it using the DSML family.

[5] http://www.eclipse.org/virgo
[6] http://www.eclipse.org/modeling/emf
[7] http://robotwebtools.org
[8] http://www.stratresearch.se/en/
[9] http://www.mrtc.mdh.se/projects/ralf3/
[10] http://www.auvsifoundation.org/competitions/robosub/past-robosub-competitions

Thanks to the generality of the **Context** language, we were able to model the operating environment of the AUV without having to extend the language. In fact, since in the context of AUVs GPS navigation is basically exploitable only when floating or used for initialising inertial navigation (typical AUV's navigation), the possibility to define specific areas and obstacles position in terms of both geographical (absolute) and relative coordinates permitted us to model the AUV's context.

Also when it comes to the **Mission** language, we found that its basic definition was expressive enough to allow the definition of missions for AUVs. This was, once again, mainly possible thanks to the possibility of defining both absolute and relative coordinates, but even as a result of the ability to define important decisional control steps (such as forks and joins) that could be driven by, e.g., the AUV's stereo vision system.

The possibility to specify mixed absolute and relative coordinates as well as the ability to define targets as objects to be identified by, e.g., the stereo vision system represents the reason why no extensions were needed for the **Behaviour** language in order to model AUV's behaviours. The only base language we had to modify in order to ex-

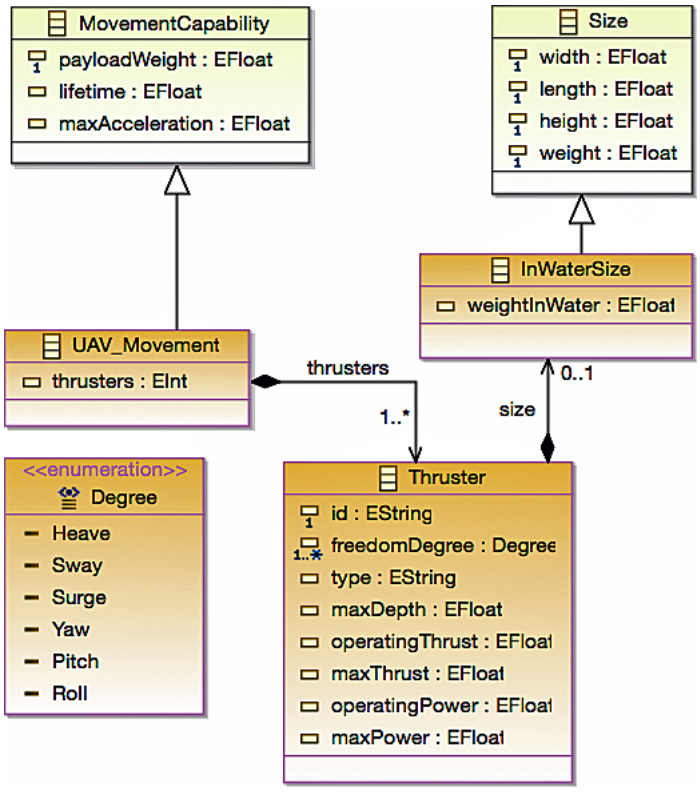


Fig. 8. Robot language extension for underwater vehicles

press characteristics mainly peculiar to AUVs was the **Robot** language. In Figure 8 we depict a sub-portion of the metamodel defining the Robot language, with specific focus on the metaclasses added for entailing AUVs. More specifically, the added concepts are:

