Journal of Systems Architecture 117 (2021) 102089

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

RoBMEX: ROS-based modelling framework for end-users and experts

Matheus Ladeira *, Yassine Ouhammou, Emmanuel Grolleau

LIAS/ISAE-ENSMA and University of Poitiers, 86960 Futuroscope, France

ARTICLE INFO

Keywords:
ROS
Model-driven engineering
Autopilot
Domain specific language

ABSTRACT

Autonomous vehicles, such as drones, are gaining great popularity due to their usability and versatility. Nowadays, a significant number of them operate using open source software, such as Robot Operating System (ROS) and the de facto standard MAVLink communication protocol, as they are free and many of them are reusable enough that they can be deployed in various different vehicles. Although these technologies offer a wide variety of resources, using them requires a reasonable background of programming and system engineering. Often, this is not achievable by common drone end-users in the short-term, as they would need to acquire a considerably large amount of know-how before working on specific domains. However, a graphical Domain Specific Modelling Language (DSML) might provide a shortcut to design drone missions using already known concepts to the end-users (or, at least, ones easier to learn). Pursuing this shortcut, RoBMEX is presented as a top-down methodology based on a set of domain specific languages able to enhance the autonomy of ROS-based systems, by allowing the creation of missions graphically, and then generating automatically executable source codes conforming to the designed missions.

1. Introduction

Unmanned vehicles, also called drones, are nowadays subject of intense studies and development due to their great potential benefits to entertainment, agriculture, construction, delivery markets and many other domains. Civilian drones may become a dominant infrastructural platform, since they are cheap, easily available and can be deployed across many industries to perform complex, expensive and dangerous tasks that humans have trouble performing [1].

Recently, several technologies have been proposed to ease the customisation of drones and especially to increase their autonomy without changing the system design. In other words, since drones may need to pass a certification process (which is a long and costly process) before they are made available for end-users, the simple orchestration of the drone-provided functions (such as "take off" and "fly to") in order to implement specific missions should not impact their certified design.[1] Considering analogous goals, several technologies in the Robotics domain aim at providing modularity and reusability to the robot's system, such as: Carnegie Mellon Robot Navigation (CARMEN) [2], Mobile and Autonomous Robotics Integration Environment (MARIE) [3], and ROS (Robot Operating System) [4]. The latter has great popularity also in the domain of drones, since it allows the expansion of their functionalities by adding to their system an algorithm running in parallel to the embedded system (on a ground station or on a companion computer on board of the drone) and that is responsible for automatising some of the pilot's tasks. Moreover, ROS works in a distributed architecture, which creates enough modularity to launch and stop functionalities at run-time without compromising the execution of the application as a whole.

Therefore, an increasing number of users have interest in implementing new and customised drone missions, so that their specific requirements are met. On the other hand, despite their considerable potential, there is still a significant cost of implementation of this technology, due to a large amount of time and effort needed for an end-user to get enough knowledge to be capable of coding the expected drone behaviours. This is where Model Driven Engineering (MDE) can act so that the implementation cost for new behaviours is reduced. MDE is a paradigm that provides tools and definitions, which bring technical and non-technical stakeholders closer together, using abstractions that allow them to better communicate with each other [5]. This is achieved by the use of models and model transformations, including automatic code generation, which often eliminates the need for end-users to deal with the code and enable them to focus on domain concepts.

Thanks to MDE capabilities, it is possible to design a set of Domain Specific Modelling Languages (DSMLs) to ease the creation of a complex mission by an end-user, who would be able to work with a more intuitive graphical language. The desired behaviour can be modelled, and the model can be used to evaluate some of the system's characteristics before its construction, as well as automatically generate

* Corresponding author.
E-mail addresses: matheus.ladeira@ensma.fr (M. Ladeira), yassine.ouhammou@ensma.fr (Y. Ouhammou), grolleau@ensma.fr (E. Grolleau).

[1] Operational certifications, nonetheless, might need review.

https://doi.org/10.1016/j.sysarc.2021.102089
Received 18 January 2020; Received in revised form 11 February 2021; Accepted 5 March 2021
Available online 10 March 2021

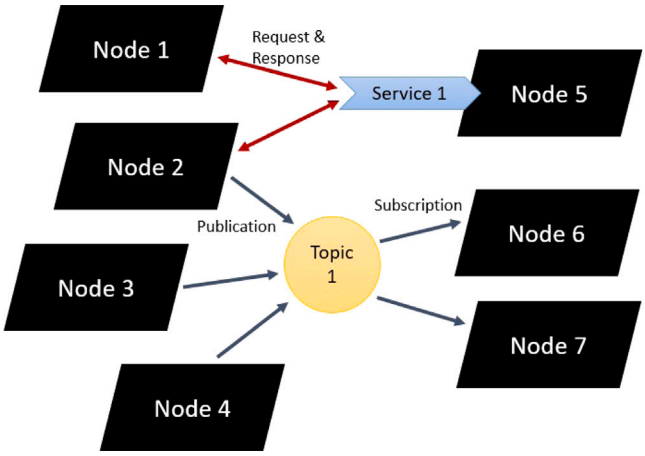

Fig. 1. Example of a ROS system containing nodes, a topic and a service.

the necessary code that implements this behaviour in a real drone. The resulting program can then be deployed as a Ground Control Station (GCS) or as a companion board, carried by the drone itself.

In this perspective, this paper presents RoBMEX: a ROS-based framework composed of a set of DSMLs dedicated to ease the specification of drone missions. RoBMEX is composed of one metamodel for ROS systems, another for general purpose operations using ROS variables, and a third one for drone missions. The generation of code was implemented for the ROS metamodel, as well as a prototype for the general purpose metamodel. A model transformation from drone missions to ROS systems is also proposed.

The rest of paper is organised as follows. Section 2 presents a background of the existing open-source technologies regarding the control of drones, as well as current possible approaches to implementing drone missions. The RoBMEX methodology and the structure overview are presented in Section 3. The foundations of RoBMEX are detailed in Section 4. A proof of concept including the tooling highlights and some illustrations of our proposal are presented in Section 5. Related works that tried to bridge the same gap are then discussed and compared to the RoBMEX approach in Section 6. Section 7 summarises and concludes this paper.

2. Background

Drones work with a control software, also called autopilot: an algorithm that contains control loops to maintain the vehicle's stability. Amongst the available open-source autopilots, Ardupilot [6] and PX4 [7] deserve attention for their wide reusability and modularity. The mentioned autopilots provide a basic set of predetermined behaviours (a.k.a. "modes") in which the vehicle can operate. Hence, they can be viewed as state machines, each state having different variables that are stabilised in specific control loops, such as the drone position, or its angular velocity.

2.1. Recent implementation technologies

To allow the autopilot to communicate with external entities, the Micro Air Vehicle Link Communication protocol (MAVLink [8]) has become a de facto standard to send and receive messages between an unmanned vehicle and a remote ground station. The protocol defines a lightweight header-only message set and structure, being open for extensions. This allows a pilot to control the drone during its flight, and also the evaluation in a ground station of some parameters.

Another widely used technology regarding drones comes from the domain of robotics. ROS (Robot Operating System) [4] framework, a middleware conceived to be used on robots, summarises most of the concepts used in robot behaviour projects, easing the development of multi-tasking applications with both synchronous and asynchronous communications.

