DEVELOPING A QUAD-COPTER CONTROLLER USING DEMES

Laouen Belloli
Universidad de Buenos Aires
C.A.B.A., Buenos Aires, Argentina
Laouen.belloli@gmail.com

Cristina Ruiz
Carleton University
Ottawa, ON, Canada
cristina.ruiz.martin@cmail.carleton.ca

Gabriel Wainer
Carleton University
Ottawa, ON, Canada
gwainer@sce.carletonca

ABSTRACT
Embedded systems are every day more used to control different machines from our lives as toys, home tools, vehicles, etc. Each machine has special hardware, designed to perform their tasks optimally. There are constrains in the embedded systems that emerge from the triple relation between the hardware, the embedded software and the environment. These constrains make hard to develop and test embedded software. Each time we develop embedded systems, we need to care about the Real-Time OS that controls the hardware, and we also need to test the software many times in the hardware because the software tested in the computer differs from the embedded version and the environment changes the software behavior. DEMES (Discrete-Events Modeling of Embedded Systems) introduces a formal methodology to develop software for embedded systems using a discrete event model formalism to implement the system software as a model, abstracting the software from the hardware facilitating the testing stage and allowing a better analysis. We have used DEMES to implement a Quad-copter controller based on a PID controller using as the modeling formalism DEVS. As a case study we have implemented the controller for the Crazyflie 2.0 quad-copter.

Author Keywords
Quad-copter controller; DEMES; PID; DEVS; Crazyflie; embedded systems; embedded design methodologies.

ACM Classification Keywords
D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification---Formal methods, Validation, Model checking; D.2.10 [SOFTWARE ENGINEERING]: Design---Methodologies; D.2.13 [SOFTWARE ENGINEERING]: Reusable software --- Reuse models; I.6.8 [SIMULATION AND MODELING]: Types of Simulation---Discrete event; J.7 [COMPUTER IN OTHER SYSTEMS]: Real time; C.3 [SPECIAL-PURPOSE AND APLICATION-BASED SYSTEMS] Real-time and embedded systems;

1. INTRODUCTION

SummerSim-TMS/DEVS, 2016 July 24-27, Montreal, QC, CAN

An embedded system can be defined as “an engineering artifact involving computation that is subject to physical constraints” [5]. That is a system involving some hardware and software in relation with an environment. Nowadays, these embedded systems are everywhere: in vending machines, in cars, plane controllers, fridges, medical devices, etc. In the near future, they will probably be in almost any electrical device. Moreover, these devices demand more and more features every day, which is translated in the need of software that is more complex.

The triple relation that constrains an embedded system (hardware, software and environment) is common to most computing systems. However, there is a big difference between embedded systems and computation. In computation, there is a separation between the physical layers and the software. However, in embedded systems there are two types of constrains. First, a reaction to the physical environment is needed. Second, the software has to be executed in a physical platform. The reaction constrains may imply specific deadlines (as in Real-Time Embedded Systems (RTES) [8]), throughput and jitter. The execution constrains may involve processors speed, power, hardware failure rates, etc. [5].

All the above constrains make it hard to develop software for embedded systems in general and to RTES in particular. We need to take into account the hardware properties and the interaction between the software and the hardware.

Traditional methodologies for embedded systems divide hardware and software design. Then, both designs are conquered. Based on this division, two types of methodologies emerged from traditional design: language-based (software-centric) and synthetic-based (hardware-centric) methods. The problem with these methodologies is that they use high-level languages such as C++ that need translation to hardware level languages to have the final product. Moreover, they do not cover hardware-software co-design, and the final design is hard to verify against the initial specification. Therefore, there exist long testing phases, error-prone products, and increased time to market [10].

Most design methodologies for embedded systems are ad-hoc based. This makes it hard to scale up them for larger systems. Additionally, they require huge testing effort without a guarantee of a bug-free product. These

deficiencies come from two main weak areas: the development cycle (different artifacts and tools are used) and the system verification process (it is hard because of the discontinuities in the development cycle) [11].

