Modeling and Performance Estimation of Robotic Systems using ROS: Application to drone-based Services

Javier Merino
Dpt. TEISA
University of Cantabria
Santander, Spain
javierm@teisa.unican.es

Raul Gomez
Dpt. TEISA
University of Cantabria
Santander, Spain
raulgv@teisa.unican.es

Hector Posadas
Dpt. TEISA
University of Cantabria
Santander, Spain
posadash@teisa.unican.es

Eugenio Villar
Dpt. TEISA
University of Cantabria
Santander, Spain
villar@teisa.unican.es

***Abstract* —Smart Robots are an integral part of the 4th Industrial Revolution. Its integration as essential components in robot-based services is not straightforward. Each robot is a cyber-physical system (CPS) where a mechanical part operates under the control of a digital board(s). Modeling and simulation of such devices has specificities to be taken into account. Model-Driven Design (MDD) has proven to be a powerful System Engineering methodology able to cope with the complexity of services built as a system of CPSs (CPSoS). In this paper, a methodology is proposed to seamlessly integrate robots into a MDD framework so that the whole service can be simulated and its performance, analyzed. Although the methodology is valid for robots in general, it has been assessed on a drone-based service.**

Keywords—component, formatting, style, styling, insert (*key words****)***

I. INTRODUCTION

Autonomous, intelligent robots are an essential technology in the 4th Industrial Revolution [1]. Beyond the traditional automatized production line, the number and complexity of robot-based services inside, but also outside industrial factories is increasing dramatically. Unmanned Aerial Vehicles (UAV) or simply, drones, are an example of robots with an increasing number of applications opening the way to a large number of new business models. So, a market of USD 45.8 billion by 2025, at a CAGR of 15.5% since 2019, has been forecasted for business using drones [2].

Robot-based services are Cyber-Physical Systems (CPS) in which the functionality has to behave correctly in close interaction with the physical-world inside which it operates. In the case of robots, this interaction is two-fold. On one side, as any other CPS, the digital part has to accommodate its behavior to the timing constraints of the physical world. On the other side, the robot itself is a mechanical CPS and, at the same time, an integral part of the system. In the general case, the robot-based service will be a complex, distributed, Cyber-Physical System of Systems (CPSoS) where a large amount of SW has to be executed by a large number of interconnected computational devices of many kinds. Devices running from small embedded systems in the edge to large computing facilities in the cloud as well as other computing devices in between (the fog) [3].

There is always a high cost associated to the deployment of robot-based services. These services may involve expensive robots in close interaction among them and with a distributed computing and communication infrastructure. In many cases, the robots carry expensive payload required by its mission. The impact of any design fault detected on field may be very expensive and time-consuming. Consequently, simulation is a key technology in the verification of robot applications. The large variety of possible robots makes very difficult the availability of models. When the characteristics of the robot are relevant to its behavior and performance, a model of the robot may be required for simulation and analysis. This may imply a large effort in modeling the robot as an electro-mechanical device [4]. Most of these simulators are based on multi-agent simulation as underlying simulation technology [5]. When the electro-mechanical part is relevant, multi-physics simulation based on finite elements is required [6]. The initial modeling effort can be avoided for those robots with common characteristics that make possible to share similar simulation models, like UAVs. So, there are several commercial and open-source UAV simulators available [7][8].

Most robotic applications make use of the Robot Operating System (ROS) [9]. ROS is a widely used standard in the robotic domain. Actually, it is not an operating system but a middleware. ROS includes a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Nevertheless, as a consequence of the large variety of robots with completely different features, ROS only brings the mechanisms to create nodes and define the communication infrastructure among them. The concrete messages (orders and status to and from the robot) are different from robot to robot so that the control code developed for a robot has to be changed completely when a new robot is used, thus, strongly limiting the reusability of the code.

The most widely used communication protocol for drones is MAVLink. MAVLink is a lightweight messaging protocol for communicating with drones (and between onboard drone components) [10]. ROS includes a package providing a communication driver for autopilots with the MAVLink communication protocol called MAVROS[11].

Model-Driven Design (MDD) has proven to be a powerful SW engineering methodology for robotic systems [12][13]. MDD can provide the SW engineering methodology required by the modeling and design of the robot-based service. Its application in robotics requires the integration of the robot and its characteristics as an additional actor in the system. In many cases, the modeling and design framework is Matlab/Simulink [12][14]. Matlab/Simulink provides the advantage of combining SW modeling and simulation in a single step.

UML is by far the language most widely used to support MDD. In order to capture the required semantics in this domain a large number of Domain-Specific Languages (DSL) has been proposed [13]. A robotic DSL may facilitate the modeling of robotic-based services, but it makes difficult the interoperability among design frameworks, reduces the integration in the framework of third-party tools and limits reusability. Being focused on robot-based systems, the DSL may not support efficiently SW (system) engineering. Particularly, when the SW engineering framework uses its own profile(s) with contradictory semantics.

S3D is a general-purpose, Model-Driven, Single-Source System Modeling & Design Framework where all the relevant information about the system is captured in the same model in order to support the different design steps from the initial functional system architecture until the SW stack to be compiled to each computing resource in the decided HW implementation [15]. S3D makes use of fundamental computational paradigms allowing a domain-independent modeling. Consequently, it can be applied to services on the Internet of Things (IoT) where several domains interact each other and a holistic modeling and analysis approach is required. S3D makes use of a reduced subset of the UML/MARTE standard profile for Real-Time and Embedded systems [16]. Only when a specific concept is absolutely necessary in a specific domain, a minimal extension to the S3D MARTE subset is made. An important result from this work is that modeling, simulation and performance analysis of robot-based services do not require any domain-specific extension of the S3D/MARTE profile.