As sketched in Fig. 1, a system running on ROS has parallel execution nodes that communicate to each other by topics (asynchronous communication) or services (loosely synchronous communication).

The technologies presented in this section allow users to configure drones so they execute planned tasks, either under the command of a pilot or in a more autonomous ways. Different approaches are possible for choosing how these technologies are going to be used in order to execute the same group of planned tasks (hereafter called "mission"), each approach with its advantages and disadvantages. The most common approaches are evaluated in the following subsection.

2.2. Drone missions

A drone mission is here considered to be a set of elements, which can be atomic (a drone behaviour, expressible in terms of a single execution task) or composed (other sets, with their respective elements). These elements must be ordered in a sequential or a parallel way, which means that each set is either parallel or sequential, but its subsets can have a different organisation.

From this definition, the simplest possibility of implementing a drone mission is to manually choose the trajectory and other activities of a drone (e.g. dumping)– choices that are made by a human operator at run-time. In this case, even if the mission has already been executed before, the operator often has to adapt its coordinates to his possibly new starting point, since these missions rely on fixed coordinates of a given global navigation system (e.g. GPS, GLONASS, BDS, etc.) as an important constitutive part. Understanding the autopilot architecture, in spite of its modularity, is time-consuming, and manual adjustments are error-prone.

If one would like to automatise the process, one should alter the autopilot code so that the drone has a "Mission-execution" mode, where the mission instructions would be embedded in the code of the autopilot. The mission would be executed when the instruction to change to this mode is received by the autopilot (from a MAVLink message, for example). This requires a considerable amount of expertise, and it requires the compilation and installation of new embedded software for every new mission that is going to be executed.

An alternative to manually changing the autopilot code is conceiving a program that is deployed in another device, such as a companion board or a GCS. This program communicates with the autopilot through some sort of bridge, like a communication protocol (e.g. MAVLink). For this, one can rely on existing application programming interfaces (APIs) to ease the design of new autonomous behaviours, such as DroneKit [9] and FlytOS [10], but learning their functions is still time-consuming and exclusively accessible to advanced developers.

Hence, there is the need for an approach that is able to face the highlighted limits and facilitate the implementation of new autonomous behaviours. The model-driven engineering (MDE) settings [5] can help to achieve this goal by working with models of drone mission concepts as first-class citizens. Models allow the system to become more effective and efficient, since they provide more objective criteria and, therefore, a more appropriate view of the system in development.

3. RoBMEX methodology

In this section, we explain what we propose as a methodology. For this, a certain scenario is considered to be true, and we created metamodels so that they can describe the missions that are instantiated in this scenario.

At first, we consider that there exists a system running ROS that is able to link a drone to a ground station in a computer, which is the case for most open source autopilots. RoBMEX must be able to create launchable node codes to work in this environment, translating

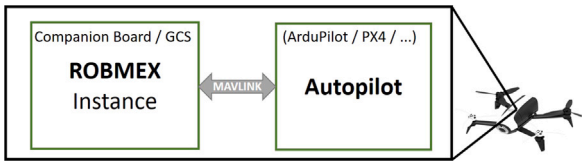

Fig. 2. Representation of a RoBMEX instance.

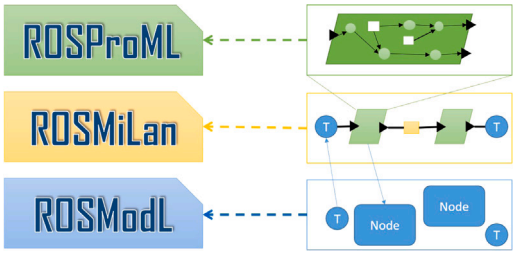

Fig. 3. The three RoBMEX DSMLs.

a specified mission using a graphical language. The RoBMEX instance will then communicate with the autopilot software through MAVLink messages, as depicted in Fig. 2.

Two main kinds of users are considered for the RoBMEX framework:

- the end-user, who has a little or no knowledge of programming, but wants to order the vehicle to do something of his/her interest;
- the programmer, who can develop functions to be used by the vehicle, but has little experience with ROS.

Both of them would have to use ROBMEX DSMLs to create instances of a ROS system. Hence, this metamodel shall be the first one created, using at least the middleware's basic structures, and being independent from any other metamodel created. With this metamodel, the developer can generate code with a well-defined structure to follow. Then, he/she would complete the missing parts with the required functionalities. In RoBMEX, this metamodel is called ROS Modelling Language (ROSModL).

For the programmer, it is interesting to be able to create simple processes, such as variable comparisons or other operations, to save them in a library. Another user, like an end-user with no expertise, would access this library to compose his/her mission. For this, common operations such as mathematical calculations, variable definitions, logical evaluations and block structures (conditionals and loops) have to be modelled. This metamodel, called ROS Process Modelling Language (ROSProML) in the RoBMEX framework, has to use some of ROS's definitions, which means that it needs to be dependent on ROSModL.

At last, composing a mission needs not only the processes built on ROS, but connections defined between their inputs and outputs. The end-user must be able to organise the whole execution of the mission, without deep knowledge regarding the system details. That is why a metamodel of a mission, based on the previous two metamodels, has to be created. It is called ROS Mission Language (ROSMiLan).

The three DSMLs interact in dynamic layers, as illustrated in Fig. 3, composing the RoBMEX framework. ROSProML, represented above the others, defines the behaviour of an execution node, the operations made with the input variables, the called functions, and the values of the outputs. RosModL, the one below, is responsible for modelling ROS systems, with its nodes and topics. ROSMiLan, between the other two, connects mission elements. For instance, a task (in a multitask mission expressed in ROSMiLan) is related to a single ROS node, and is connected to inputs and outputs that can be represented as topics in ROSModL.

4. RoBMEX foundations

Let us recall that every DSML is based on two main concepts: abstract syntax (the metamodel structure) and concrete syntax(s) which can be textual, graphic or both of them. Therefore, this section is dedicated to detail the content of each one of the three RoBMEX DSMLs' abstract syntax. The following metamodels are expressed in Ecore language [11]. It is a MOF (Meta-Object Facility) implementation inside Eclipse IDE (Integrated Development Environment).

4.1. ROS modelling language

The ROS Modelling Language (ROSModL) represents any ROS system. It focuses on ROS's key-features such as nodes, topics and services. Its metamodel is illustrated in Fig. 4.

`RosSystem` is the root class of ROSModL. Every instance of `RosSystem` has an attribute `version` indicating the ROS version the designer wants to use (e.g., Indigo, Jade or Kinetic). This is very helpful to generate adequate code. Also, every instance of `RosSystem` is composed of a set of `RosPackage` instances. Each instance has a defining name, a path to its location on the current file system, and a boolean value indicating if it already exists (true) or if it requires to be created (false). A `RosPackage` instance may be composed of instances of `Node`, `Topic`, `Service`, `MsgDataStruct`, `SrvDataStruct`, and also other instances of `RosPackage` as sub-packages.

