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

Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione

Abstract—Robots are meant to replace humans for a broad variety of everyday tasks such as environmental monitoring or patrolling large public areas for security assurance. The main focus of researchers and practitioners has been on providing tailored software and hardware solutions for very specific and often complex tasks. On one hand, these solutions show great potential and provide advanced capabilities for solving the specific task. On the other hand, the polarized attention to task-specific solutions makes them hard to reuse, customize, and combine.

In this paper we propose a family of domain-specific modeling languages for the specification of civilian missions of mobile multi-robot systems. These missions are meant to be described in terms of *models* that are (i) closer to the general problem domain, (ii) independent from the underlying technologies, (iii) ready to be analyzed, simulated, and executed, and (iv) extensible to new application domains, thus opening up the use of robots to even non-technical operators. Moreover, we show the applicability of the proposed family of languages in two real-world application domains: unmanned multicopters and autonomous underwater vehicles.

Index Terms—Domain-Specific Languages, Robotics, Model-Driven Engineering, Software Engineering.

1 Introduction

Our future everyday life will be characterised by the presence of robots which, by moving underwater, on terrain, or flying, will make everyday tasks simpler and will disclose a new world of opportunities. A mobile multi-robot system (MMRS) is a special kind of robotic system in which robots act as a team. Each robot in the team is supposed to accomplish a well defined task, thus contributing to the accomplishment of a global team mission.

Having a MMRS brings two main advantages: (i) an MMRS is able to carry out a mission faster than a single robot would; (ii) an MMRS is able to accomplish missions that require specific capabilities that cannot be found in a single robot - an MMRS can maximise the leverage on specialists designed for a very specific purpose (e.g., scout an area or pick up items).

The definition of a mission is already a complex task when only considering a single robot. This is due to the fact that many details regarding the robot need to be taken into account as well as to the fact that specific technical expertise about the static and dynamic characteristics of the robot might be required. Specifying and managing missions for MMRSs get even more complicated. That is the reason of why there is the need of powerful software engineering approaches and methodologies, specifically tailored to the development and maintenance of MMRSs.

However, at the state of the art, on-site operators must deeply know all the types of used mobile robots in terms of, e.g., flight dynamics and hardware capabilities in order to correctly operate with them. On-site operators need to simultaneously control a large number of robots during the mission execution. Moreover, professional use of robots often is realized by allocating two operators for each robot: the first controlling the movements of the robot, the second controlling the instrumentation, like photo camera and engine used to move the photo camera.

Vendors provide low-level APIs and basic primitives to program robots, thus making mission development an error-prone activity. As clearly stated in the Robotics 2020 - Multi-Annual Roadmap For Robotics in Europe[1]: "*Usually there are no system development processes (highlighted by a lack of overall architectural models and methods). This results in the need for craftsmanship in building robotic systems instead of following established engineering processes*". Moreover, tasks are very specific and this limits the possibilities for their reuse across missions and organizations. As a consequence, current approaches are affordable only for users that have a strong expertise in the dynamics and technical characteristics of the used robots.

Model-driven Engineering (MDE) [1] has established itself as a powerful way to simplify the design, implementation and execution of software systems in industry [2]; in this paper we explore its exploitation for creating robotic systems for real-world applications. MDE promotes a shift from human-oriented document-driven approaches to computer-assisted artifacts that can be programmatically read and exploited 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); the concept of DSML is particularly suited for managing variability [3], which is a typical characteristic of robotic systems. MDE is a technology which has reached mature level in other domains, but not yet in robotics. In this domain, MDE enables the development of robotic applications in terms of

- *F. Ciccozzi is with the School of Innovation, Design and Engineering (IDT), MRTC, Mälardalen University, Västerås, Sweden. E-mail: federico.ciccozzi@mdh.se*
- *D. Di Ruscio is with the Information Engineering, Computer Science and Mathematics Department, University of L'Aquila, L'Aquila, Italy E-mail: davide.diruscio@univaq.it*
- *I. Malavolta is with the Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. E-mail: i.malavolta@vu.nl*
- *P. Pelliccione is with the Department of Computer Science and Engineering, Chalmers University of Technology | University of Gothenburg, Gothenburg, Sweden E-mail: patrizio@chalmers.se*

1. http://sparc-robotics.eu/roadmap/

prescriptive models defined with concepts that are less dependent on the underlying platform and closer to the problem domain [4]. Compared to common code-centric approaches, MDE simplifies the specification, understanding, and maintenance of complex robotic problems by abstraction (via models) and automation (via model manipulations) [5].

In this paper we propose a family of DSMLs for the specification of civilian missions of MMRSs. The proposed DSMLs are organized in a layered fashion, going from languages that are mainly conceived for the end-user, that is those for describing missions and their spatial context, an intermediate language for the description of the single robot's detailed behaviour (transparent to the user), and a robot language to be used for the specification of hardware and low-level details of each robot type in the MMRS. This paper extends and refines a previous paper [6], in which we published first results in the direction of defining a family of DSMLs for specifying MMRSs. To demonstrate the applicability of the proposed DSMLs in practice, in [6] we instantiated the family of languages for the domain of civilian missions of autonomous multicopters. However, the first version of the languages presented in [6] 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.

The remainder of the paper is structured as follows. An introduction to civilian missions is given in Section 2. The architecture of the proposed family of DSMLs is described in Section 3, while their instantiation and implementation into two specific application domains is shown in Section 4. In particular, the description of their application on autonomous multicopters is provided in Section 4.1, whereas their instantiation and implementation to the domain of autonomous underwater vehicles is presented in Section 4.2. Section 5 provides a snapshot of the research literature related to the contribution brought by this paper. 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 a broad set of civilian missions have been discussed. In [7] Skrzypietz proposes a classification of civilian missions for Unmanned Aircraft Systems (UAS) by diving them into six categories:

- *Scientific Research*, such as atmospheric, geological research, forestry, but also studying hurricanes, observing volcanoes, agriculture and forestry.
- *Disaster Prevention and Management*, such as damage assessment or search for survivors after natural disasters or large vehicle crashes.
- *Homeland Security*, such as surveillance of sensitive zones, securing large public events.
- *Protection of Critical Infrastructure*, such as monitoring oil and gas pipelines, protecting maritime transportation from piracy, observing traffic flows.
- *Communications*, such as broadband communication, telecommunication relays.
- *Environmental Protection*, such as control of pollution emission, protection of water resources.