In order to facilitate the coding and improve reusability of ROS code, an S3D interface allowing the development of drone-independent control code, is proposed.

Performance analysis of the application code on each computing resource can make use of native simulation technology to estimate non-functional characteristics as execution time and energy. Using native simulation to estimate the performance of middleware is cumbersome as the technique has to be applied to the whole code in the corresponding library. In some cases, when the source-code is not available, it is just impossible. A performance estimation methodology for the ROS infrastructure is proposed. The complete modeling and performance analysis methodology is exercised on a delivery service among buildings using rovers and drones.

II. STATE OF THE ART

As commented above, Matlab/Simulink is a widely used modeling, simulation and design framework combining MDD and simulation. The Robotics System Toolbox™ from Mathworks provides tools and algorithms to design, simulate, and test manipulators, mobile robots, and humanoid robots. For humanoid robots and manipulators, this toolbox includes algorithms for collision checking, trajectory generation, forward and reverse kinematics, and dynamics using a rigid-body tree representation. In the case of mobile robots, it includes algorithms for mapping, locating, path planning, path tracking, and motion control. The toolbox provides reference examples of common industrial robotics applications [17]. The Toolbox can operate in conjunction with Matlab/Simulink and the ROS Toolbox [18], thus supporting MDD, simulation and code generation of robot-based services. Using Matlab/Simulink brings all the advantages of this SW modeling and design framework but also all its limitations. Among them, scalability, simulation speed and implementation efficiency. Moreover, performance analysis and design-space exploration are difficult. A similar approach is represented by SimCenter AmeSim [19]. It can also provide a very precise modeling and simulation of robots. In both cases, simulation accuracy comes at the cost of a smaller simulation speed.

In [13] a comprehensive overview and analysis of existing MDD approaches in robotics has been made. Almost all the proposals define their own DSL. The need for a common modeling language (ML) is highlighted by some works but, the solution proposed is a new ML [20]. In few cases, the ML is shared by some groups. Acceptance is not the only problem for a DSL. If it captures all the semantics needed by a certain domain, e.g. robotics, it will be weak in supporting SW (system) engineering at large. MDD based on UML and/or DSLs put the focus on the system architecture and the code generation, usually on a general-purpose computer. As a consequence, very few MDD frameworks support simulation and performance analysis on distributed, heterogeneous platforms. From the large list of MDD approaches for robotic systems in [13], only 10% address simulation. In most cases, the simulation is functional, and no performance analysis is made. Nevertheless, this is a serious limitation for two main reasons. On the one hand, the development of robotic systems requires simulation in order to avoid detecting design failures very late, at the prototyping stage with the corresponding high cost. On the other hand, a robot is an edge device and therefore, subject of resource-constrained design. Therefore, performance analysis and optimization are key design activities.

S3D support system simulation and performance analysis using native simulation [21]. Native simulation technology brings enough flexibility to seamlessly enable system simulation at different abstraction levels. Based on it, a multi-level simulation framework for robot-based services able to simulate the system at a very high, pure functional level at the earliest stages of the design cycle, has been proposed. Using the same infrastructure, simulation can be enriched with more accurate models for the execution times of the functional components once the code has been developed. The ROS infrastructure should be included. Finally, this code can be simulated against realistic models of the robots using simulators such as Ardupilot. Although it is possible to annotate the C++ code in order to estimate its performance in terms of execution time and energy, estimation of the impact of the underlying ROS infrastructure using native simulation is much more difficult as it would require cross-compiling and annotating the complete ROS library. Moreover, this would lead to higher simulation times with a small impact in accuracy. Being an external library, it would bring few optimization alternatives. Nevertheless, if precise

performance estimation is required (e.g., for design constraint validation or design-space exploration), it is necessary to take these execution times into account. As the percentage of ROS code in the application code is small, even a rough estimation of the ROS code performance would improve accuracy.

III. Modeling of Robot-Based Services

A. S3D Model of Robot-based Services

The goal of the modeling methodology proposed is to enable the integration of robots into the S3D system model-driven design framework. Therefore, the focus is put on the service being analyzed but taking into account the fact that part of the system functionality and performance strongly depends on the behavior of robots as mechatronic devices. In Fig. 1, a delivery service for parcels among hospitals in a city is shown:

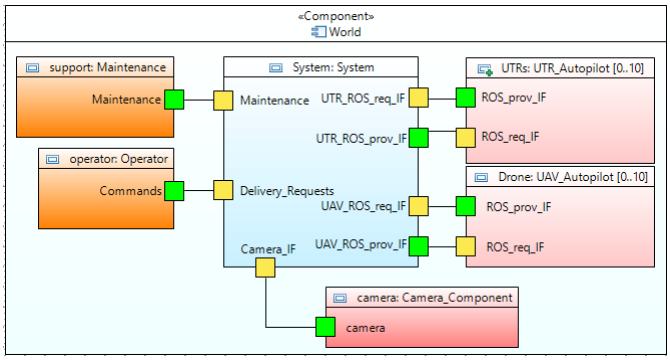

Fig. 1. The system and its environment.