Each instance of `Node` class has a name, an execution frequency, and may have a reference to existing code in case it is already defined in the file system. It can provide or request access to any number of instances of `Service`, and publish or subscribe to any number of instances of `Topic`. The latter is defined by a name, and relate to a single instance of `MsgDataStruct` class. `Service` class is also defined by a name and is related to a single instance of `SrvDataStruct` class. While `MsgDataStruct` class has one group of `Attribute` instances, `SrvDataStruct` class has two groups of `Attribute`: one for request and the other one for response. Instances of `Attribute` class are defined by a name and one of these types: `Simple`, with a defined ROS variable type (with or without a defined value); `Array`, with a defined size and another attribute as its variable type; or `ComposedAttribute`, containing itself another `MsgDataStruct` instance as its type.

With this defined metamodel, it is now possible to use a code generator (e.g., Acceleo [12]) to automatically generate a compilable C++ code from any ROSModL instance, since almost every information needed to generate the ROS system code is contained in the model. In fact, a transformation code has been written in Acceleo to execute this transformation. After running the transformation code directly on top of a ROSModL instance, the generated C++ code serves as a skeleton for a ROS system, containing all files necessary to compile the system, including package files. The only information this metamodel does not address is that of the core algorithm inside ROS nodes: this is taken into account by the ROSProML metamodel.

4.2. ROS process modelling language

The ROS Process Modelling Language (ROSProML) can instantiate a general process being executed in a ROS system. A process here is defined by a functionality, a set of instructions executed in order. The metamodel is presented in Fig. 5.

`Functionality` is the root class of this model. Its instances are characterised by a name and a frequency of execution. They are composed of (i) a single instance of `InternalBlock` class as its main block of code, (ii) group of instances of `DataPort` as input and output variables, (iii) a group of instances of `ServicePort` and (iv) their related instances of `ServiceBlock` class.

It is important to notice that the frequency of execution is, in this case, a reference value. In the process of generating the code,

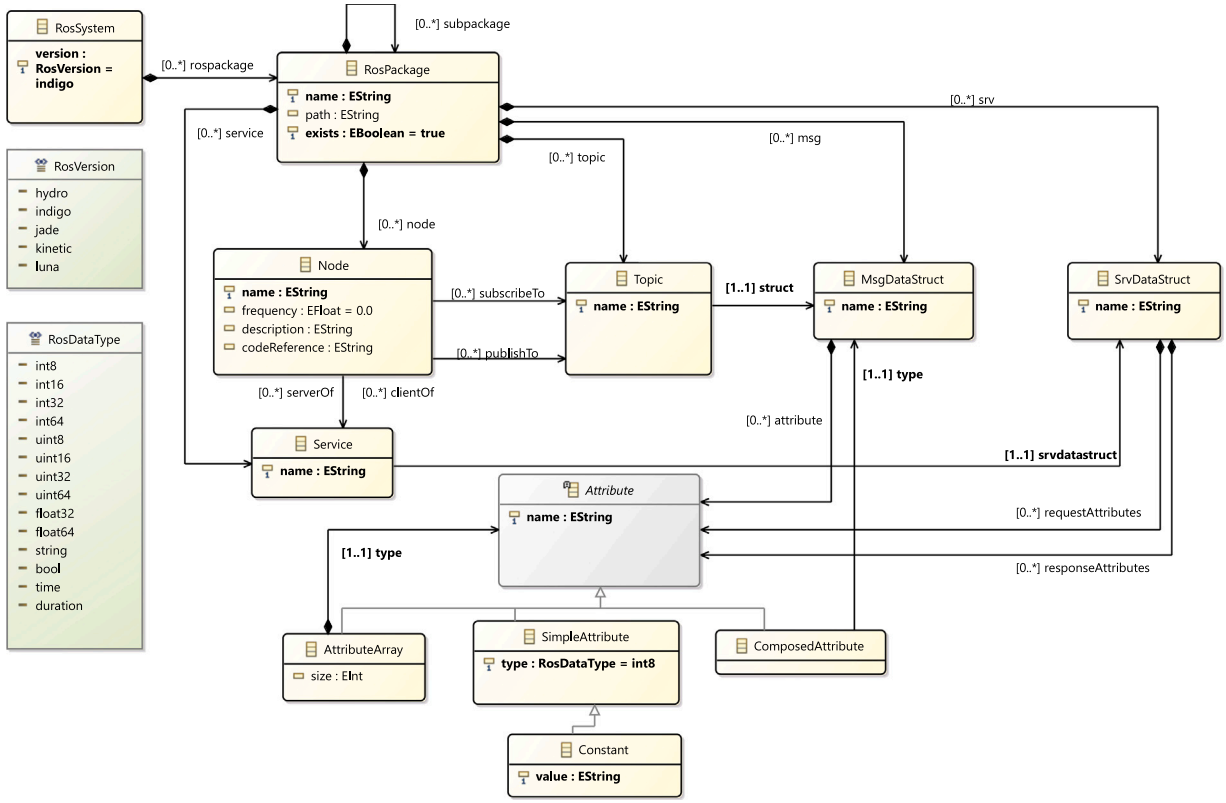

Fig. 4. ROSModL metamodel.