According to Washington Post[2], venture investors in the United States have drenched drone-related start-ups with $40.9 million in the period January-September 2013, which corresponds to more than double the amount for all of 2012. In addition, according to the "Unmanned Aerial Vehicle (UAV) Market (2013-2018)" market research, the global UAV Market (2013-2018) is expected to reach $8,351.1 million by 2018[3]. The reason behind these impressive numbers is the number of advantages that the use of these devices brings, like:

- *reduced costs*: civilian missions typically require 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 analyzed 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 analyzed by means of queries specifically conceived with respect to the properties to be checked. MDE discloses the possibility to systematically focus on different levels of abstractions at which all the involved stakeholders can operate with the aim of (i) boosting the quality of MMRSs in terms of e.g., reliability, safety and reusability, (ii) taming the idiosyncratic variability and complexity of MMRSs, and (iii) promoting the reuse of software and hardware components across MMRSs.

In the remainder of the paper we propose a family of DSMLs 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

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

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

TABLE 1
Features of the proposed modelling languages

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 Domain-Specific Modeling 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 composed of three different languages, presented below, each of them specifically tailored to represent a key aspect of the mission being modeled. Table 1 summarizes their key features, whereas Figure 1 represents their relationships.

The *Monitoring Mission Language* (MML) is specifically conceived to be used by domain experts. It is made of two distinct layers: *mission layer* and *context layer*. The mission layer brings the concepts which are used to specify civilian missions without referring to specific aspects like the technical characteristics of the robots that will perform the missions. Practically, this layer permits to specify missions as sequences of tasks, suitably linked together via task dependencies, forks and joins. The specification of a mission is accompanied by the definition of the spatial context in which the mission is meant to be physically executed. The context layer is defined for this purpose.

Type and configuration of the individual robots that are meant to carry out the specified mission are defined via the *Robot Language* (RL).

The *Behaviour Language* (BL) provides the means to define a specification of the atomic movements and actions of the individual robots carrying out the mission and it is transparent to the domain experts. In Fig. 1 we show the relationships among the languages: MML and BL have a reference to RL. The reasons for these references are two: (i) the mission layer of MML contains a list of the individual robots composing the MMRS, for each of them a type must be properly specified, and (ii) BL, which contains all the low-level movements and actions of the individual robots and needs to be aware of the robots' type in order to correctly instruct the robots at run-time. By referring to Table 1, the stakeholders of the DSMLs are:

1) *Operator*: the on-the-field stakeholder who specifies the mission in terms of an MML model. Possible operators could be, e.g. fire fighters, policemen, or geologists;
2) *Robot engineer*: the person who models a specific type of robot (e.g., the Phantom 2 from DJI technology[4]) via an RL model as well as develops a specific controller for instructing the robot according to the basic operations supported by BL;
3) *Platform extender*: the domain and MDE expert who is in charge of extending the mission layer of MML, 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:* mission and context layers of MML are separated so to enable the reuse of context models in different missions and across different organizations. Moreover, the mission layer is constrained by the context layer in the sense that they are placed in the same location; this brings the possibility to compose the two layers in a straightforward and automatic manner.
- *Task definitions extensibility:* in order to be as close as possible to the domain of the mission being modeled, the language underlying the MML mission layer has been designed to be extensible. More specifically, a platform extender has the possibility to extend the mission layer of MML in terms of new types of task for a particular application domain, such as monitoring, security, firefighting, just to mention a few. Extensions to the MML mission layer are meant to be performed once per application domain and reused in different missions as well as across different organizations.
- *Behaviour generation capability:* the family of languages has been defined in a way that permits the automatic generation of BL models from MML and RL models. More specifically, it is possible to (i) derive BL models which are naturally consistent to the related MML models, and (ii) relieve the operator from the intricacy of low-level details of the individual robots and their atomic actions. In other words, the control routines needed for controlling the various robots in the MMRS is dependent on BL only and can be derived by automatically generating control code from BL models or by directly executing BL models at runtime via interpretation (we do not constrain platform developers to any of those two strategies for the sake of flexibility).
- *Analysis-orientation:* correctness-related aspects such as safety and security are critical in civilian missions for MMRSs. The proposed DSMLs have been defined as

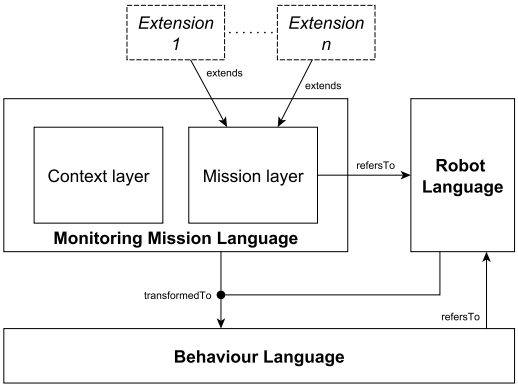


Fig. 1. The family of domain-specific modeling languages

4. http://www.dji.com/product/phantom-2

generic as possible in order to enable the specification of civilian missions from a high-level of abstraction. On one hand correctness-related aspects are not meant to be part of the DSMLs. On the other hand they are placed at an abstraction level which allows analysis tools to inspect models defined through the DSMLs for assessing safety and security aspects. BL is, according to us, the best candidate to become input artifact for this type of analysis. In fact, BL models can be pretty easily translated into corresponding well-known formats, such as state-machines, Petri nets, process algebras, for which a plethora of analysis tools are available. Additionally, BL models could be leveraged at run-time (models@runtime [8]) and then exploited by the robots for monitoring, understanding their actions, as well as for (self-)adaptation and re-configuration.

In the next sections we describe the various DSMLs in terms of their corresponding metamodel, i.e. the abstract syntax of the specific DSML. The interested reader can find the current implementation of the proposed metamodels (made publicly available as Eclipse EMF-based models) at http://cs.gssi.infn.it/files/nl.vu.ieeeaccess.zip. When it comes to the concrete syntax of the DSMLs, MML is provided with a graphical representation in the form of, e.g., an overlay on a geographical map, which represents the robot tasks, as well as the various dependencies and contextual items (refer to Section 4.1.2 for an example). BL and RL are instead given as a textual concrete syntax only, in terms of XML, since they are meant to be either created and manipulated by domain experts or automatically produced by specific software.

3.1 Monitoring Mission Language (MML)

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

3.1.1 Mission Layer

The mission layer of MML has been defined by extracting the concepts that are directly used for defining monitoring missions. As defined in MML, a monitoring mission is composed of a number of dependent Tasks that are meant to be executed by a Team of Robots. Robot has its own *home* position provided in terms of the corresponding Coordinate element (see Fig. 2). For trading off expressiveness and flexibility of the language, a coordinate can be defined as either a GeoCoordinate (geographic coordinate) or a RelativeCoordinate (relative coordinate). A geographic coordinate is defined as a geographic point identified through longitude and latitude values, in accordance to the specified coordinate reference system (see the *crs* attribute of Mission). Additionally, GeoCoordinate can provide information about the geodetic height of the coordinate through the *altitude* optional attribute, when needed. The *depth* optional attribute carries the vertical distance below a surface which can be either ground or water. A relative coordinate is represented by the distance in the three-dimensional space (x,y,z) to an origin coordinate in meters and it is specified in terms of the x, y, and z values. Basically, a relative coordinate is a point in the Cartesian space, where the origin coordinate is identified by the *reference* feature.