Lately, formal methods show great potential in dealing with the above-mentioned issues [17]. However, they remain hard to scale up and they usually do not take into account the physical environment that the embedded system controls, resulting in expensive testing without guarantees of bug-free products. Model-based design techniques handle well heterogeneity and solve the scale up problem, but they lack of formal modeling and effective model transformation [11].

The solution to the above-mentioned problems is the use of a formal Modeling and Simulation (M&S) methodology. It combines the advantages of simulation-based approaches with the rigor of the formal methodologies [18]. DEMES (Discrete-Event Modeling of Embedded Systems) [17] is an M&S-based development methodology based on discrete-event systems specifications (DEVS). It enables the study of real-time systems and their interaction with the environment. DEVS [19], is a hierarchical and modular formalism for modeling Discrete Events Systems (DES). This formalism provides DEMES with the incrementally testing characteristic and allows testing without modifying the models (and therefore the code). Moreover, M&S techniques in general, and DEMES in particular, allow building testing scenarios easily and simulated them, without the need of the hardware and/or the need of the real environment.

Niyonkuru & Wainer developed two kernels based on DEMES: Embedded CD++ and Embedded CDBoost [10,11,13]. Both kernels differ in the communication mechanism used between model execution engines. These kernels are essential to the model-driven development concept as they are built around transforming a model of a system into the real product. These kernels are in charge of this process. Therefore, the two kernels allow the users to run their models directly on bare-metal without the need of an operating system [10].

For this work, we choose Embedded CDBoost to develop a controller for a quad-copter as it has shown better overhead performance [10].

The objective of this paper is to present how DEVS is a suitable methodology to develop a controller for a quad-copter. Our aim is also to present how Embedded CDBoost is an appropriated tool to embed the model in a real hardware without model modification. To achieve this objective, we will use as case study the Crazyflie2.0 [20,21], a small quad-copter mainly used for research, education and development.

In order to accomplish these objectives, in this work, we first introduce a DEVS model of a PID controller. We design the PID controller based on the one embedded in the Crazyflie2.0. We modeled it using DEVS and we implemented it using CDBoost. To embed it in the hardware we use Embedded CDBoost. In this process, we followed the DEMES technique as it uses a formal Modeling and Simulation (M&S) methodology.

The main advantage of our PID model is that is hardware independent. The changes needed to embed the model in different hardware are outside the model. We can adjust it to different hardware using Embedded CDBoost and just matching the inputs and outputs of the model with the outputs and inputs provided and need by the hardware. Furthermore, the embedded model using Embedded CDBoost is operating system independent.

Another advantage of our PID model is that we deal with the close loop frequency in a more accurate way regarding that in the Crazyflie2.0. The DEVS time-based characteristic allows us to easily use the elapse time to close the loop in the PID. This is more accurate that a fixed frequency because it is the exact elapsed time since the last data received until the actual data. Additionally, it also allows us to modify the frequency in the PID without a model modification, just modifying the time the data is sent.

In the development process, we have also developed a testing strategy for our model using computational simulation comparing the results against the ones from the real hardware log file in the same scenario. Moreover, we have implementing traditional testing techniques using the BOOST test module. This module allows as to automatically generating unit test to compare the implementation in our DEVS models against the functions embedded in the Crazyflie2.0.

The rest of the paper is organized as follows. In section 2, we explain the related work to methodologies used to implement and embedded controllers in real hardware. In section 3, we describe the methodologies used in this work (DEMES and DEVS) and the tools used to implement and embed the model (CDBoost and Embedded CDBoost.). In section 4, we introduce the hardware we use. In section 5, we describe the model developed and its implementation. In section 6, we present the results of our work. Finally, in section 7 we remark the conclusions of this work and state the lines for future work.

2. RELATED WORK

A quad-copter controller is an embedded system used to control, in real time, the movement of an Unmanned Aerial Vehicle (UAV). Then, the methodologies for the testing and development of quad-copter controllers are the same as the methodologies for embedded systems.