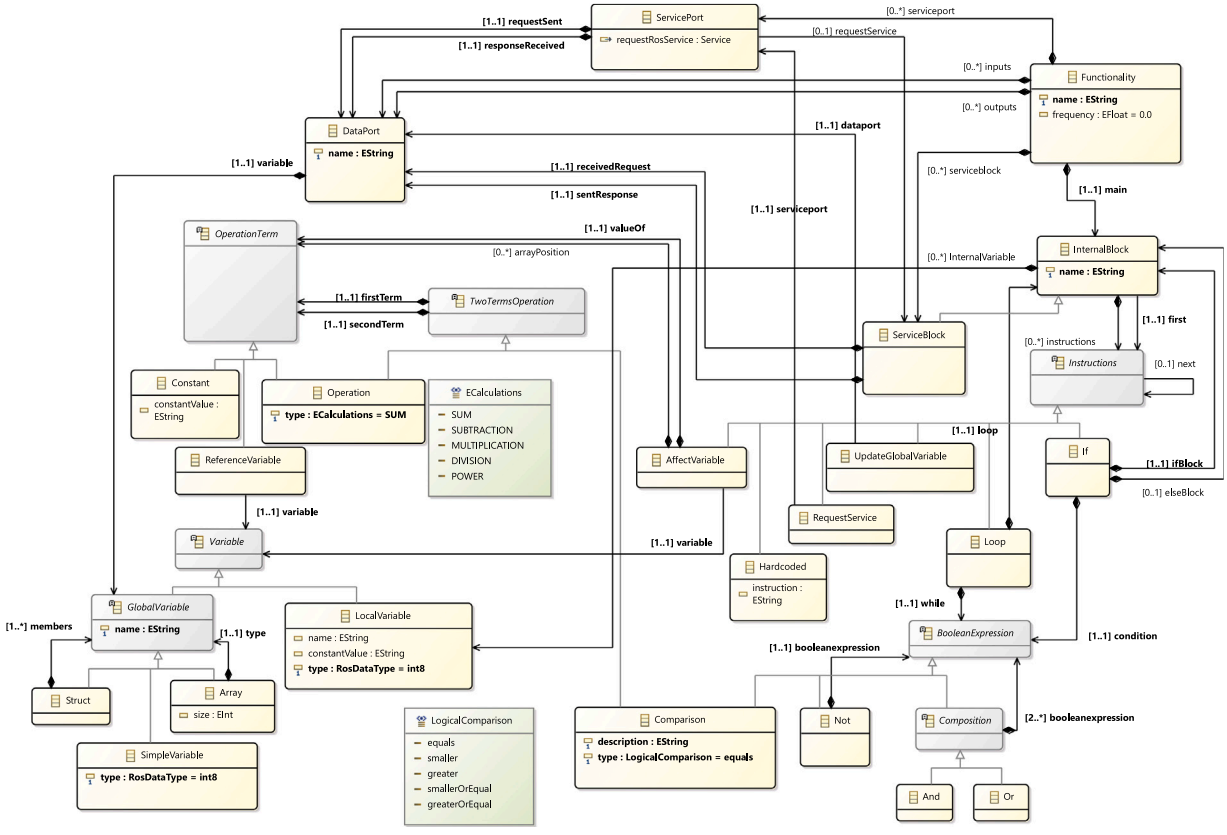

Fig. 5. ROSProML metamodel.

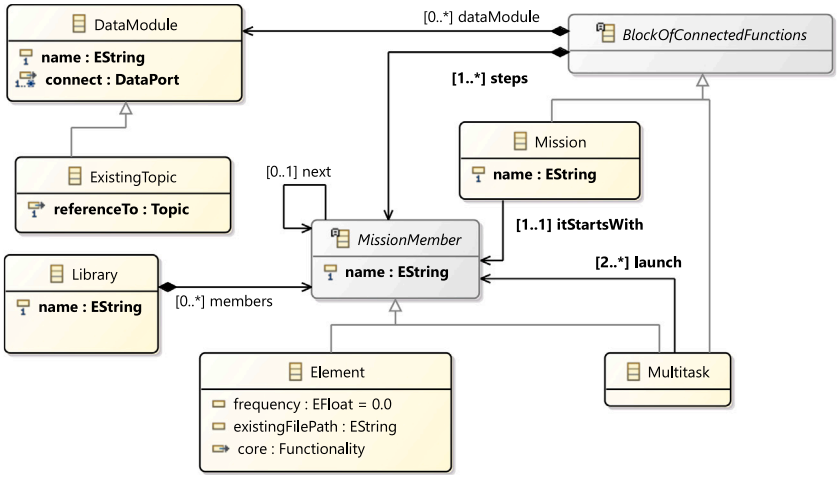

Fig. 6. ROSMiLan metamodel.

for example, a `Functionality` instance will be referenced by an instance of ROSModL, and this is the one responsible for determining the execution frequency. The lack of a defined frequency in a `Node` instance is what will make it possible for the final execution frequency to be defined by the ROSProML's class attribute.

Also, `ServiceBlock` represents the implementation of a server for a ROS service — a Client-Server kind of communication. This means that their instances will provide an algorithm that will send a response to a specific `DataPort` instance after receiving an input from another `DataPort` instance. Complementary to that, a `ServicePort` instance, acting as a client, is what will request a response from a service implemented elsewhere.

`InternalBlock` is a class representing a block of code, defined by a name, that is composed of instances of `Instructions`. `Instructions` is an abstract class, where each of its concrete implementation is related to one or more variables of some kinds, according to the role it plays. Each kind of instruction will have a specific set of relationships with different features: for example, instances of `Loop` and `If` will each relate to a boolean expression, which in this case can be either a comparison (implemented by the class `Comparison`) between existing values (of a compatible ROS type) or a composition of comparisons made by logical operators. Each `Comparison` instance relates to two `OperationTerm` instances, in other words, to a constant, a variable or an operation containing its own pair of `OperationTerm` instances. (A textual description of the comparison was introduced as a Description attribute to ease the interpretation of the instance, but it is not relevant to the model itself.) On the other hand, the class `UpgradeGlobalVariable` is connected only to `DataPort`, representing the operation of exposing a value to a port. As another example, the class `Hardcoded` represents an instruction that had to be written manually, so the designer can have the possibility to use features that are not described by the metamodel. `ServicePort` class connects some of its elements with ROSModL entities, in order to allow the connection between the two DSMLs. By using this class, a `Functionality` instance can be connected to a `Service` instance from ROSModL.

With this definition, it was possible to implement an Acceleo code prototype that transforms the instances of this metamodel into C++ code.

4.3. ROS mission language

The ROS Mission Language (ROSMiLan) metamodel, presented in Fig. 6, has its root in the class `BlockOfConnectedFunctions`. This class is abstract, and its concrete heirs are the classes `Mission` and `Multitask`. Its instances have a set of at least one `MissionMember` instance and they may have a set of `DataModules`, which connects to a `DataPort` from ROSProML, or to a topic from ROSModL.

Instances of `Mission` class are a named set of sequential members, inheriting from the class `BlockOfConnectedFunctions`, that can be executed in a ROS environment. It must specify one instance of `MissionMember` that will start the mission execution.

`DataModule` instances, responsible for transporting the values of variables from a module to another, have a defining name and must connect to at least one `DataPort` instance from ROSProML. `ExistingTopic` class inherits the characteristics of `DataModule`, but with an extra reference to a `Topic` instance from ROSModL — therefore, it is supposed to be used when one wishes to use a ROS topic already instantiated in an external system. `MissionMember` instances are defined by a name and may be of type `Element` or `Multitask`. An instance of `Element` has an execution frequency and either a core functionality or the path to an existing algorithm in the file system. This functionality is another link to ROSProML. An instance of `Multitask`, on the other hand, simply connects to at least two instances of `MissionMember` that will start executing in parallel. (This means that an instance of `Multitask` can contain other instances of `Multitask`, in a limitless recursion, which allows the instantiation of as many parallel tasks as one wishes, while keeping the possibility of using a block of multiple tasks as a whole inside a sequential structure, with other blocks being executed before and after the referred block.) `Library` instances are named sets of unordered members (instances of `Element` or `Multitask`), which serve as a repository for sharing mission components between developers.

In order to understand the essence of this metamodel, it should be said that instances of `Mission` are used to generate code. Recalling that a ROSMiLan instance connects mission elements, it has been decided that it would do so by instantiating a new ROS system, which would contain the desired mission. Then, a ROSMiLan instance could be transformed in a ROSModL instance and, from it, have its code generated. This system has to contain at least an execution node, responsible for the first sequence of elements. When these elements are structured only in a sequential manner, there is no need to use more than one node, but every `Multitask` instance will require more nodes. Considering that each `Multitask` instance refers to a number N of "children" (elements referred to by a `launch` reference), this would represent the execution of N parallel tasks if all of them are `Element` instances. In other words, for the considered model, at least N nodes have to be in execution in a given moment. To ease the implementation, the original node, containing the `Elements` that came before the `Multitask` instance, will be assigned to simply wait for the execution of all the "children" nodes to finish, so it can continue with the sequence of `Element` instances later. Hence, each `Multitask`