- `Degree`: defined as an enumeration, this concept represents the six degrees of freedom (heave, sway, surge, yaw, pitch, roll) typical of an underwater vehicles;
- `InWaterSize`: due to buoyancy, a force opposite to gravity, objects immersed in water display a lower weight than outside water. In order to enable the definition of AUV's weight when immersed, we defined this metaclass, as an extension of the base concept `Size`, with the attribute *weightInWater* which would represent the object's weight when completely under water;
- `UAV_Movement`: in order to specify the different basic movements the AUV is able to perform we specialized the basic metaclass `MovementCapability` with the concept of `UAV_Movement`. This specializing metaclass owns the attribute *thrustersN* which specifies the number of thrusters the AUV is equipped with and which is reflected by the relation *thrusters* that points to the actual thruster elements (see `Thruster`);
- `Thruster`: in order to model the AUV's thrusters we defined the metaclass `Thruster`.
  - *id* represents a unique alphanumeric identifier for the specific modelled thruster;
  - *freedomDegree* indicates the non-empty set of degrees of freedom that the AUV can exploit when moving thanks to the specific thruster (sometimes in combination with other thrusters). This attribute is typed by one of the literals in the enumeration `Degree`;
  - *type* defines the thruster's type and consists of plain text;
  - *maxDepth* represents the maximum depth at which the AUV can fully operate;
  - *operatingThrust* indicates the reaction force at which the AUV can continually operate;
  - *maxThrust* indicates the peak thrust at which the AUV can operate for short periods;
  - *operatingPower* represents the power consumption of the AUV at operating thrust;
  - *maxPower* represents the maximum power consumption on the AUV when operating at peak thrust.

AUV Implementation In the flying drone instantiation (see Section 4.1) we exploited a top-down approach starting from models, defined using the DSMLs family, and refined them until code was generated within the FLYAQ platform. For the AUV, since the system and its implementation were already in place and thoroughly tested, we decided for a bottom-up approach: starting from the implementation we reverse engineered it towards the DSMLs family in order to assess capabilities of the base languages and identify needed extensions.

The underlying platform of the AUV's software is the Debian Wheezy 7.0 operating system[11] running a Mini-ITX computer. The hardware is composed of both off-

[11] `https://www.debian.org/`

the-shelf components (Mini-ITX board, sensors, actuators) and custom developed electronic boards; moreover, the system exploits a heterogeneous set of processing units, i.e. CPUs, GPUs, and FPGAs. Communication between the heterogeneous electronic boards is handled through CAN buses.

The software architecture is composed of two component-oriented layers, namely *Control layer* and *Mission layer* described below.

Control layer: it is composed of three controller components and a common API component, and it enables communication with the hardware platform, i.e., reading from sensors and managing actuators. The CAN controller component takes care of messages exchange through the common bus, while the sensor controller handles the data sent from the sensors. The actuator controller manages the sending of actions to be performed by the actuators. The API component enables access to sensor data and actuator commands to the mission layer by means of function calls. The control layer is implemented in Ada because of its I/O standard libraries, which have simplified the CAN controller development; additionally, it provides a fail-safe system with a watchdog algorithm.

Mission layer: it uses the API provided from the control layer and defines the software components necessary for complex tasks, such as object detection, object recognition, navigation, communication to the swarm, and mission execution. The mission layer consists of the following three main software components: *decision center*, *vision*, and *movement*. Decision center is the core mission component, developed in Ada, and it represents the "brains" of the AUV, where decisions are taken upon information received from the vision component about detected objects. In response to these decisions, the decision center determines the actions the AUV shall perform and defines the movements of the AUV accordingly (through the movement component). From an abstract point of view, the mission software's behaviour can be represented by a hierarchical state-machine where the upper region represents the overall mission, while composed states describe the sub-missions. The decision center is implemented in Ada.

Since the AUV is equipped with two stereo vision systems (each of which composed of two identical cameras), one on the front and one on the bottom, the vision component consists of three sub-components: bottom vision, front vision, and frame handler. Each component runs as a separate application and is implemented in C++.

The frame handler exploits the FlyCapture SDK[12] to get frames from both cameras of a stereo vision system and then forwards the appropriate frames to the bottom and forward vision components. An initial frame manipulation is performed by a set of desired brightness, resolution, and other characteristics related to vision that can vary a lot depending on in-water depth, water cleanliness, and weather conditions.

Image frames are transferred from the frame handler to the front and bottom vision applications via shared memory for increasing performance. The applications analyze the image frames to detect objects of interest and send the resulting information back to the decision center via UDP sockets.

For real-time object detection and recognition, including color-space conversion, morphological transformations, Hough transformations, Canny edge detection, we leverage C/C++ libraries for OpenCV [**?**].