Apart from the ones described above, the mission layer contains three abstract task metaclasses, which represent specific types of geometric entities as follows. PointTask represents tasks referring to a specific point of the MMRS' operational environment; the coordinate of such a point are identified by the *point* reference. LineTask expresses tasks, which refer to a set of points forming a polyline in the MMRS' operational environment. More specifically, a line task is made of one initial position (*initialPosition* reference), which represents the starting point of the task, and a non-empty set of points (points reference) representing the polyline. PolygonTask represents tasks referring to specific areas in the operational environment. A polygon task is made of an initial position (*initialPosition* reference), and a set of points (shell reference) representing the borders of the area in which the task will be executed.

In addition, ControlTask is an abstract metaclass provided to represent the synchronization tasks of the mission. More specifically, any task of the modeled mission can possibly be carried out by either one or more robots as well as in sequence (see the metaclass Join) or in parallel (see the metaclass Fork) to other tasks of the same mission.

As we mentioned before, the mission layer and its related metadels are meant to be extensible since it only specifies a set of general tasks that shall be specialized to match the specific requirements of the civilian mission being modeled (see Sect. 2), as well as the types of robot that are meant to perform the mission. Extensibility and versatility of the language enable strong adherence to environmental monitoring mission domains. Imagine that an operator would need to monitor solar panel fields in a rural environment; MML could be extended by adding the concept of solar panel group, tasks for the acquisition of thermal images, tasks for the discovery of solar panel damage, and so forth.

3.1.2 Context Layer

The specification of monitoring missions includes also the description of the context where they will be executed. According to common notions of context modeling [9], the context layer addresses spatial and situational contexts. More specifically, it is used to define (i) specific portions of a geographical area that are characterized by some relevant properties, and (ii) elements that can alter the execution of the modeled mission but are not part of the mission itself. For instance, a context model may describe information about known obstacles, emergency landing spots, and forbidden flying areas.

In Figure 3 we depict the metamodel definition for the context layer of MML. A monitoring mission is meant to be executed in a Context, which consists of a number of *forbiddenAreas*, *emergencyAreas*, and *obstacles*. A forbidden area indicates a particular area in the operational environment where robots cannot access whatsoever. An emergency area is instead a safe area where robots can enter to in case of emergency situations (e.g., emerging landing). The common characteristic of forbidden and emergency areas is that they are described by their *shell*, meant as the ordered set of points of the polygon representing their shape. Points of the shell are described in terms of their corresponding Coordinate object (represented by the same metaclass as in the mission layer). Each area has a *safetyDistance* attribute 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* [10]. This operation creates larger obstacles, defined by the shadow traced as the sphere walks a loop around each area while maintaining contact with it [10].

The information contained into the context layer of MML plays a key role in the proposed family of DSMLs since it allows

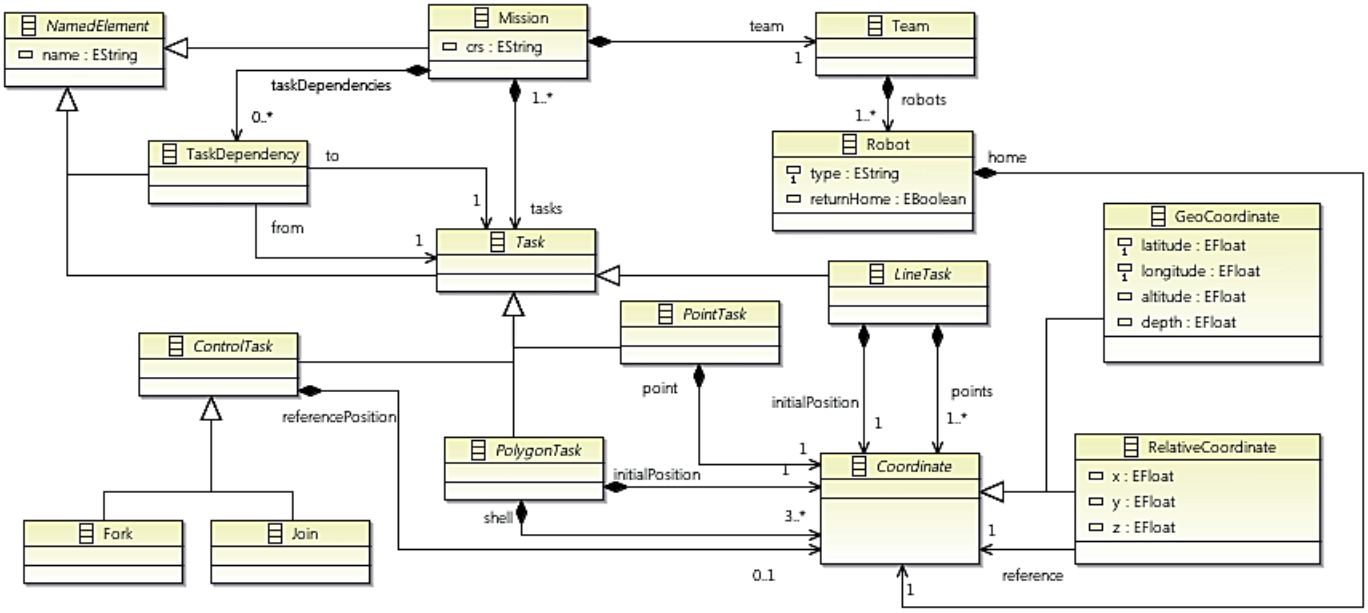


Fig. 2. Monitoring Mission Language - mission layer

potential tools to correctly derive the movements that a robot needs to make to simultaneously execute the mission and satisfy possible environmental constraints (e.g., presence of fixed obstacles).