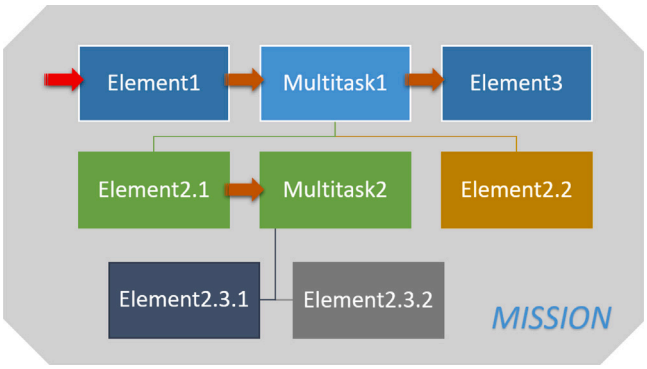

Fig. 7. Example of a ROSMiLan instance with nested Multitasks. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

instance adds N nodes to the system that already contained 1 node. To illustrate this, Fig. 7 depicts an example of a hypothetical ROSMiLan instance, where each colour represents a different execution sequence, the bright red arrow points to the first task to be executed, brown arrows to the following one, and `Multitask` instances are connected to their "children". Fig. 8 represents the corresponding ROS system, where each node appears as a rounded yellow rectangle with an arrow inside, representing its duration in time (the horizontal axis), tasks being executed are represented in blue arrows, idle tasks are represented in grey arrows, and start/ending signals are represented as vertical arrows.

4.4. Relations between metamodels

The three metamodels mentioned before work in cooperation. Not only that, but ROSProML is dependent on ROSModL because of the reference to the class `Service`, and ROSMiLan is dependent on the other two because of the references to `Topic` (ROSModL), `DataPort` and `Functionality` (ROSMiLan) classes.

Due to the different objectives and contexts of use of the dependent metamodels, they will often need to have redundant information, such as the attribute Frequency in `Node` (ROSModL), `Functionality` (ROSProML) and `Element` (ROSMiLan). In the case of a Frequency attribute, for example, although the three different attributes define the same property of the modelled system (a task's periodicity), the context in which it is explicitly defined adds some information to this definition: in ROSModL, an explicit frequency of execution is a descriptive attribute made visible mostly to allow a better system analysis; in ROSProML, it means that the modelled task must imperatively execute in the specified time constraints e.g. due to physical limitations that would destabilise the system otherwise; in ROSMiLan, the frequency attribute would be an imposition of a certain system architecture that derives, for example, of a real-time analysis of the system's software and hardware.

5. Proof of concept

This section is dedicated to give an overview of the developed tool supporting our proposal and its usage through a case study. We use it as a proof of concept, envisaging as a first step the generation of compilable (free of syntax errors) code.

5.1. Tooling

We have used the capabilities provided by Eclipse Modelling Framework [11] plugin to set up the metamodels of the three DSMLs as well as the tree editors dedicated to the instantiation. The development of another user-friendly graphical concrete syntax is in progress. This latter is based on the Sirius[2] tool. Fig. 9 shows the structure of the RoBMEX tool and the existing relationships between different components of the developed plugin, which expresses the proposed DSMLs. Furthermore, we have developed a C/C++ code generator. The implementation of this latter is based on Java programming language and Acceleo [12].

The metamodels and transformation tools are available at: https://forge.lias-lab.fr/projects/robmex.

5.2. Example

In order to demonstrate the capability of the proposed framework, a simple scenario was conceived with two variants. In this scenario, a flying drone equipped with a proximity sensor needs to execute a sequence of activities:

1. Arm the motors;
2. Takeoff;
3. Move (towards an objective);
4. Land.

This sequence is treated as the drone's mission that we want to implement, and it is represented in Fig. 10. The Move step, that takes place after the step Takeoff has finished, is itself a mission, where the drone has to execute two parallel loops: setting the direction of movement, and evaluating the proximity to any obstacle. The latter will determine, using static parameters and drivers to the sensor, if the distance to a detected obstacle is safe or not. In case it is not safe, a signal is sent to the former so that an "avoid" mechanism can take place, and the mechanism will be sustained until the received signal returns its status to "safe". The Land step will be executed after the reception of the relative command from the pilot, which will finish both the parallel tasks simultaneously.

In the first variant of the case, the mission will be composed using modules that communicate directly with the drone's drivers. In a second variant, the modules have to communicate with an autopilot that only supports MAVLink communication and hence will need a kind of translation. For this, an existing ROS system called MAVROS [13] will be used. This system is a MAVLink extendable communication node for ROS with proxy for Ground Control Station, and it is responsible of converting data from inside ROS systems (in topics and services) into MAVLink messages and send them to an autopilot software — and vice-versa. Using the ROS command `rqt_graph` after initialising MAVROS, it is possible to get a graphical representation of the MAVROS node in execution, as well as the topics and services it relates to. From an excerpt of this representation, in Fig. 11, the MAVROS node is seen as an elliptical form, while some of its topics and services are represented within rectangles. The arrows indicate the sense to which data flows (MAVROS publishes to a topic when there is an arrow going from it to the topic, and it subscribes to it whenever there is an arrow going from the topic to the node).

5.2.1. Direct communication to drivers

As a first variant, the mission is instantiated as in Fig. 12 using ROSMiLan. Each `Element` instance relates to a specific C/C++ file that contains the algorithm for the desired behaviour — either pointing to an already existing one at the location defined by its attribute "existingFilePath", or automatically generating the file in case it is non-existent. The instance of `DataModule` represents the value that is transmitted from "EvaluateDistance" to "DecideMotionDirection". The latter communicates directly with the autopilot.

[2] https://www.eclipse.org/sirius/.

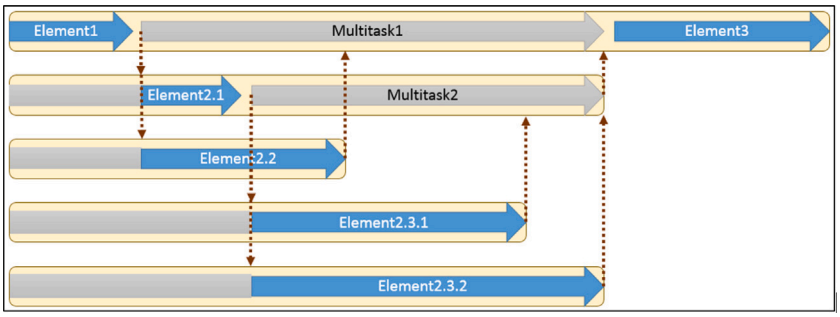

Fig. 8. Corresponding ROS system regarding the ROSMiLan example of Fig. 7. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

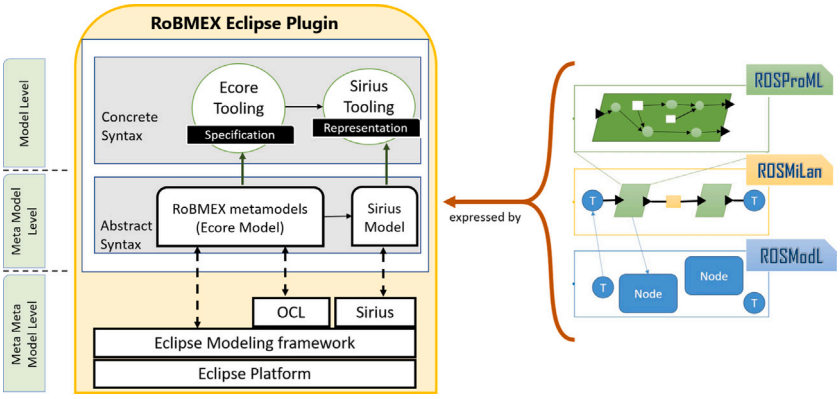

Fig. 9. Structure of the RoBMEX editor.