A quad-copter consists in four independent propellers, located orthogonally Figure 1. The quad-copter has movement in 3 different axes: roll, pitch and yaw.

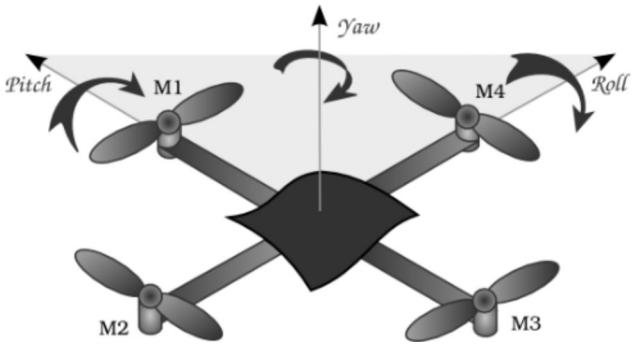

Figure 1. How the quad-copter propellers rotates.

A rotation in the roll axis is obtained by unbalancing the propellers 2 and 4, a rotation in the pitch axis is obtained by unbalancing the propellers 1 and 3 and a rotation in the yaw axis is obtained by changing simultaneously the power in propellers 1 and 3 or 2 and 4.

There are many different controller algorithms for quad-copters that are used today. Each algorithm focus in a different aspect. Which algorithm should be used to develop a quad-copter controller depends on a combination of the hardware specifications and the development purpose of the quad-copter. An analysis and comparison between different algorithms is shown in [1]. For the purpose of this work, we have used a Proportional–Integrative–Derivative controller (PID controller) [7] due to its simplicity and easy implementation. A PID controller for a quad-copter is a closed-loop controller, where the main goal is to reduce as much as possible the error between the current position of the quad-copter (yaw, pitch, roll and thrust) and the desired position. In order to achieve this, the PID algorithm uses the integrative, derivative and proportional values of the discrete function generated by calculating the mentioned error. This is done for each iteration of the closed loop to minimize the error function in a smooth way. Because the error is calculated in a discrete amount of time, and the quad-copter movement is a continuous phenomenon, the probabilities of reaching a zero error in any iteration is zero. Then, a threshold is used to determine if the error is close enough from zero to be treated as zero.

As an embedded system, the verification in quad-copter controllers is a hard task, and in many cases, when dealing with critical applications, it is crucial to have this process well done to ensure safety. As explained in [10] , traditional methodologies for embedded systems have a lack in verification processes, mainly because the verified software is commonly not exactly the same as the final embedded software in the platform. For those methodologies focused on either language-based or synthetic-based, they do not consider a hardware software co-design and thus, it is difficult to compare the obtained product against the specifications. More recent methodologies, as the model-base designs, introduce a higher abstraction level, where systems are described as abstract models and they can be implemented in several platform targets. These abstract models are iteratively translated to lower-level languages until deployable software is reached. Examples of this are the Unified Model Language (UML) and the Architecture Analysis and Design Language (AADL). For these abstract models, there exist commercial tool such as Mathlab/Simulink, Rational Rhapsody and Modelica that help in the verification stage. Some of them have automatic code generation, continuous verification test and verification of embedded systems. Even with this improvement, model-based design have a lack in precise semantic. Thus, the abstract model can be interpreted in several ways, making hard to verify the final product against the abstract model. This issue may result in fatal errors. An example of abstract modeling in Mathlab/Simulink for quad-copter controller can be found in [14].

A formal methodology is needed in order to achieve correct verification between the abstract model and the final product. For this purpose, DEMES uses DEVS as the formal modeling formalism and by implementing computable models, the abstract model can be the same as the embedded one avoiding new bugs that may appear in the model translation [17]. Moreover, DEVS is a well-defined formalism with a strong semantic allowing more rigorous analysis to achieve a verification of the system correctness and to study its properties.