The system receives orders from requesters for certain goods. Provided it is a valid request and the good is in the store, the system has to select and control the movement of Unmanned Terrestrial Rovers (UTR) and Unmanned Aerial Vehicles (UAV) in a fleet to take the parcel from the store and send it to the final destination. UTRs oversee taking the parcel to the drone port and from the destination drone port bring it to the requester. The drones are in charge of moving the parcels among drone ports. All the process is under the supervision of the provider who is informed of the service and its status at any time.

In Fig. 2, the functional architecture of the system is shown:

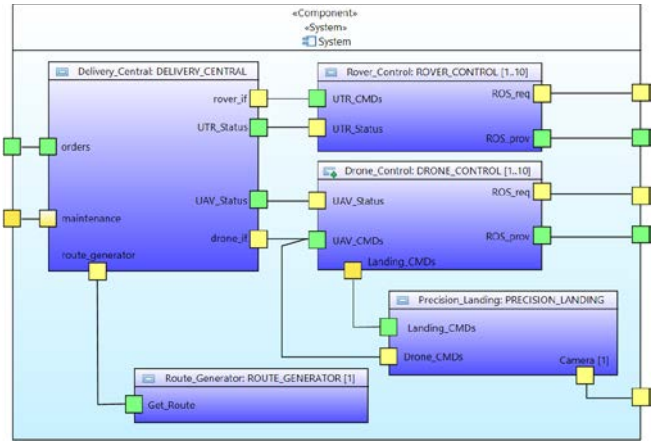

Fig. 2. The system's functional architecture.

The system is composed of five different functional components. The Delivery_Central, in charge of managing and following the requests, selecting the appropriate rover and drone, asking the Route_Generator for the best path to be followed by rovers and drones and send it to them. The Rover_Control and Drone_Control components are in charge of controlling the robots stablishing a dialog with them using ROS. The Precision_Landing component takes the control of the drone when it is close to the drone port in order to ensure that the drone takes and delivers the good in perfect shape to and from the right position.

In S3D, each component may be associated to different models in order to cover the simulation and performance analysis tasks during the different stages along the system design process. So, during system requirement analysis and partitioning, models with a minimal functionality associated to constant execution times and energy consumptions are needed. After component development, the full code is available and then, the complete behavior of the system can be simulated. Moreover, components might be in different languages. A property in the generalization of the component will provide the programming language used and the path where the corresponding file is:

$language=C++

$path=C:/projects/C4D/UC3/Demo2/Files

'Rover_Control' is not a 'pure' C++ component as it makes use of ROS functions and therefore, it needs the ROS library to be compiled. The C++ implementation of ROS is called 'roscpp'. In order to let the simulation model generator (mSSYN) to know that the ROS infrastructure has to be integrated as part of the model, it is enough to identify the programming language used as 'roscpp':

$language=roscpp

$path=C:/projects/C4D/UC3/Demo2/Files

Therefore, no modification to S3D is required due to the fact that in robot-based systems using ROS, the ROS library has to be included.

In S3D, Provided/Required interfaces are explicitly declared in the model with the list of methods they include. So, mSSYN, will connect the two component instances and linked them through the corresponding interfaces. As an example, the interface in the 'Get_Path' port will link the services required by the 'Delivery_Central' with the interface in the 'Get_Route' port of the 'Route_Generator' component providing them (Fig. 2). Based on the properties in the port and the interface, different models of computation and communication (MoCC) are supported. Contrarily, ROS follows a publish/subscribe MoCC that cannot be implemented in the same way as the rest of S3D service/client methods. The way decided to handle this difference and enable mSSYN to automatically generate the model is quite simple: just let the interfaces empty. In that case, mSSYN does nothing but launching the ROS master process because there is a ROS node in the system, and the master will be in charge of deploying the infrastructure to enable ROS communication methods.

Components in the verification environment can also be associated to different functional files. This also affects the models for the drones, i.e., the autopilot and the model of the aerodynamics. Therefore, integrating an external executable into the simulation framework may be required for some external aerodynamics simulators (e.g., Gazebo, SimCenter

AmeSim …), while others are included in the drone software (e.g., ArduCopter+SITL).

B. Common control interface for drones

ROS does not provide a unified way to use a flight controller such as Ardupilot, PX4, Paparazzi or DJI. Although it defines how to register a node, to what topic to publish and subscribe and how to publish a message in a particular topic, the content of the messages is different from one robot to another. As a consequence, the 'roscpp' code to control a robot (i.e. a Paparazzi drone) is different than the code to control another robot (i.e. a DJI) even when the orders are the same (i.e. take off, go to a GPS position, etc.). S3D targets pure platform independent modeling so that a particular code can be mapped to different execution units. In order to overcome this problem, an interface class that allows the user to perform the basic functions of a drone is proposed. This class is inherited by all the drone controllers, establishing a unique frame and unifying the control operations of the above-mentioned controllers. Each of the drone control classes should implement the interface functions according to its particular functionality. This solution could be applied to any family of robots sharing similar actions.

The C++ interface proposed has been developed for the Ardupilot, Px4, Paparazzi and DJI drones but could be extended to any other. The interface has differentiated two function groups, the GPS operation group and the non-GPS operation group as shown in Fig. 3.

GPS operations depend on aircraft positioning and require an operational positioning system, while non-GPS operations do not require operational positioning.