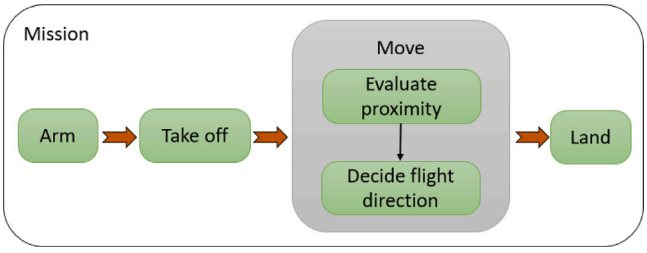

Fig. 10. Representation of the designed mission.

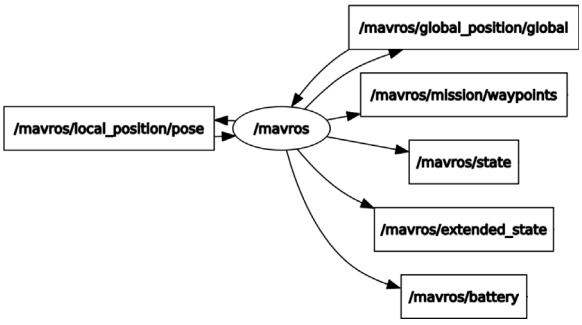

Fig. 11. Excerpt of the ensemble of MAVROS's topics and services, extracted from `rqt_graph`.

In this case, the referred C/C++ files can be created as instances of ROSProML. As an example, we instantiate the basic functionalities of the elements of Move, seen in Fig. 13 (reads distance and outputs the value of a comparison of the measured distance and an internal threshold, pointed by "DecideMotionDirection") and Fig. 14 (sends a positive or a negative value to motors depending on the value of a boolean variable, representing whether there is or not an obstacle ahead, pointed by "EvaluateDistance").

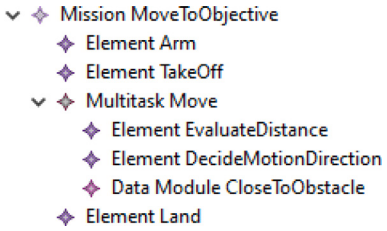

Fig. 12. ROSMiLan instance of the designed mission, with no MAVLink communication.

This ROSMiLan instance could, in theory, be transformed into a ROSModL instance according to the principles shown in Section 4. Then, a ROSMiLan instance could be transformed in a ROSModL instance and, from it, have its code generated. Using a sample of a transformation model written in Acceleo for automatically generating code, it was possible to generate compilable (i.e. absent of compilation errors) functionality codes from the ROSProML instances, as presented in Listing 1 and Listing 2.

It is clear from the generated codes that they would not work in a real environment, since publishing to the global variables was not addressed. This is purposeful, once the objective was only to do a sample of code generation: the complete version would require considerable amount of work. Hence, we chose to focus on the most basic elements of the metamodel, which excluded dealing with global variables. Nonetheless, we do not see any constraint to consider them for automatically generating code.

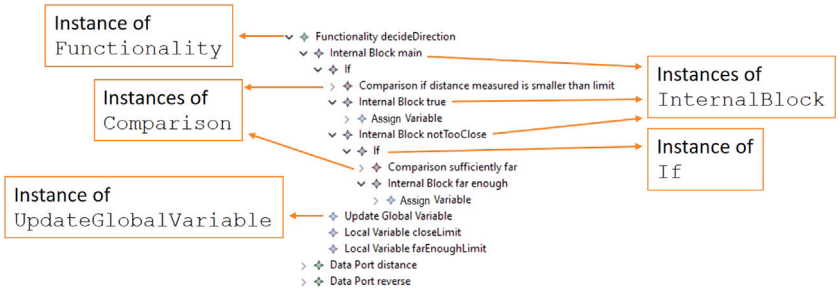

Fig. 13. Excerpt of the ROSProML instance DecideDirection.

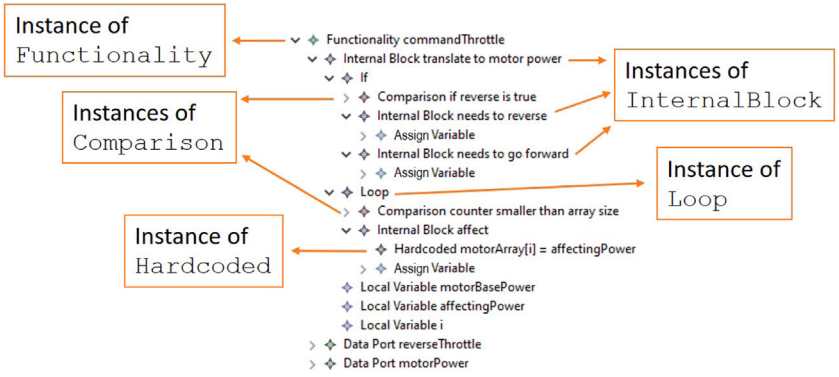

Fig. 14. Excerpt of the ROSProML instance CommandThrottle.

```
// AUTOMATICALLY GENERATED FILE FOR:
    decideDirection

// Structs:



// Global Variables:
  long double  m;
  bool  value;

void main_decideDirection() {
   short int  closeLimit  = 10 ;
   short int  farEnoughLimit  = 20 ;
   if ( (m)  <  (closeLimit) ) {
    value = true ;
   }  else {
     if ( (m)  >  (farEnoughLimit) ) {
      value = false ;
     }
   }
   /* publish to reverse */
}
```

Listing 1: Generated code for the ROSProML instance DecideDirection

```
// AUTOMATICALLY GENERATED FILE FOR:
    commandThrottle

// Structs:



// Global Variables:
  bool  reverse;
  long double  [4] motorArray;

void main_commandThrottle() {
   long double  motorBasePower  = 100 ;
   long double  affectingPower ;
   long long int  i ;
   if ( (reverse)  ==  (true) ) {
    affectingPower =  (motorBasePower)  *
    (-1)  ;
   }  else {
    affectingPower = motorBasePower ;
   }
   while ( (i)  <  (4) ) {
    motorArray[i] = affectingPower ;
    i =  (i)  +  (1)  ;
   }

}
```

Listing 2: Generated code for the ROSProML instance CommandThrottle

5.2.2. MAVROS communication

As a second variant, the same mission is instantiated but with a "DecideMotionDirection" that relates to an existing MAVROS topic, as is shown in Fig. 15. For this, MAVROS had to be instantiated so that its elements could be accessible by the model, as seen in Fig. 16.