3.2 Robot Language (RL)

RL enables the specification of the technical characteristics specific to the robot types performing in the modeled mission (see Fig. 4). As one may expect, Robot is the core concept of RL. The language provides concepts to specify the following information:

- *name*: unique name of the type of robot being modeled;
- *onBoardObstacleAvoidance*: to specify if the specific robot type is endowed with mechanisms for autonomous collision avoidance;

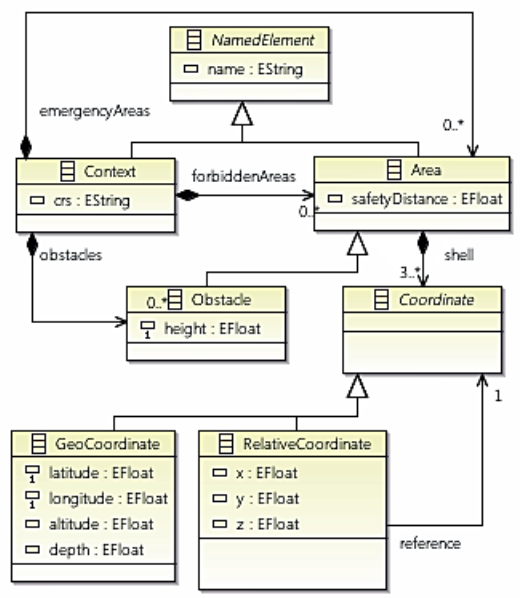


Fig. 3. Monitoring Mission Language - context layer

- *gyro*, *gps*, *accelerometer*, *magnetometer*, *barometer*, and *pressuremeter*: boolean attributes that are used to specify the sensors available on board of the robot type;
- *communicationRange*: used to indicate the maximum range of the supported radio control, expressed in meters;
- *dataRate*: used to indicate the data transmission rate between robot and control station, expressed in Kbps;
- *radioFrequency*: used to indicate the radio frequency used by the robot to communicate with the control station, expressed in MHz;
- *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.

The concept Size indicates the size of the robot in terms of its *width*, *length*, *height*, and *weight* attributes.

Device represents additional devices at the robot's disposal for gathering data (e.g., camera, thermal sensors) and for performing actions (e.g, lights, led lamps, 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,

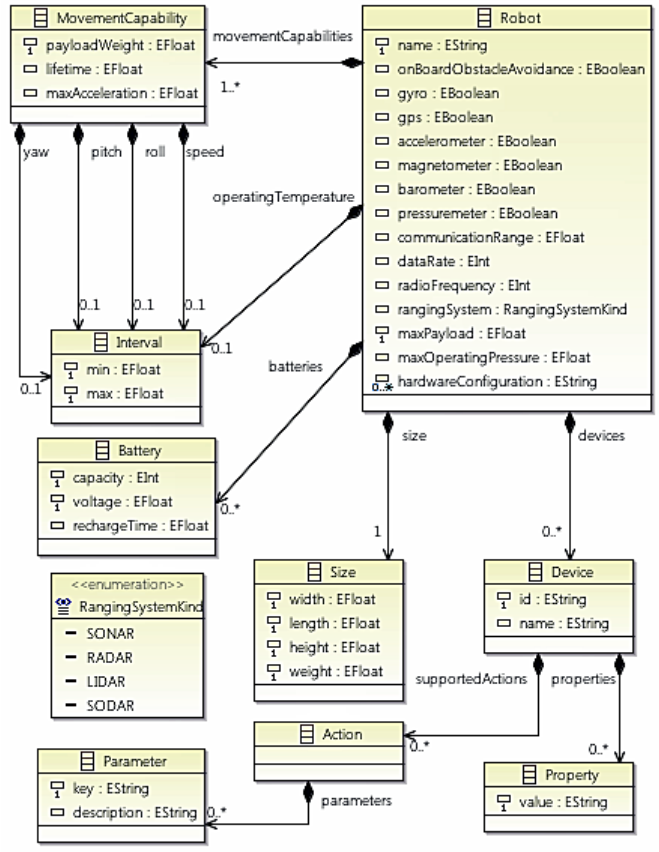


Fig. 4. The Robot Language

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 allows us to define 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* value;
- *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 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 enables the reuse and sharing of RL models in different missions, projects, and organizations.

3.3 Behaviour Language (BL)

The behaviour language permits to specify the chain of atomic movements of each individual robot in the MMRS that are needed to carry out the missions specified through MML. Figure 5 shows the metamodel of the behaviour language, where its main metaclasses are highlighted in light red. A BL model describes the Behaviour of each robot 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 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 controller of a specific robot type to tune the low-level aspects related to the control of the robot, 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 are described later in this section).

Each robot can perform a set of Moves (considered as the atomic movements) during the mission. The BL language supports the following set of movements:

- Start: the first movement used to begin any sequence of movements, each robot behaviour can have one and only one start movement;
- Stop: the final movement used to end any sequence of movements, each robot behaviour can have one and only one stop movement;
- HoldPosition: it indicates that the robot must keep its current position for a *duration* number of seconds;
- HeadTo: it indicates a rotation of the robot with respect to some information inherited from the Heading (described later in this section).