The developed interface is the following:

```
1  class I_RosDron {
2  public:
3      virtual ~I_RosDron() {}
4
5      //Advanced features
6      virtual void uc_drone_gps_fly(float height) = 0;
7      virtual void uc_drone_return_ground() = 0;
8      virtual void uc_drone_gps_go(double lat, double lon, double alt) = 0;
9
10     //Mode no GPS
11     virtual void uc_drone_joystick(double x, double y, double z, double r) = 0;
12     virtual void uc_drone_stabilize_fly() = 0;
13     virtual void uc_drone_alt-hold_fly() = 0;
14     virtual void uc_drone_land_ground() = 0;
15 };
```

Fig. 3. The multi-drone interface.

As with the ROS interfaces in the model, this one has not to be described in each port of the component as it does not require an implicit connection among the component instances as the usual, client-server, S3D interfaces.

For a better understanding of the interface, let us provide an example of its usage. Consider a drone initialized as a pointer to an object of one of the above-mentioned drone control modules named "drone".

To perform the first takeoff after the vehicle is connected, the "drone->uc_drone_gps_fly();" statement would be used. This function, like the rest of this interface, will remain blocked until the drone is taken off the ground and there rises to the altitude designated in each case by the controller.

Immediately afterwards the drone can be sent to a GPS position using the following statement: "drone->uc_drone_gps_go(43.4747386, -3.7986268, 50);". This instruction receives as parameters the coordinates in north latitude in degrees, east longitude in degrees, and the absolute height in meters from sea level. To use this function, it is essential to use the uc_drone_gps_fly function in advance, as it will change the flight mode of the aircraft if necessary.

At the end of the flight mission, the instruction "drone->uc_drone_return_ground()" can be used, which will cause the drone to return to the starting point and land safely.

The following functions do not require an orientation system, but a transition between modes (GPS/noGPS) can be performed during the flight.

To start a flight in fully manual mode (stabilized only) the "drone->uc_drone_stabilize_fly();" statement can be executed. Furthermore, to start a flight with height hold (stabilized and maintaining height) the instruction "drone->uc_drone_alt-hold_fly();" can be executed. This instruction is equivalent to the previous one, but if the throttle is maintained at 50% the height is automatically maintained.

These two modes above require data from a joystick controller. This requires that the controller update function is sent at least once before running the instructions above. This function is named "drone->uc_drone_joystick(0,0,500,0);", where the arguments are pitch, roll, throttle, and yaw. Throttle takes values between 0 and 1000 while the rest take values between -1000 and 1000.

To land safely without GPS the instruction "drone->uc_drone_land_ground()" can be used, which will automatically descend the aircraft until it touches ground.

This set of instructions has been chosen because it allows to perform almost any drone mission. Although not all the functions of the drone can be accessed, the generated code is independent of the flight controller so that only drone-specific commands have to be isolated from the drone-independent code.

This interface is useful when multiple drones are used. Different flight controllers have different procedures. Even, DJI uses a distinct approach in its different drones to determine the status of the aircraft, which is published in the "/dji_sdk/flight_status" topic. Next, these procedures and their equivalent in the proposed interface are described.

Paparazzi UAV integrates a ROS communication module named "pprzros". It only publishes on the topic "/pprzros/to_ros" and receives publications from the topic "/pprzros/from_ros". To use this controller, the developer has to work with a predefined data structure containing all the required drone control fields, both for sending and receiving data. In this case, the data of all drones that are flying goes through the same "pprzros" and broadcasted to all the ROS modules they listen to.

For Ardupilot and PX4 controllers, the MAVROS control module has been used. Nevertheless, even between these two controllers differences exists in the way they operate. For example, in PX4 the auto-control mode is called "offboard", while the same mode in ArduPilot is called "guided". To perform a flight in guided mode, the operation is divided into 3 fundamental parts: takeoff, mission and landing.

For taking off in PX4 it is enough to order arming engines and indicate the first reference point, while ArduPilot needs to be armed first and then sending the "takeoff" command. In

DJI and Paparazzi UAV controllers the common interface must assemble and take off the aircraft in the same way as ArduPilot and PX4, respectively. If using the proposed common interface, the developer will not have to create a complex sequence with its utility limited to a particular drone: only by executing the "uc_drone_gps_fly()" function, the necessary tasks for the drone to be in the air will be performed.

Landing in PX4 requires activating the "auto.land" mode, while in ArduPilot the serial equivalent mode is named "LAND". Paparazzi UAV and DJI have specific modes of landing that are activated by changing a particular value. Using our approach, running the "uc_drone_land_ground()" function would get the same result in each case.

When sending mission coordinates, ArduPilot and PX4 behave almost the same, since the only difference is that PX4 requires a constant refreshment of the mission, while ArduPilot only requires receiving the command once, moving to the destination until instructed otherwise or until it reaches the target. Using the proposed "uc_drone_gps_go()" function of the interface, the executed code will block until the drone has reached the destination.

For manual control, some flight controllers have several modes available. The most interesting modes are the stabilized mode, which allows a manual flight with few iterations, and the height lock, which allows to keep the drone at a fixed height while manually directing the drone. In these cases, a 4-axis flight control is used: x-axis for pitch, y-axis for roll, z-axis for throttle and r-axis for yaw. Using the "uc_drone_joystick()" function of the interface the criteria and data types are unified so it can be transparent to the developer.

This interface allows the user to only have to develop with a standardized drone and not the actual drone model to be implemented, allowing to change the model later or even use multiple different drone models. An additional advantage is a strong reduction in the lines of code to be written.

Next, real code of our approach and the original autopilot control instructions are compared. Arducopter is used as the autopilot for the following examples.