[12] `http://www.ptgrey.com/flycapture-sdk`

The movement component is implemented in Ada and provides movement primitives and complex navigational functions. An example of a complex behavior is the alignment to an object, where the movement component uses a data stream from the vision component with position information about the detected object and uses this data to change its position accordingly.

5 Related work

Over the last years the adoption of MDE for developing complex robotic systems is getting more and more traction [**?**,**?**]. As an indicator of this trend, the international workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob) has been organized in the last five years and focuses exclusively on DSLs for modelling robotic systems. In particular, several DSLs have been proposed to deal with a number of issues, such as for producing and analysing behaviour descriptions at a higher level of abstraction, to enable property verification (e.g., safety and performance), for performing simulations, and for generating implementation code. A recent and thorough survey of the available domain-specific modeling languages in robotics is presented in [**?**]. In the following we make an overview of the approaches that are more similar to ours in terms of the proposed modelling concepts, their overall features, and used technologies. In many papers the adoption of MDE for developing robot software systems has focused on control or mechanical design aspects. In [**?**] the authors go further by proposing the adoption of models to manage the complete development of robotic software systems. Similarly, in [**?**] the authors propose an approach that uses models both at design- and run-time to support robots during their decision making process. In [**?**] the author proposes an Eclipse based environment for the development of robot control systems, including generation of application code skeleton. In [**?**] the authors propose a rule-based language for specifying collaborative robot applications. The proposed techniques permit to manage the complexity of specifying collaborative behavior and of managing the communication among robot teams.

Other relevant trends in adopting MDE in the domain of robotic systems is (i) focusing on implementation code generation, and (ii) targeting robotic system programmers. For instance, in [**?**] the authors propose the adoption of MontiArc architectural language, which provides a component and connector view of the structure of the robotic systems, enriched with I/O^w automata for describing the behaviour of the various components of the system; the code generation engine targets Java and Python for deployment, EMF Ecore for the graphical editing of the architectural models, and MONA for formal verification. Also, the approach proposed in [**?**] is based on a domain-specific language for representing the mission as a set of communicating components, whose interaction is restricted by safety rules that must always hold during the mission; this approach only supports components that communicate via the topic-based publish-subscribe communication pattern, thus it is not fully platform-independent. In [**?**], four domain-specific languages are presented for representing component-based robotic systems. That paper aims at improving the life-cycle of robotics components by following the Model-Driven Architecture (MDA) methodology, more specifically: the CDSL language represents general component properties, the IDSL language defines

their interfaces, the DDSL language represents the deployment configuration, and the PDSL language represents the initialization of components' parameters.

The domain-specific language presented in [**?**] represents a robotic mission as a set of routing elements for moving the robot from some location to another; actions can be performed when the robot reaches certain waypoints, and filters. Conditional branches are used to add a certain level of flexibility to the modelled missions; actions can also be performed in parallel. The proposed DSL has a graph-based graphical concrete syntax and its concepts are quite close to those of our Behaviour Language. However the proposed approach does not provide any concept or language for further abstracting the low-level details of the mission (as we do via the MML language). As said, the main stakeholders of the above mentioned approaches are programmers/engineers, and all of them target implementation code automatic generation.

If we consider commercially available robot control systems, we can notice that interfaces for graphically specifying waypoints and issuing commands can be found in a number of products (e.g., the Phantom 2 from DJI technology[13] or the IRIS+ from 3D Robotics[14]). In a sense, those products are providing domain-specific languages for specifying the mission of their robots from an abstract point of view. Indeed, many of these products do not require the user to write the code of the mission, rather they allow users without any programming experience to control the robots in order to accomplish the modelled mission. However, this comes at the sacrifice of useful programming constructs such as conditional branches, loops, the management of multiple robots in the context of the same mission, the management of robots of different types and vendors, etc. Even worse, those approaches provide very little support for checking and enforcing non-functional qualities, such as safety, performance, etc. Our DSL family provides the needed information and modelling infrastructure for addressing all the above mentioned deficiencies.