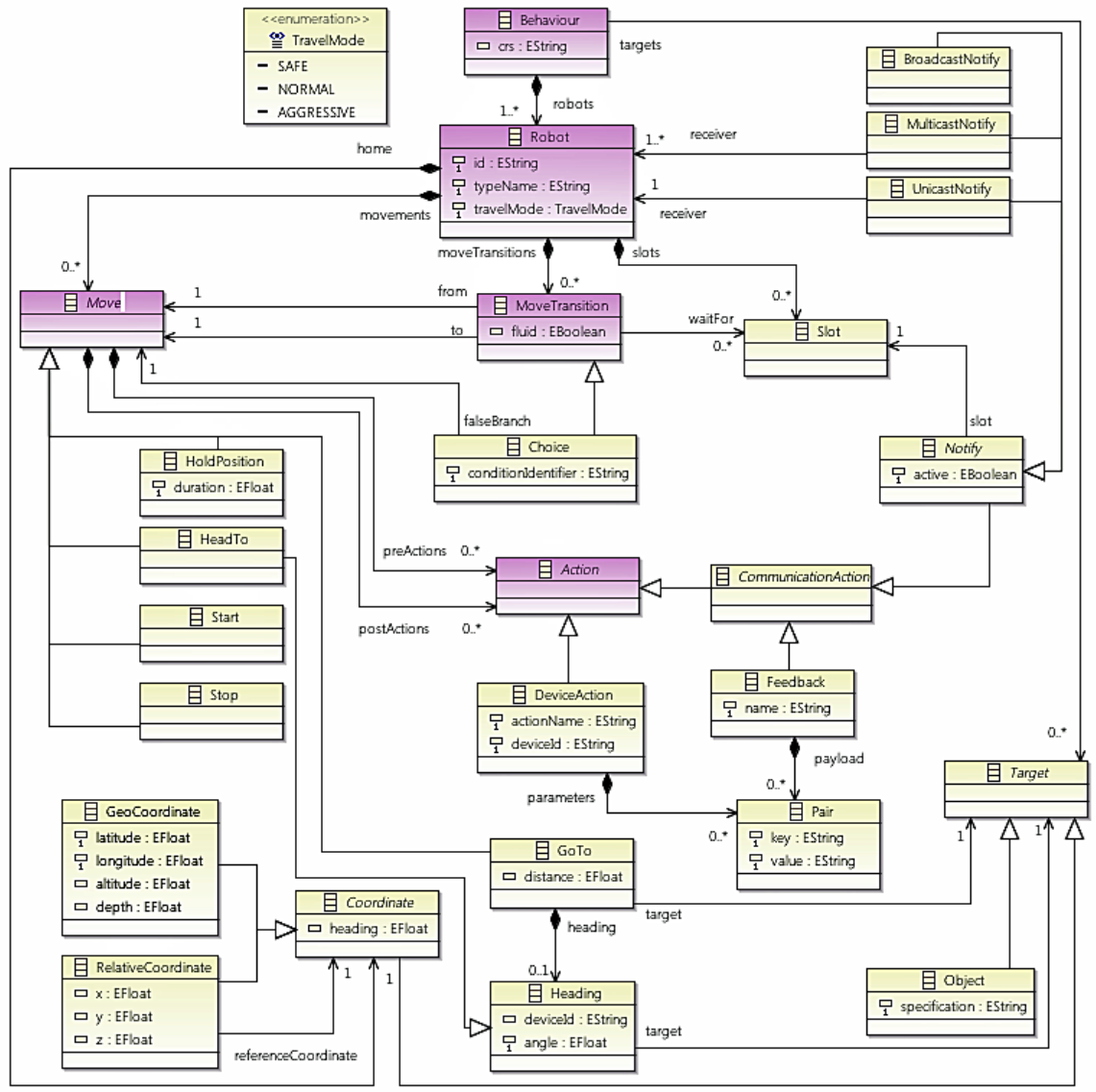


Fig. 5. The Behaviour Language

- GoTo: it describes 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).

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, when *fluid* $==$ *true* the robot would execute the modeled movements 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 $true$, then 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: used to specify a control action using a device mounted on the robot. More specifically, the name of the action, the device meant to perform the action, and the parameters of the action 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: used to describe the communication among the robots collaborating to the mission execution. The possible actions that can be performed are:
  - 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: represents a superclass of all the possible notification actions that a robot can perform. Specializations are: UnicastNotify, MulticastNotify, and BroadcastNotify. The target of 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 are applied to manage autonomous multicopters and underwater vehicles.

4 Applications of the Proposed DSML Family

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 DSMLs; 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 Autonomous multicopters

In a research project in collaboration with Telecom Italia, we are developing an open-source platform for the definition and execution of environmental monitoring missions. The platform is called FLYAQ[5] [11], [12] and its main purpose is to make it possible for non-technical operators to easily describe monitoring missions of MMRSs composed of flying drones at a high level of abstraction. FLYAQ aims in fact at hiding the complexity of the low-level and flight dynamics-related details of the employed drones.

The platform focuses on multicopters, which are a special type of unmanned aerial vehicles, shaped as multirotor helicopters, and lifted and propelled by at least four rotors [13]. In this section we show the instantiation of the family of DSMLs proposed in Section 3 to enact the specification of missions to be executed by MMRSs composed of autonomous multicopters.

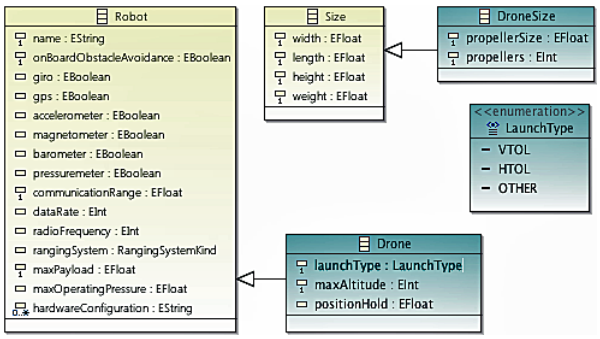
4.1.1 Extending the DSMLs for Multicopters

When extending the family of DSMLs, we noticed that we did not need to enhance MML since it fitted the new application domain as it originally was. For what concerns the mission layer, if we look at a team of autonomous multicopters from an abstract perspective, they can be generally seen as a team of robots performing tasks towards the fulfillment of a common mission goal. Tasks are defined in terms of polygons, polylines, or points within a given operational mission environment. If there are dependencies among tasks (synchronization between start and end of different tasks), they are controlled by fork and join operators. Concerning the context layer the situation was similar. When describing the context of a mission performed by multicopters, the main characteristics are those already brought by the original context layer, that is to say: possible presence of obstacles (represented by Obstacle), emergency landing areas (*emergencyAreas* reference), and no-fly zones where multicopters cannot fly over (*forbiddenAreas* reference).

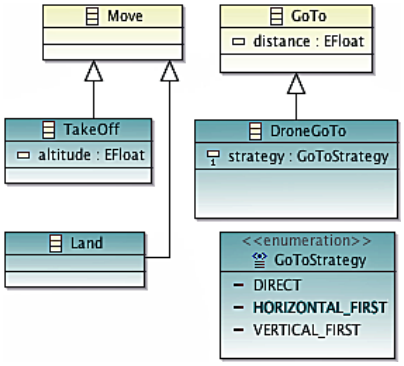
While ML did not need any enhancement, the situation for RL was different, since we had to include additional concepts which were specific to the nature of the managed robots (i.e., flying multicopters). In Fig. 6(a) we show a fragment of the RL metamodel showing the added metaclasses, which are:

- Drone: extends Robot of MML with the following set of additional attributes:
  - *launchType*: it contains the information about how the flying drone can be launched. It is an enumeration and the possible values are: Vertical Take-Off and Landing (*VTOL*), Horizontal Take-Off and Landing (*HTOL*), or other (*OTHER*);
  - *maxAltitude*: it indicates the maximum altitude reachable by the drone;
  - *positionHold*: it indicates the maximum wind speed (in meters per second) tolerated by the drone to be able to maintain a given geographical position.

5. http://www.flyaq.it/



(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

The attributes specified in the Drone metaclass are specific to the aerial vehicles domain and they are needed in order to represent domain-specific concepts (e.g., *maxAltitude* and *launchType*).

- DroneSize: it represents an extension of Size in terms of two attributes, (i) propellers, indicating the number of propellers of the drone, and (ii) propellerSize, indicating the size of the propellers of the drone in millimeters. These two attributes have been added to cope with the impressive variability of flying drones in the market with respect to the number and size of propeller properties [13], and then to cover possible diverging behaviours depending on those two properties. For example, the popular AscTec Firefly[6] 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 hard to think of any drone with four propellers providing this very specific safety function.