Using the code generation algorithm developed for ROSModL, it was possible to generate a compilable (i.e. absent of compilation errors) C++ code which contains the structure needed to work with ROS, though lacking the behaviour of the generated functions, which were

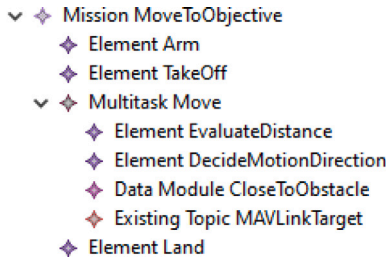

Fig. 15. ROSMiLan instance of the designed mission, with MAVLink communication through MAVROS topics.

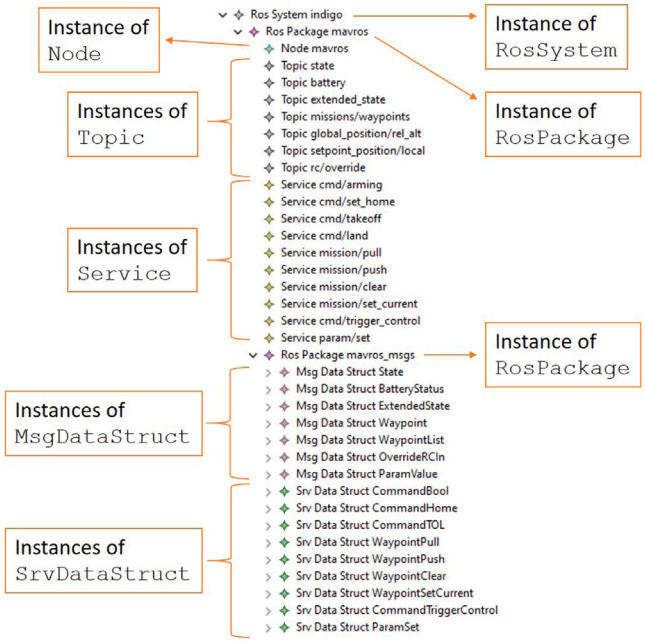

Fig. 16. Excerpt of a ROSModL instance representing MAVROS.

```
#include "mavros_msgs/WaypointSetCurrent.h"
#include "mavros_msgs/CommandBool.h"

#include "mavros/gen_node_mavros_body.h"


// Callbacks to subscribed topics:

void callback_rc_override(const mavros_msgs
    ::OverrideRCIn::ConstPtr& msg)
{
 // Code
}

void callback_setpoint_position_local(const
     geometry_msgs::Pose::ConstPtr& msg)
{
 // Code
}
```

Listing 3: Excerpt of the generated code for the ROSModL instance of MAVROS

not determined. An excerpt of the generated code can be seen in Listing 3.

6. Related work

There have been some efforts in the direction of easing the creation and execution of personalised drone missions. The following researches have brought important contributions. Yet, they differ from RoBMEX in certain key points.

RoboChart is a DSML for robotics that represents sets of state machines [14]. This modelling tool is built on formal semantics that enable the verification of the models. It provides constructs to model real-time concurrent designs and includes the notions of robotic platforms with distributed controllers, which may contain parallel threads. On the other hand, automatic code generation is not fully supported by the software, nor is its integration in ROS.

FlyAQ [15] is a tool able to automatically generate a complete Python code for executing a mission. Using two different sets of objects on a map, a mission specification (trajectories, regions of interest and activities) is merged with a context definition (geofencing and obstacles) to automatically generate a sequence of functions that complete the designated mission taking the constraints into account. Then, this sequence is automatically transformed in a python script that commands a drone fleet over a GCS using DroneKit [9], a Python implementation of MAVLink commands. It is important to note that the generated application is meant to be only executed on a GCS.

FlytOS is an operating system built on ROS (Robot Operating System) and Linux, made to facilitate the development of drones [10]. It provides APIs and SDKs for building high-level applications, and interacts with the drone/autopilot through drone-adapters, exposing the high-level APIs in ROS, C++, Python, REST and Websocket. However, it does not provide a modelling tool for the developer.

RobMoSys [16] is a project willing to create a design approach that manages the interfaces between different developer roles, thus separating concerns, in an efficient and systematic way. Its proposal envisages a set of fully model-driven methods and tools for composition-oriented engineering of robotics systems. Following this approach, SmartMDSD [17] is an IDE capable of modelling robotic systems, and automatically generating the skeletons of ROS systems that can be manually completed and compiled. Although, it cannot define the functions in the generated code, and it needs an already existing external ROS system so that the generated code can be executed.

The ReApp project [18] is an approach that uses MDE and ROS to generate and to arrange software components so that a robot behaviour can be instantiated. It is very similar to the RoBMEX approach, using different editors to different needs, but when developing software components, the code generation requires the user to manually complete the code.

SmartSoft [19,20] is a framework for using different software components to compose a robotic system, in a similar approach to that of RoBMEX, but also tools for optimising the design. They do not describe, however, any kind of DSML for creating and generating components.

RobotML [21] was part of the now finished PROTEUS project. It uses robot ontologies to provide "a common ground for designing and implementing component-based robotic systems". This approach focuses on deploying code to OROCOS-RTT5, RTMaps6, Urbi7 or Arrocam (which were defined by the Proteus project), while RoBMEX is focused on – but not limited to – ROS.

The BRIDE framework [22] is a modelling tool that generates code from an abstract representation of algorithms and from the model of a ROS system. The approach, although very close to RoBMEX's, does not use a third modelling language to link ROS models and the algorithms referred by their elements. Instead, the user has to conceive her/himself a ROS system.

Table 1 summarises the characteristics of the discussed alternatives to instantiating drone missions.

Table 1
Summary of mission implementation technologies.[a]

Ref.	Method	I.x0	D.	P.	S.C.	C.C.G.	ROS
–	Human at GCS	No	Yes	No	No	No	No
–	Rewrite autopilot	Yes	No	No	No	No	No
[14]	RoboChart	Yes	No	Yes	Yes	In dev.	No
[15]	FlyAQ	No	Yes	No	No	Yes	No
[16]	RobMoSys	Yes	Yes	Yes	Yes	No	Yes
[18]	ReApp	Yes	Yes	Yes	Yes	No	Yes
[19,20]	SmartSoft	Yes	Yes	Yes	Yes	No	Yes
[21]	RobotML	Yes	Yes	Yes	Yes	Yes	No
[22]	BRIDE	Yes	Yes	Yes	Yes	Yes	Yes
–	RoBMEX	Yes	Yes	Yes	Yes	In dev.	Yes

[a]*Ref.* stands for Reference, *I.x0* for Independent of Initial Position, *D.* for Dynamic (able to change mission at run-time), *P.* for Parallel, *S.C.* for Share Components, *C.C.G.* for Complete Code is Generation, *ROS* for Integration of ROS, and *In dev.* for In development.

7. Conclusion

In this paper, current open source technologies involving the conception and execution of drone behaviours, or missions, were discussed: ROS and MAVLink. Besides presenting a certain degree of modularity and reusability, they also present limitations, especially regarding the time and effort taken by non-experts to get the necessary know-how to use them for their purposes.

Faced with the interest in reducing these efforts, this paper proposed the RoBMEX approach, composed of three complementary DSMLs. To model ROS, we presented ROSModL, which is the basis of the framework for being capable of describing the structure of a general ROS system. Each of its execution nodes can be modelled using ROSProML, a DSML designed for algorithm experts that can represent the process of transforming input signals into the desired outputs using basic operations, function calls, comparisons and logic blocks. The connection between execution nodes can be instantiated using ROSMiLan, the DSML responsible for designing the mission itself by giving a non-expert end-user a simpler language that does not require any expertise in terms of the drone technologies.