Using our proposal, the source code of a flight mission that sends the drone to three GPS coordinates and lands it at the last point, is the following:

```
1  drone->uc_drone_gps_fly(float height);
2  drone->uc_drone_gps_go(46.0828, -103.899, 940);
3  drone->uc_drone_gps_go(46.0828, -103.820, 940);
4  drone->uc_drone_gps_go(46.0820, -103.822, 920);
5  drone->uc_drone_land_ground();
```

Fig. 4. Drone-independent mission code.

This same program would require 60 lines of code for Ardupilot, increasing 12 times the original size:

```
1  mavros_msgs::CommandTOL cmdTOL{};
2  mavros_msgs::CommandBool cmdBool;
3  mavros_msgs::SetMode mode;
4  geographic_msgs::GeoPoseStamped pose;
5  double vel;
6  mode.request.custom_mode="GUIDED";
7  ros::Rate r(10);
8  do{
9    ros::spinOnce();
10   while(state.mode!=mode.request.custom_mode){
11     set_mode_client.call(mode);
12     r.sleep();
13     ros::spinOnce();
14   }
15   cmdBool.request.value=true;
16   while(state.mode==mode.request.custom_mode && !state.armed){
17     arming_client.call(cmdBool);
18     r.sleep();
19     ros::spinOnce();
20   }
21   cmdTOL.request.altitude=2;
22   while( state.mode==mode.request.custom_mode && state.armed &&
23 state.system_status!=4){
24     takeoff_client.call(cmdTOL);
25     r.sleep(); ros::spinOnce();
26   } while(std::abs(2-altRel)>0.25) ros::spinOnce();
27 } while(state.mode!=mode.request.custom_mode || !state.armed ||
28 state.system_status!=4);
29 double lat[] = {46.0828,46.0828,46.0820};
30 double lon[] = {-103.899,-103.820,-103.822};
31 double alt[] = {940,940,920};
32 for(int i=0;i<3;i++){
33   pose.pose.position.latitude = lat[i];
34   pose.pose.position.longitude = lon[i];
35   pose.pose.position.altitude = alt[i];
36   global_pos_pub.publish(pose);
37   ros::spinOnce();
38   r.sleep();
39   do{
40     ros::spinOnce();
41     r.sleep();
42     vel=(std::abs(motion.twist.linear.x)+std::abs(motion.twist.linear.y)+
43 std::abs(motion.twist.linear.z))*0.2+vel*0.8;
44   } while(vel<1.5 && state.mode!="GUIDED"); //Wait_start_motion
45   ros::spinOnce();
46 vel=(std::abs(motion.twist.linear.x)+std::abs(motion.twist.linear.y)+std::abs(
47 motion.twist.linear.z))*0.2+vel*0.8;
48   } while(vel>1.5 && state.mode!="GUIDED"); //Wait_stop_motion
49 }
50 mode.request.custom_mode="LAND";
51 do{
52   ros::spinOnce();
53   r.sleep();
54   while(state.mode!=mode.request.custom_mode){
55     set_mode_client.call(mode);
56     ros::spinOnce();
57     r.sleep();
58   }while(state.mode==mode.request.custom_mode  &&
59 state.armed)ros::spinOnce();
60 } while(state.mode!=mode.request.custom_mode || state.armed);
```

Fig. 5. Ardupilot code implementing the mission in Fig. 4.

This simplification is possible because the flight controller procedures are large and complex, meaning that many of them can be reduced into a simple function. As an example, Ardupilot requires 3 steps to initialize and takeoff the drone, while by using the proposed interface, it can be done in a single step. As an example of the code inside a method of the proposed interface, the code in Fig. 6 corresponds to the procedure for initiating the GPS-guided flight of a drone with an Ardupilot controller, equivalent to the “uc_drone_gps_fly” function:

uc_drone_gps_fly(float height)

```
1  mavros_msgs::CommandTOL cmdTOL{};
2  mavros_msgs::CommandBool cmdBool;
3  mavros_msgs::SetMode mode;
4  mode.request.custom_mode="GUIDED";
5  ros::Rate r(10);
6  do{
7    ros::spinOnce();
8    while(state.mode!=mode.request.custom_mode){
9      set_mode_client.call(mode);
10     r.sleep();
11     ros::spinOnce();
12   }
13   cmdBool.request.value=true;
14   while(state.mode==mode.request.custom_mode &&!state.armed){
15     arming_client.call(cmdBool);
16     r.sleep();
17     ros::spinOnce();
18   }
19   cmdTOL.request.altitude=height;
20   while(state.mode==mode.request.custom_mode && state.armed &&
21 state.system_status!=4){
22     takeoff_client.call(cmdTOL);
23     r.sleep();
24     ros::spinOnce();
```

```
25    }
26    while(state.system_status==4&& (relAlt<height) ros::spinOnce();
27  }while(state.mode!=mode.request.custom_mode ||!state.armed ||
28  state.system_status!=4);
```

Fig. 6. Implementation of the uc_drone_gps_fly() function.

In lines 1-3 the necessary message structures are defined and in line 4 the guided flight mode is indicated. In line 5, a frequency of 10Hz has been defined to be used when looping to send and receive messages. Then, a loop is started from line 6 to line 27, repeating as long as the drone is not flying in guided mode. Lines 7, 11, 17 and 24 correspond to the ROS statement used to update the values of subscriptions. In the loop of line 8 the mode is changed (line 9), continuing with the execution when this change is confirmed.