Concerning BL, we had to introduce some additional concepts, depicted 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. Moreover, we enhanced GoTo to enable the specification of the trajectory type that the multicopter must follow to reach a coordinate point. The possible trajectory types specify how the multicopter can reach a given point in a 3-dimensional space, and they are:

- *DIRECT*: if the multicopter can follow a straight line from its current position to the target point;
- *HORIZONTAL_FIRST*: the multicopter moves horizontally until it goes below or above the target point, which it reaches by adjusting its altitude;
- *VERTICAL_FIRST*: the multicopter moves vertically until it reaches the altitude of the target point, which it then reaches by moving horizontally.

The trajectory type concept was added to BL so to provide more flexibility with respect to allowed movements of the multicopters in the operational environment. In fact, we could have left the GoTo movement unchanged, meaning that multicopters always had to move directly to the target point. Although possible, we argue that solution would have been too restrictive for performing real-world missions.

Overall, the fact that we did not have to extend MML (neither the mission nor the context layers) was quite surprising, but at the same time it shows that the language was already expressive enough to suffice the needs of a concrete real-world project. On the other hand, some lightweight adaptation was needed (e.g., for specifying the propeller size of a multicopter) in the languages at a lower abstraction level (RL and BL). Thus, the performed extensions were not substantial and they were not disruptive neither for the structure of the two languages nor for the relationships among them.

4.1.2 FLYAQ Implementation

In this section we show how we developed the FLYAQ platform by leveraging the family of DSMLs. In Figure 7 we depict a simplified abstract view of FLYAQ (for the interested reader, additional details can be found in [11]). The main interface to the platform is web-based and it allows on-site operators to design missions, store them, and monitor their execution via a standard web browser connected with the platform through a secure HTTP connection. The design decision of leveraging web-based technology was motivated by the advantage for operators to interact with the platform with potentially any kind of device capable of running standard web browsers, from tablets to laptops, from mobile phones to powerful desktop computers. Multicopters are instructed and controlled by the platform via MAVLink communication. Doing so, radio modems are able to retain control over the communication for a distance of up to eight miles.

The FLYAQ platform come with a web-based **graphical interface** for the specification of missions in the ground station at a high-level of abstraction and seamlessly integrated with Open Street Map[7]. More specifically, the FLYAQ web-interface is conceived as a domain-specific editor for specifying both the mission (see Section 3.1.1) and the context (see Section 3.1.2) layers of MML. Such a graphical interface is developed using HTML5, JavaScript, CSS3, and web sockets for managing real-time communications with the multicopters (i.e., for getting the telemetry feedback from each multicopter in the field).

FLYAQ is equipped with an internal **engine** based on *model transformations* and *formal reasoning* to automatically derive low-level steps in terms of a BL model from a mission specified in terms of an MML model. To achieve its goal, the model transformation process takes into consideration the context defined in the MML model as well as the RL models representing the drones that are meant to perform the mission. The FLYAQ internal

6. http://www.asctec.de/uav-applications/research/products/asctec-firefly/

7. https://www.openstreetmap.org/#map=5/51.500/-0.100.

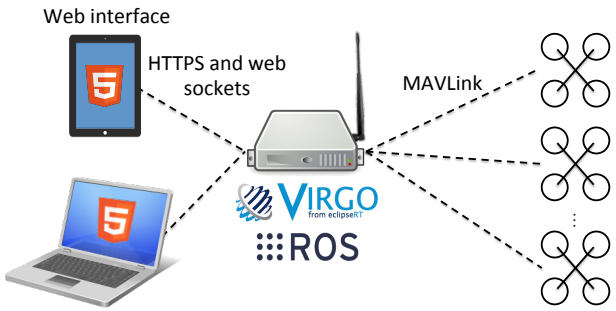


Fig. 7. Overview of the FLYAQ platform

engine is implemented using Eclipse Virgo[8], the Eclipse Modeling Framework (EMF[9]), Java, and exposes a Rest API to the FLYAQ web interface.

In addition, in order to make the platform totally agnostic of the multicopters entailed for a particular mission (abstraction is a pillar of MDE), we provide a **layer of controllers** which abstracts the specific multicopter types to all the other components of the platform. Such a layer is implemented using Java and ROS [14] as well as ROS' extension, rosbridge[10], which is a middleware communication framework specifically tailored for real-time communication with robots. The controllers can be implemented in any programming language thanks to the ROS communication middleware.

4.2 Autonomous underwater vehicles

At Mälardalen University, in collaboration with the Swedish Foundation for Strategic Research[11] and ABB Corporate Research, we are running the research project "Ralf3–Software for Embedded High Performance Architecture"[12]. The main demonstrator is represented by an autonomous underwater vehicle (AUV) [15] 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.

4.2.1 Extending the DSMLs for AUVs

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[13] in both 2013 and 2014, by modeling it by means of the DSML family.

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.

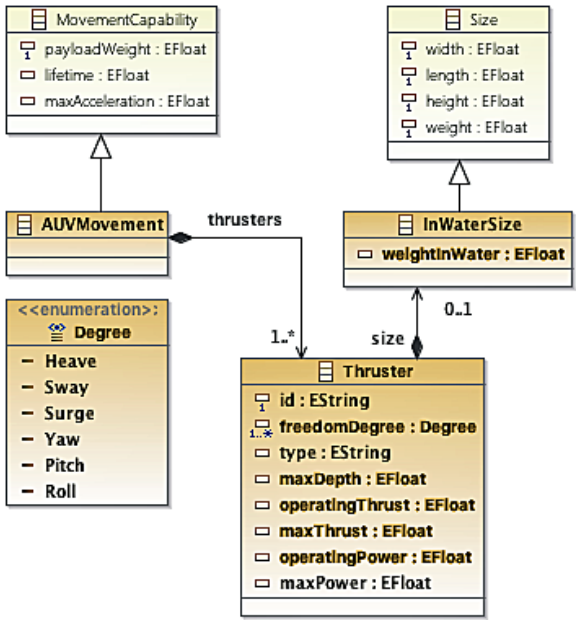


Fig. 8. Robot language extension for underwater vehicles

The only base language we had to modify in order to express 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

8. http://www.eclipse.org/virgo
9. http://www.eclipse.org/modeling/emf
10. http://robotwebtools.org
11. http://www.stratresearch.se/en/
12. http://www.mrtc.mdh.se/projects/ralf3/
13. http://www.auvsifoundation.org/competitions/robosub/past-robosub-competitions

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 represents the object's weight when completely under water;

- AUVMovement: in order to specify the different basic movements the AUV is able to perform we specialized the basic metaclass MovementCapability with the concept of AUVMovement. This specializing metaclass owns the *thrusters* reference that points to the actual thruster elements the AUV is equipped with (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.

4.2.2 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[14] running a Mini-ITX computer. The hardware is composed of both off-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 message exchanges 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[15] 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 [16].