For what concerns the activity of *mission planning and definition*, several approaches focus on the definition of (either GPS-based or vision-based) waypoints and trajectories in the real world that must be navigated by the robot in the field [**?**,**?**], but to the best of our knowledge *instruments which allow operators to define a complete mission of multiple robots are still missing*.

Differently from the approaches outlined above, our focus is on *i)* the definition of the various tasks of a civilian mission at a higher level of abstraction, *ii)* allowing operators without any programming skill to create, deploy, and execute robotic missions, and *iii)* on the automatic deduction of the behaviour of the robots that will execute the modeled missions. More specifically, our aim is to provide non-technical users with the instruments to easily define missions and execute them by means of multitudes of robots. Currently available technologies somehow permit to develop missions and control the involved robots, even though only software or control engineers and domain experts have the required knowledge and are able to use the complex tools to do so. The proposed DSMLs family allows operators to straightforwardly define monitoring missions of swarms of robots by masking all the complexity of the low-level and movement dynamics-related information of the robots.

[13] `http://www.dji.com/product/phantom-2`
[14] `https://store.3drobotics.com/products/IRIS`

6 Conclusions and Future work

In this paper we presented a family of domain-specific modeling languages for specifying civilian missions of mobile multi-robot systems. The family of languages is organized in a two-layer architecture, in which the uppermost layer contains languages for the end user, while the other layer, hidden to the user, contains a working language describing the detailed behaviour of each robot of the mission. The family of domain-specific modeling languages has been instantiated to the domain of autonomous unmanned aerial vehicles and to the domain of autonomous underwater vehicles.

The various tasks in the mission layer are based on their location of interest (i.e., points, lines, and polygons), as future work we are reasoning on how to extend the MML mission layer with timing constraints and other environmental factors, such as path crowdness, convenience (e.g., in terms of used resources), safety, etc. Also, we are planning to experiment the family of domain-specific languages to other kind of robots. This will give us the possibility to further refine the family, and to start experimenting with strategies and mechanisms for combining together different kinds of robots (e.g., ground vehicles with aerial vehicles) that collaboratively perform the same mission.

Moreover, the extension of the DSMLs family for adding new concepts related to specific domains has been realized manually, i.e., by manually extending the base metamodels of the DSMLs family in order to obtain the extended metamodels presented in Sections 4.1 and 4.2. As a future work we are planning to leverage a more systematic language extension process, with properties such as language independence, possibility to compose and decompose the involved languages, some level of automation, and so on. The authors have already worked on the topic [**?**], and the integration of a systematic mechanism for languages extension is currently being evaluated.

Also, we are analysing the current architecture of the FLYAQ platform in order to better understand how it can be refactored into a fully generic robotic platform. In this context, the resulting platform will enable future third-party software developers and researchers to reuse the generic components of the platform supporting the core of the DSMLs family. We are releasing the DSL family and the FLYAQ platform as open-source software (under the MIT license) so to stimulate reuse and collaboration within the community.

Finally, we are planning to perform a set of controlled experiments to objectively assess (i) the expressivity of the DSL family with respect to the robotic systems domain, (ii) the scalability and performance of the proposed languages architecture and generic robotic platform when dealing with a large number of robots, and (iii) the ergonomics and effectiveness of the concrete syntax and user interface provided by the generic robotic platform to in-the-field operators.

Acknowledgments

We would like to thank Roberto Antonini and Marco Gaspardone for their suggestions and useful discussions about the design of the languages. This work is partly supported by the Startup Grant of Working Capital 2012[15], Art. 10 Nationally funded by

[15] `http://www.workingcapital.telecomitalia.it/`

MIUR, Ricostruire project (RIDITT - Rete Italiana per la Diffusione dell'Innovazione e il Trasferimento Tecnologico alle imprese), Swedish Foundation for Strategic Research through the RALF3 project, Knowledge Foundation[16] through the SMARTCore project[17]. This work is also partially supported by the MIUR, prot. 2012E47TM2, Prin project IDEAS.

[16] http://www.kks.se/
[17] http://www.es.mdh.se/projects/377-SMARTCore