In lines 13-18 the drone is armed, looping if the drone status remains unarmed. On line 19 taking off is requested. The operation will be completed when the drone state changes to 4. Finally, in line 26 there is a waiting loop, which is responsible for waiting while the expected height is reached.

Eventually, if the flight mode is changed or the drone is disarmed, the loop will exit to start over.

IV. Performance Analysis of Robot-Based Services

A. C++ and 'roscpp' code

Functional code in S3D (e.g. C++) can be associated to constant execution times and energy using the 'uc_add_times" function, or be simulated using native simulation which would provide much more accurate estimation of performance figures.

When dealing with 'roscpp' code, the C++ code is managed exactly the same way but each time a ROS function is found, the methodology in the next section is applied. Recall that doing nothing would correspond to an estimated time and energy of '0' which, in fact, would imply a certain error. The goal of the methodology proposed is to decrease this error.

B. ROS methods

The proposed method consists in adding a predefined workload each time a ROS function is called. These workloads have been estimated for each function using POSIX process clocks, measuring the time elapsed by the function itself and all the involved ROS threads running in background on a certain node, in a sufficiently large sample. These times are influenced by two main factors: the processor's frequency and its architecture. Measures have been performed on a PC (Intel i5-3470 x86_64) and on an embedded platform (Raspberry Pi 4, with an ARM Cortex-A72 ARMv8). For frequency scaling, first assumption was to consider a model with an ideal, lineal relationship between core frequency and elapsed time to compute a certain load. However, error measured using this model was unacceptable. For a more precise estimation, times have been obtained at two different frequencies for each processor:

- Intel: 1600MHz and 3000MHz
- ARM: 600MHz and 1500MHz

This set of architectures and frequency combinations allows us to cover a large set of scenarios. If nodes run on a PC/Server it will likely be an Intel, and in the case of embedded platforms, ARM is the dominant processor.

Among the complete set of ROS library functions, the most frequently used, have been analyzed. The proposed technique could be applied to the rest of functions as well. Service calls have not been considered as the corresponding execution time is negligible with respect the service execution time itself. Regarding its execution time behavior, ROS functions can be divided in two main groups: functions whose execution time has a dependency with an external variable '*x*' and functions which exhibit an execution time which only depends on frequency F.

1) Dependent functions

These functions show an elapsed time which can be mathematically expressed using linear dependency with the 'x' variable:

$$\#Time\ (ns) = A \cdot x + B$$

Both factors A and B show a linear dependency with frequency F:

$$A = a \cdot f(MHz) + b$$
$$B = c \cdot f(MHz) + d$$

As a consequence, the final equation is:

$$\#Time\ (ns) = (a \cdot f(MHz) + b) \cdot x + (c \cdot f(MHz) + d) \quad (1)$$

Let us measure these parameters for the two dependent functions.

a) Publish

Sending messages from a publisher to a subscriber node is carried out calling the "ros::Publisher::publish" function. As described in [22], publishing is performed as point-to-point communication between the publisher and each subscriber. Thus, publishing elapsed time is directly proportional with the number of subscribers on a specific topic. Results are shown in Fig. 3. This lineal behavior is observed for all the scenarios.

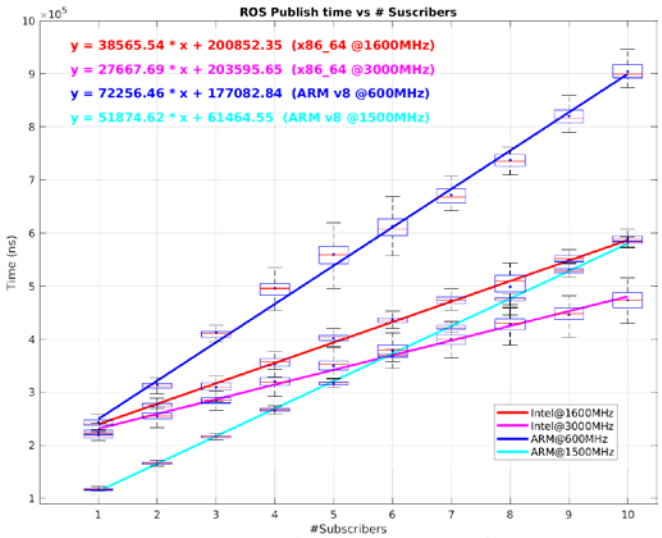

Fig. 7. Observed execution time for the 'publish' ROS function.

From the equations shown in Fig. 3, the following expressions are obtained, being *nSubs* the number of subscribers to a certain topic:

$$\#Time\_Publish(Intel) = (-7,78 \cdot f(MHz) + 51020) \cdot nSubs + (1,959 \cdot f(MHz) + 197717)\ ns$$

$$\#Time\_Publish(ARM) = (-22,656 \cdot f(MHz) + 85844) \cdot nSubs + (-128,46 \cdot f(MHz) + 254162)\ ns$$

It is worth mentioning that this results would degrade its accuracy outside the range of frequencies considered in each CPU. Additionally, several tests were made varying publish rate, message size and buffers size. However, no meaningful change was detected on the obtained results.

b) Sleep

ROS sleep functions are called from two main classes: ros::Rate and ros::Duration. The first performs a sleep of a certain period given a rate or frequency in hertzs, while the second receives a duration in seconds. These functions operate by polling, checking if the current wall-clock time has reached the desired value. Thus, there is a direct dependency between the sleeping time and the load generated on the processor. Internally, the sleep function of ros::Rate calls the sleep function of ros::Duration after converting rate to seconds, so the load has been assumed to be the same for both of them after being verified experimentally.