The movement component is implemented in Ada and provides movement primitives and complex navigational functions. An example of a complex behaviour 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 [17], [18]. As

14. https://www.debian.org/

15. http://www.ptgrey.com/flycapture-sdk

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 modeling robotic systems. In particular, several DSLs have been proposed to deal with a number of issues, such as for producing and analyzing 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 [19]. In the following we make an overview of the approaches that are more similar to ours in terms of the proposed modeling concepts, their overall features, and used technologies.

In many papers the adoption of MDE for developing robot software systems has focused on control [20], [21] or mechanical [22] design aspects. In [23] authors go further by proposing the adoption of models to manage the complete development of robotic software systems. Similarly, in [24] authors propose an approach that uses models both at design- and run-time to support robots during their decision making process. In [25] the author proposes an Eclipse-based environment for the development of robot control systems, including generation of application code skeleton. In [26] a rule-based language is proposed for specifying collaborative robot applications. The proposed techniques permit to manage the complexity of specifying collaborative behaviour 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 [27] 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 [28] 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 [29], 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 [30] 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 the automatic generation of the implementation code.

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[16] or the IRIS+ from 3D Robotics[17]). 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 modeled 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 modeling infrastructure for addressing all the above mentioned deficiencies.

MMRSs are systems that operate in multiple modes and, even though the individual modes are performed linearly, the switching between them usually introduces non-linearity in its description and resolution. In the literature, control strategies for autonomous robots are modelled as Markovian jump linear systems [31], [32], especially to address fault-tolerance. In our solution we do not address the issues deriving from non-linearity in mode switching but rather focus on the linear individual modes.

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 [33], [34], 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.

6 Conclusions and Future work

In this paper we described a family of three domain-specific modeling languages tailored for the specification of civilian missions for mobile multi-robot systems. The family is structured in a two-layer fashion. In the higher (more abstract) layer we placed the two languages which are intended for the end user, while in the

16. http://www.dji.com/product/phantom-2
17. https://store.3drobotics.com/products/IRIS

lower layer we placed the language for the specification of robot's detailed low-level behaviours, which is hidden to the user. For demonstrating the applicability of the family of languages to real-world applications, we instantiated the family for specific usage in two application domains: autonomous unmanned aerial vehicles and autonomous underwater vehicles.

In this version of the mission layer of the Monitoring Mission Language tasks are defined in terms of their locations of interest (i.e., points, lines, and polygons) only. This is effective but at the same time limits the applicability, in terms of mission types, of the family of languages. As future work we plan to enhance the mission layer with timing properties and other environmental aspects (e.g., resource usage, safety, crowdedness) to broaden the application scope of the family of languages. Moreover, we have already undertaken preparation steps for experimenting the family of languages with other types of robots. This will be pivotal to fine-tune the languages as sell as to establish the basics for making different kinds of robots (e.g., ground vehicles with aerial vehicles) to actively collaborate in a mission.

It is noteworthy to stress the fact that the extension to the family for adding domain-specific concepts was realized by manually extending the metamodels related to the various languages in the family. As enhancement of our MDE approach, we are working on a more systematic and semi-automated language extension mechanism. We have already worked with similar issues [35], [36], and the integration of a systematic mechanism for languages extension is currently ongoing.

Additionally, we are investigating the current architecture of the FLYAQ platform with the goal of refactoring it into a fully generic robotic platform for mission specification. Doing so, the new platform will enable third-party software developers and researchers to reuse generic components of the platform supporting the core of the family of languages. We are currently in the process of releasing the family of languages and the FLYAQ platform as open-source software 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 family of languages 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 on-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 partially supported by the Startup Grant of Working Capital 2012[18], Art. 10 Nationally funded by 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[19] through the SMART-Core project[20]. This work is also partially supported by the MIUR, prot. 2012E47TM2, Prin project IDEAS.

18. http://www.workingcapital.telecomitalia.it/
19. http://www.kks.se/
20. http://www.es.mdh.se/projects/377-SMARTCore

References

[1] D. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb 2006.

[2] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 471–480.

[3] 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," *it-Information Technology*, vol. 57, no. 2, pp. 85–98, 2015.

[4] D. Brugali, "Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 155–166, 2015.

[5] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231146

[6] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems," in *Proceedings of the 1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014), available online at: http://ceur-ws.org/Vol-1319/*, July 21 2014., pp. 16–29.

[7] T. Skrzypietz, *Unmanned Aircraft Systems for Civilian Missions*, ser. BIGS policy paper: Brandenburgisches Institut für Gesellschaft und Sicherheit. BIGS, 2012. [Online]. Available: http://books.google.se/books?id=jQVPmwEACAAJ

[8] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas, *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*. Cham: Springer International Publishing, 2014, pp. 101–136. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08915-7_4

[9] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni, "A survey of context modelling and reasoning techniques," *Pervasive and Mobile Computing*, vol. 6, no. 2, pp. 161 – 180, 2010.

[10] S. S. Skiena, "The algorithm design manual, 1997," *Stony Brook, NY: Telos Pr*, vol. 504.

[11] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Nov 2015, pp. 801–806.

[12] D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Automatic Generation of detailed Flight Plans from High-level Mission Descriptions," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2016.

[13] H. Lim, J. Park, D. Lee, and H. J. Kim, "Build your own quadrotor: Open-source projects on unmanned aerial vehicles," *Robotics Automation Magazine, IEEE*, vol. 19, no. 3, pp. 33–45, Sept 2012.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," *ICRA workshop on open source software*, vol. 3, no. 3.2, p. 5, 2009.

[15] Y. Xu, Y. Pang, Y. Gan, and Y. Sun, "Auv-state-of-the-art and prospect," *CAAI Transactions on Intelligent Systems*, vol. 1, no. 1, pp. 9–16, 2006.

[16] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Commun. ACM*, vol. 55, no. 6, pp. 61–69, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2184319.2184337

[17] D. Brugali and P. Scandurra, "Component-based robotic engineering (part i)[tutorial]," *Robotics & Automation Magazine, IEEE*, vol. 16, no. 4, pp. 84–96, 2009.

[18] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 1, pp. 100–112, 2010.

[19] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 195–206.

[20] D. Brugali and L. Gherardi, "Hyperflex: A model driven toolchain for designing and configuring software control systems for autonomous robots," in *Robot Operating System (ROS)*. Springer, 2016, pp. 509–534.

[21] F. J. Ortiz, C. C. Insaurralde, D. Alonso, F. Sánchez, and Y. R. Petillot, "Model-driven analysis and design for software development of autonomous underwater vehicles," *Robotica*, vol. 33, no. 08, pp. 1731–1750, 2015.

[22] Y.-C. Chou, K.-J. Huang, W.-S. Yu, and P.-C. Lin, "Model-based development of leaping in a hexapod robot," *IEEE Transactions on Robotics*, vol. 31, no. 1, pp. 40–54, 2015.