Then, an example of a ROSModL instance was made, with a variant that is based on the existing system MAVROS and another that is completely independent of other technologies. Some of MAVROS's elements were successfully represented using ROSModL, and could then be referenced by an instance of ROSMiLan (a model of a system that intends to communicate with an executing MAVROS node). Two examples of ROSProML instances were also made based on hypothetical tasks, and had most of their code automatically generated in C/C++.

We plan to stress RoBMEX through different use cases in order to obtain a wide choices of reusable libraries and missions. In addition, we plan to align the framework to use tools of real-time analysis, which can evaluate the schedulability of the system and calculate the response time for its tasks, in the light of [23,24]. This would allow the tool to have a library of drone mission components that would also be evaluated in real-time aspects (e.g. deadlocks and missing deadlines) for validating the design as soon as it is modelled, such as in [25]. Our work can be also extended in order to design swarm missions. In this case our work should be coupled with [26] in order to take into account the collective adaptation problems especially in run-time mode. We are working on additional concrete syntax, especially the graphical ones, allowing to easily instantiate drone missions. Also, RoBMEX will be downloadable to be shared with the community under the LGPL licence [27].

Uniting the code created by the transformation over ROSModL and the one over ROSProML, it will be possible to generate a complete compilable and executable file. This shall be the mission of the code generation for ROSMiLan models, which is still in development. It is important to note that the code generation can have other targets — not only C/C++ but also Python, and if no ROSModL instance is used, other generators can be implemented to work on ROSProML and ROSMiLan instances to create code in other languages used in embedded systems, such as Ada.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is part of the COMP4DRONES project (https://www.comp4drones.eu/), which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement N. 826610

References

[1] B. Rao, A.G. Gopi, R. Maione, The societal impact of commercial drones, Technol. Soc. 45 (2016) 83–90.
[2] M. Montemerlo, N. Roy, Thrun. S., Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit, in: Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453), Vol. 3, IEEE, 2003, pp. 2436–2441.
[3] C. Cote, Y. Brosseau, D. Letourneau, C. Raïevsky, F. Michaud, Robotic software integration using marie, Int. J. Adv. Robot. Syst. 3 (1) (2006) 10.
[4] O. Robotics, ROS.org | About ROS, 2019, https://www.ros.org/about-ros/, [Online; accessed January 8, 2020].
[5] A.R. Da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Comput. Lang. Syst. Struct. 43 (2015) 139–155.
[6] ArduPilot, ArduPilot :: About, 2016, https://ardupilot.org/index.php/about, [Online; accessed December 13, 2019].
[7] Dronecode, Open source for Drones - PX4 Open Source Autopilot, 2018, https://px4.io/, [Online; accessed December 13, 2019].
[8] Dronecode, Introduction . MAVLink developer guide, 2019, https://mavlink.io/, [Online; accessed December 16, 2019].
[9] DroneKit, Dronekit, 2015, https://dronekit.io/, [Online; accessed January 9, 2020].
[10] I. FlytBase, FlytOS: Operating system for drones, 2019, https://flytbase.com/flytos/, [Online; accessed January 10, 2020].
[11] Eclipse, Eclipse modeling framework, 2019, https://www.eclipse.org/modeling/emf/, [Online; accessed December 13, 2019].
[12] I. Eclipse Foundation, Acceleo | Home, 2019, https://www.eclipse.org/acceleo/, [Online; accessed January 10, 2020].
[13] V. Ermakov, Mavros - ROS wiki, 2018, http://wiki.ros.org/mavros, [Online; accessed January 14, 2020].
[14] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, Robochart: modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. (2019) 1–53.
[15] FLYAQ, FLYAQ, 2019, http://www.flyaq.it/, [Online; accessed January 10, 2020].
[16] RobMoSys, RobMoSys - composable models and software, 2019, https://robmosys.eu/, [Online; accessed January 10, 2020].
[17] S. Dennis, L. Alex, L. Matthias, S. Christian, The SmartMDSD toolchain: An integrated MDSD workflow and integrated development environment (IDE) for robotics software, J. Softw. Eng. Robot. (2016).
[18] M. Wenger, W. Eisenmenger, G. Neugschwandtner, B. Schneider, A. Zoitl, A model based engineering tool for ros component compositioning, configuration and generation of deployment information, in: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2016, pp. 1–8.
[19] A. Lotz, J.F. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, C. Schlegel, Towards a stepwise variability management process for complex systems: A robotics perspective, in: Artificial Intelligence: Concepts, Methodologies, Tools, and Applications, IGI Global, 2017, pp. 2411–2430.
[20] 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 Robot. Autom. Mag. 22 (3) (2015) 155–166.
[21] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, M. Ziane, Robotml a domain-specific language to design, simulate and deploy robotic applications, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer, 2012, pp. 149–160.
[22] BRICS, Bride - ROS wiki, 2014, http://wiki.ros.org/bride, [Online; accessed June 22, 2020].
[23] M. Foughali, P.-E. Hladik, Bridging the gap between formal verification and schedulability analysis: The case of robotics, J. Syst. Archit. 111 (2020) 101817.

[24] D. Casini, T. Blaß, I. Lütkebohle, B.B. Brandenburg, Response-time analysis of ros 2 processing chains under reservation-based scheduling, in: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[25] M. Abbas, R. Rioboo, C.-B. Ben-Yelles, C.F. Snook, Formal modeling and verification of uml activity diagrams (uad) with focalize, J. Syst. Archit. (2020) 101911.
[26] D. Bozhinoski, D. Garlan, I. Malavolta, P. Pelliccione, Managing safety and mission completion via collective run-time adaptation, J. Syst. Archit. 95 (2019) 19–35.
[27] LIAS, Documentation - RoBMEX - Welcome to the LIAS forge, 2021, https://forge.lias-lab.fr/projects/robmex/wiki/Documentation, [Online; accessed February 4th, 2021].

Matheus Ladeira is an aerospace engineer from UFMG (Brazil), and a Ph.D. student at ISAE-ENSMA since 2019. He is currently a member of the Real-Time Embedded Systems (RTES) team inside LIAS laboratory. His current research field involves the creation of modelling tools to aid the conception, optimisation, and validation of embedded software for drone systems. His education was focused embedded software and aerospace system simulations in the industry and in academy.

Dr. Yassine Ouhammou is Associate Professor at ISAE-ENSMA and member of LIAS laboratory. His current research fields include the capitalisation of performance analysis knowledge in order to be shared and reused easily during RTES model-based design as well as conceptual modelling of optimisation methodologies dedicated to complex system deployment. He has participated in several Research & Development projects with academic and industrial partners in the area of model-based performance analysis of critical real-time systems. Recently, he has been involved in developing a modelling framework for drones using MDE paradigm and ROS-based components.

Emmanuel Grolleau is full professor at ISAE-ENSMA, and co-head of LIAS Lab. His research focuses on performance analysis of embedded and realtime systems, real-time scheduling, and real-time systems design.