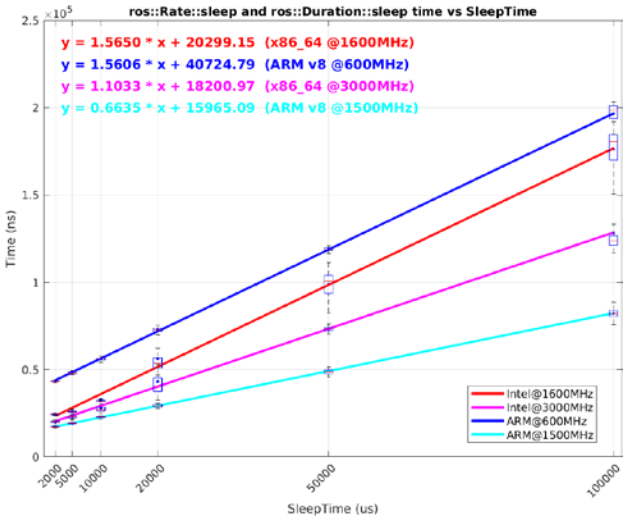

Fig. 8. Observed execution time for the 'sleep' ROS function.

From the equations shown in Fig. 4, the following expressions are obtained, being T the SleepTime time in microseconds:

#Time_Sleep (Intel)=
(-0,33·f(MHz)+2,0925) ·T +
(-1,498·f(MHz)+22697) ns

#Time_Sleep (ARM) =
(-0,996·f(MHz)+2,1587)·nSubs +
(-27,51·f(MHz)+57231) ns

2) Non-dependent functions

In these functions, equation (1) is still valid but without dependency from any 'x' factor (a=b=0):

#Time (ns) = c·f(MHz)+d (2)

The following are the functions under this behavior.

a) Spin

Messages published on a topic are received by the subscriber through a callback function, which is sent as an argument during the subscription. When a message is received, the callback function is not called directly, but rather the request it is queued until *ros::spinOnce* function is called. This allows, for instance, receiving messages at a desired rate so the node processing capacity is not exceeded.

Although this function is not dependent on external factors, it has been observed that the first time it is called by a node it takes longer. Results are shown in Table 1.

TABLE I. EXECUTION TIMES FOR THE 'SPIN' FUNCTION.

Processor	Avg. time of the first execution (ns)	Avg. time of the rest of executions (ns)
Intel @ 1600MHz	21,500	6,377
Intel @ 3000MHz	20,380	6,270
ARM @ 600MHz	54,381	21,860
ARM @ 1500MHz	24,403	9,048

b) Initialization and registration

This set is mainly formed by those functions, which are usually called one single time at node's creation. Since no dependency has been detected, best estimation consists in averaging the times obtained for a large set of executions.

TABLE II. EXECUTION TIMES FOR INITIALIZATION AND REGISTRATION FUNCTIONS.

Function	Avg. time(ns) Intel @1600MHz	Avg. time (ns) Intel @3000MHz	Avg. Time (ns) ARM @600MHz	Avg. time (ns) ARM @1500MHz
init	433,923	30,0054	18,795,260	16,195,670
NodeHandle	4,137,052	2,322,393	5,583,100	4,885,054
serviceClient	58,706	47,143	78,275	68,099
advertise	316,862	195,434	591,091	525,178
suscribe	1,078,574	731,497	1,445,835	1,360,615
advertiseServer	142,290	121,073	189,720	24,403

V. EXPERIMENTAL RESULTS

In this section, performance simulation results are presented. The application example used consists in a last mile drone-based delivery service, described above. The drone's autopilot is Arducopter and receives orders from the drone controller, which is a ROS node modelled using 'roscpp'. A MAVROS node has been instantiated to establish communication between the drone controller and the Arducopter.

In our case, we are interested in performance estimation of the drone controller ROS node. To verify the correctness of the ROS annotation, the simulated code has not been dynamically analyzed using native simulation techniques, so the ROS function calls are the only load considered. In this way we can directly handle the ROS impact without the interference from the rest of code.

The use case has first been simulated using the methodology described in Section I. These estimated execution times are now compared with the actual figures measured on the target, by dynamically measuring and accumulating the real processor time of the ROS functions with POSIX clocks. TABLE III presents the results obtained. As expected, there is a relevant error in the estimated time. This error is caused by several factors related with the global, high-level approximation to the problem. As the host where the measures are taken is an Intel CPU, the error is lower when the target is also an Intel. Since the error is lower than 100% in all scenarios the total error on a complete, annotated

example is reduced as the alternative of not considering any (i.e., ROS execution time considered negligible) would imply a 100% error.

TABLE III. ESTIMATED VS REAL EXECUTION TIMES OF THE ROS INFRASTRUCTURE.

		TIME (ns)		
	FREQUENCY	**ESTIMATED**	**MEASURED**	**DIFF (%)**
INTEL	1600	485,873,399	582,581,163	16.60
	3000	318,626,359	461,462,380	30.95
ARM	600	521,466,358	904,953,908	42.38
	1500	233,965,058	398,265,784	41.25

This impact will depend on the percentage of ROS code in a given application. If ROS code (rc) where 0% of the total application code (tac), the error in the estimation (tae) would be that of native simulation (nse, typically in the range 10-25%). If ROS code where 100% of the application code (ac) the error would be that of the proposed methodology (ree, in the range 15-45%). In general, the error would be in between both extremes following the equation:

tae = %ns.nse + %rc.ree

The following table brings these figures to the use case used in the paper:

TABLE IV. IMPACT OF THE ERROR IN THE ROS EXECUTION TIME.