[23] C. Schlegel, T. Hassler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *Advanced Robotics, 2009. ICAR 2009. International Conference on*, June 2009, pp. 1–8.

[24] A. Steck, A. Lotz, and C. Schlegel, "Model-driven engineering and runtime model-usage in service robotics," in *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, ser. GPCE '11, 2011, pp. 73–82.

[25] P. Trojanek, "Model-driven engineering approach to design and implementation of robot control system," *CoRR*, vol. abs/1302.5085, 2013.

[26] S. Gtz, M. Leuthuser, J. Reimann, J. Schroeter, C. Wende, C. Wilke, and U. Amann, "A role-based language for collaborative robot applications," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Communications in Computer and Information Science, R. Hhnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, Eds. Springer Berlin Heidelberg, 2012, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34781-8_1

[27] J. O. Ringert, B. Rumpe, and A. Wortmann, "Montiarcautomaton: Modeling architecture and behavior of robotic systems," *CoRR*, vol. abs/1409.2310, 2014. [Online]. Available: http://arxiv.org/abs/1409.2310

[28] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz, "Towards rule-based dynamic safety monitoring for mobile robots," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 207–218.

[29] A. Romero-Garcés, L. Manso, M. A. Gutierrez, R. Cintas, and P. Bustos, "Improving the lifecycle of robotics components using domain-specific languages," *CoRR*, vol. abs/1301.6022, 2013. [Online]. Available: http://arxiv.org/abs/1301.6022

[30] B. Schwartz, L. Nägele, A. Angerer, and B. A. MacDonald, "Towards a graphical language for quadrotor missions," *CoRR*, vol. abs/1412.1961, 2014. [Online]. Available: http://arxiv.org/abs/1412.1961

[31] Y. Wei, J. Qiu, H. R. Karimi, and M. Wang, "Model approximation for two-dimensional markovian jump systems with state-delays and imperfect mode information," *Multidimensional Systems and Signal Processing*, vol. 26, no. 3, pp. 575–597, 2015.

[32] Y. Wei, J. Qiu, and H. R. Karimi, "Quantized h-infinity filtering for continuous-time markovian jump systems with deficient mode information," *Asian Journal of Control*, vol. 17, no. 5, pp. 1914–1923, 2015.

[33] S. Bouabdallah and R. Siegwart, "Full control of a quadrotor," in *Intl. Conf. on Intelligent Robots and Systems*, 2007, pp. 153 –158.

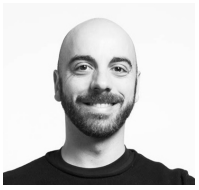
[34] F. Kendoul, Y. Zhenyu, and K. Nonami, "Embedded autopilot for accurate waypoint navigation and trajectory tracking: Application to miniature rotorcraft uavs," in *Intl. Conf. on Robotics and Automation*, may 2009, pp. 2884 –2890.

[35] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "Developing Next Generation ADLs Through MDE Techniques," in *Procs. ICSE'10*. ACM, 2010, pp. 85–94.

[36] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "Model-driven techniques to enhance architectural languages interoperability," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 26–42.

Federico Ciccozzi is Assistant Professor at Mälardalen University, Sweden, School of Innovation, Design and Engineering where he received his PhD degree in 2014. His research focuses on the definition of metamodels and model transformations for several automation aspects in the model-driven development of component-based embedded real-time systems, such as code generation, preservation of system properties, back-propagation, to mention a few. Moreover, he carries out research in the area of multi-paradigm modelling, model versioning, (co)evolution and synchronisation, as well as application of model-driven and component-based techniques to (multi-)robot systems. He has co-authored more than 35 publications in journals and international conferences and workshops in these areas. He has been serving the community as conference track and workshop organiser, expert panelist, program committee member and reviewer for conferences, workshops and international journals. In his research activity he has collaborated with several companies and research institutions such as Ericsson, ABB, Alten, Thales, CEA list, etc. He is a member of the IEEE. More information is available at http://www.es.mdh.se/staff/266-Federico_Ciccozzi.

Davide Di Ruscio is Assistant Professor at the Department of Information Engineering Computer Science and Mathematics of the University of L'Aquila. His main research interests are related to several aspects of Model Driven Engineering (MDE) including domain specific modelling languages, model transformation, model differencing, model evolution, and coupled evolution. He has published more than 80 papers in various journals, conferences and workshops on such topics. He has been co- guest editor of a number of special issues. He has been in the PC and involved in the organization of several workshops and conferences, and reviewer of many journals like IEEE Transactions on Software Engineering, Science of Computer Programming, Software and Systems Modeling, and Journal of Systems and Software. He is member of the steering committee of the International Conference on Model Transformation (ICMT), of the Software Language Engineering (SLE) conference, of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE), and of the Workshop on Modelling in Software Engineering at ICSE (MiSE). More information is available at http://www.di.univaq.it/diruscio.



Ivano Malavolta is Assistant Professor at the Vrije Universiteit Amsterdam, The Netherlands, Department of Computer Science, Faculty of Sciences. His research focuses on software architecture, model-driven engineering (MDE), and mobile-enabled systems, especially how MDE techniques can be exploited for architecting complex and mobile-enabled software systems at the right level of abstraction. Recently, he is applying empirical methods to assess practices and trends in the field of software engineering. He authored more than 50 papers in international journals and peer-reviewed international conferences proceedings; they include articles published in the IEEE Transactions on Software Engineering (TSE) and the International Conference on Software Engineering (ICSE), which are considered the leading journal and conference in the field of software engineering, respectively. He received a PhD in computer science from the University of LAquila in 2012. He is a member of ACM and IEEE. More information is available at http://www.ivanomalavolta.com.

Patrizio Pelliccione is Associate Professor at the Chalmers University of Technology and University of Gothenburg, Sweden, Department of Computer Science and Engineering and he is also Assistant Professor (on leave) at the University of L'Aquila, Italy, Department of Information Engineering, Computer Science and Mathematics. He got his PhD in 2005 at the University of L'Aquila and from February 1 2014 he is Docent in Software Engineering, title given by the University of Gothenburg. His research topics are mainly in software engineering, software architectures modelling and verification, autonomous systems, and formal methods. He has co-authored more than 100 publications in journals and international conferences and workshops in these topics. He has been on the program committees for several conferences, and is a reviewer for top journals in the software engineering domain. He is very active in European and National projects. In his research activity he has collaborated with several industries such as Volvo Cars, Volvo AB, Ericsson, Jeppesen, Axis communication, Thales Italia, Selex Marconi telecommunications, Siemens, Saab, TERMA, etc. More information is available at http://www.patriziopelliccione.com.