DEMES allows not only to model the embedded system, but also the hardware and the environment. Then, the software can still be developed, even if the hardware is not ready, by replacing the needed hardware by models. Once the hardware is ready, we replace the models by the real hardware. In addition, it allows simulating different environmental scenarios that may not be possible to study in the real platform, either because of moral issues or because of physical/monetary impossibilities. Then, DEMES not only allows modeling the hardware and the environment, but also do this without any modifications in the software that is being tested in a computer when we embedded it in the platform, no matter in which platform the software will be embedded.

As an embedded system, a quad-copter controller developed with traditional methodologies deals with the above-mentioned drawbacks. Example of these issues are shown in [1,4,7,14], where mathematical models of quad-copters are done and validation is achieved by running the controller algorithms in the simulated quad-copter models. Whenever those controllers must be embedded in real quad-copter platforms, they must be translated to lower-level software that runs in the specific embedded system and the already achieved validation is no longer valid.

When using DEMES for quad-copter controllers, the sensors can be modeled and simulated in order to allow a computational study of how the controller works and to be abstracted from specific sensors specifications. This allows implementing a flexible controller that can easily be embedded in different target platforms without any

modification in the model. In addition, mathematical models can be used in order to simulate the effect of the controller behavior in the quad-copter and thus, be able to test the correctness of the model. Because of the well-defined hierarchical structure of DEVS models these new models are completely separated from the controller and no modifications are needed. Furthermore, model reusability is also achieved in DEMES because of the modularity of DEVS models.

In order to implement a quad-copter controller using the DEMES methodology we have used a kernel (Embedded CDBoost) that allows the users to run their models directly on bare-metal without the need of an operating system [10].

3. METHODOLOGY

In this section, we explain the methodologies and tools used in this work. We present DEVS (Discrete EVent System Specification) as the formalism to model and simulate dynamic systems. We also introduce DEMES (Discrete-Event Modeling of Embedded Systems) as a formal M&S method to analyze real-time systems and their interaction with the physical environment. It also allows us to use the original models as part of the final product. Finally, we describe the main features to the tools we use to implement the model: CDBoost simulator and Embedded CDBoost.

3.1. DEVS formalism

Discrete Events system Specification (DEVS) [19] is a hierarchical and modular formalism for modeling Discrete Events Systems (DES). The hierarchical and modular structure of DEVS allows defining multiple models that are coupled to work together in a single and model by connecting their input and output through messages. In the same way, the resulting model can also be coupled with others models defining multiple layers in the hierarchical structure.

In DEVS, there are two kinds of models: atomic, which define the behavior of the system, and coupled, which describes de structure of the system.

Atomic models define the behavior of the system. The transition functions implement the behavior and change the state of the system. In addition, when the system state changes, the atomic model decide how long the system remains in the new state until an internal transition is triggered. Before every internal transition occurs, the output function is called in order to obtain the output from the current state.

An atomic model is defined as the next 8-tuple:

Where:

Is the set of input events.

Is the set of output events.

Is the set of sequential states.

is the time advance function that determines the time until the next internal transition.

is the internal transition function that determines the next state when external events arrive. Is a set of bas over elements in X.

is the internal transition function that determines the internal state transition of the model.

is the confluent transition function that determines the next state when external events arrive at the same time that the internal function has to be trigger.

is the output function that determines the output of the model based on its current state.

Coupled models are defined connecting multiple DEVS models throw messages. These models can be either, coupled or atomic models. The hierarchical structure of a DEVS model is defined in the coupled models. Each time a coupled model is generated from others models, a new level appear in the hierarchy. Then, the hierarchical structure is constructed by multiple couplings.

A coupled model is defined as the next 8-tuple:

Where:

Is the set of input events.

Is the set of output events.

Is the set of the names of the sub-components.

Is the set of sub-components where .

is the set of external input coupling (EIC). It determines which sub-components handle the external input of the coupled model.

Is the set in internal coupling (IC). It defines the relation between output/input of the sub-component.

Is the set of external output coupling (EOC). It determines which sub-components will send message as the output of the coupled model.

One advantage of the DEVS hierarchical structure is that it allows us to test our models at different levels. We can do unit test of the atomic models, and gradually do integration test connecting the atomic model until we get at the top model level. If an atomic model changes, we only need to repeat the unit test of this particular model and those integration ones that include the modified model.