	FREQUENCY	**Total Application (tac)(ms)**	**%ROS code (%rc)**	**Impact of ROS estimation error (%)**	**Impact of no estimation**	**Improvement**
INTEL	1600	22,875	2.55	0.42	2.55	83.5%
	3000	22,018	2.10	0.65	2.10	69.0%
ARM	600	213,446	0.42	0.18	0.42	57.1%
	1500	87,688	0.45	0.19	0.45	57.8%

As shown, the impact of the error in the estimation of the ROS execution time is small and, in any case, smaller than doing nothing by considering that the execution time of the ROS code is 0. The methodology achieves an improvement in the ROS execution time estimation which can be as high as 83% when the percentage of ROS code is small.

VI. CONCLUSIONS

In this paper, MDD is proposed for modeling, simulation and performance analysis of SW intensive robot-based services. A first interesting result is that if the MDD modeling methodology is general enough, no extension is needed being enough to identify which components make use of ROS. In that case, the ROS infrastructure can be automatically integrated in the system. For certain kind of robots with many similar functions, like drones, a common list of methods can be defined so that the same code is valid for all of them just by associating the functions with the appropriate code. An additional advantage of the common language is a strong reduction in the programming effort which can be reduced around 10 times.

A methodology for rough estimation of execution time of the ROS code has been proposed. Although the error can be as high as 45%, the impact is small as the ROS code will be only a fraction of the total executable. In any case, an improvement over not annotating any execution time for ROS is achieved and this improvement is higher when it is more needed, that is, when the percentage of ROS code is higher.

VII. ACKNOWLEDGMENT

This work has been partially funded by the EU and the Spanish MICINN through the ECSEL Comp4Drones project and the TEC2017-86722-C4-3-R PLATINO project respectively.

VIII. REFERENCES

[1] G. C. Fernandez, S. M. Gutierrez, E. S. Ruiz, F. M. Perez and M. C. Gil: "Robotics, the New Industrial Revolution" in *IEEE Technology and Society Magazine*, V.31, N.2, pp. 51-58, Summer 2012.

[2] Unmanned Aerial Vehicle (UAV) Market. MarketsandMarkets. 2020.

[3] C. Wöbker, A. Seitz, H. Mueller and B. Bruegge: "Fogernetes: Deployment and management of fog computing applications," proc. of *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1-7.

[4] N. R. Ramli, S. Razali and M. Osman: "An overview of simulation software for non-experts to perform multi-robot experiments", proc. of the *2015 International Symposium on Agents, Multi-Agent Systems and Robotics (ISAMSR)*, IEEE, 2015, pp. 77-82.

[5] S. Abar, G. K. Theodoropoulos, P. Lemarinier and G. M.P. O'Hare: "Agent Based Modelling and Simulation tools: A review of the state-of-art software", Computer Science Review, V.24, 2017, pp.13-33.

[6] https://www.autodesk.com/products/simulation.

[7] A. I. Hentati, L. Krichen, M. Fourati and L. C. Fourati. Simulation Tools, Environments and Frameworks for UAV Systems Performance Analysis. Proceedings of the 14th International Wireless Communications & Mobile Computing Conference (IWCMC), Limassol, 2018, pp. 1495-1500, doi: 10.1109/IWCMC.2018.8450505.

[8] E. Ebeid, M. Skriver, K. H. Terkildsen, K. Jensen, U. P. Schultz, A survey of Open-Source UAV flight controllers and flight simulators, Microprocessors and Microsystems, V.61, 2018, pp.11-20.

[9] ros.org.

[10] https://mavlink.io/en.

[11] http://wiki.ros.org/mavros.

[12] J.-A. Maxa, M. S. B. Mahmoud & N. Larrieu: "Model Driven Development for Embedded Software: Application to Communications for Drone Swarm", ISTE Press – Elsevier, March 2018.

[13] E. A. Silva, E. Valentin, J. R. H. Carvalho and R.S. Barreto: "A survey of Model Driven Engineering in robotics", Journal of Computer Languages, V.62, 2021.

[14] P. G. G. Queiroz and R. Braga; "A Critical Embedded System Product Line Model-based Approach", in proc. of the International Conference on Software Engineering & Knowledge Engineering, *SEKE*, 2014.

[15] https://es.mathworks.com/products/ros.html.

[16] https://www.mathworks.com/products/robotics.html.

[17] https://www.plm.automation.siemens.com/global/en/products/simcenter/simcenter-amesim.html.

[18] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi and M. Ziane: "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications", in: I. Noda, N. Ando, D. Brugali and J.J. Kuffner (Eds.): "Simulation, Modeling, and Programming for Autonomous Robots. SIMPAR 2012", Lecture Notes in Computer Science, V.7628, Springer, 2012.

[19] F. Herrera, J. Medina, E. Villar: "Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design approach", in Soonhoi Ha and Jürgen Teich (Eds): "Handbook of Hardware/Software Codesign", Springer. 2017.

[20] B. Selic and S. Gerard: "Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems", Elsevier, 2013.

[21] O. Bringmann, W. Ecker, A. Gerstlauer, et al., "The Next Generation of Virtual Prototyping: Ultra-fast Yet Accurate Simulation of HW/SW Systems", Proc. of DATE 2015.

[22] http://wiki.ros.org/Master.