Another advantage of DEVS to model the controller of a quad-copter is that DEVS is a time-based formalism. This time-based characteristic allows us to easily deal with the close loop frequency of the PID using the elapsed time of the model.

3.2. DEMES (Discrete-Event Modeling of Embedded Systems)

Every DEMES [17] is an M&S-based development methodology based on discrete-event systems specifications. It enables the study of real-time systems and their interaction with the environment. It also allows us to use the developed model as part of the final product. To

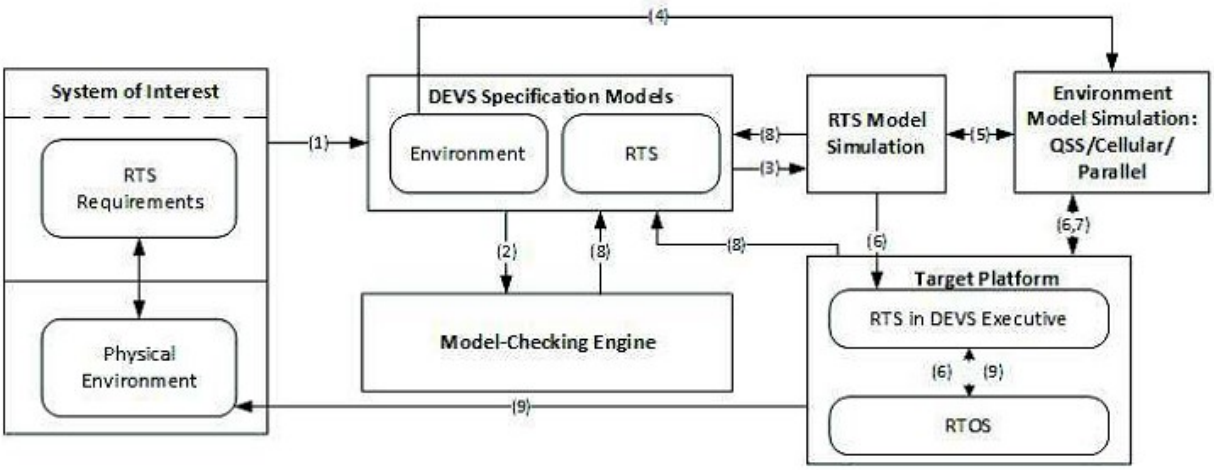

Figure 2. Demes developement cicle (taken from [13])

achieve this, a formal M&S methodology (DEVS) is used in the first stages.

DEMES development cycle is depicted in Figure 2.

The process starts with the definition and modeling of the System of Interest, both the real time system (RTS) and its environment. These models can be done directly on DEVS or using another technique and then translated to DEVS (1). Once the DEVS models are developed, model checking is used for verification and validation of the model properties (2). At the same time, the RTS DEVS models are used to run the simulations in the simulated DEVS environment (3). We can also simulate the physical environment together with the DEVS RTS models (4). Sometimes is exponential do model checking. We can simulate the individual behavior of DEVS sub models using different scenarios of the environment (5). The tested sub models can be incrementally deployed in the target platform (6). A real-time executive executes the models on the hardware (9). If the hardware is not readily available, the software components can still be developed incrementally and tested against a model of the hardware. As the design process evolves, both software and hardware models can be refined, progressively setting checkpoints in real prototypes. At this point, those parts that are still unverified in the formal and simulated environments are tested. Most of the testing phase (7) can be done using simulation, under a risk-free testing environment. DEMES allows doing design changes incrementally in a spiral cycle (8). The development cycle finishes with the RTS fully tested and every model deployed on the target platform.

DEMES combines the advantages of M&S with the methodological rigor provided by DEVS formalism [19]

DEMES [17] enables the incremental construction of such embedded applications using a discrete-event architecture for both simulation and the target product architecture. The use of DEVS for DEMES offers the following advantages:

- **Reliability:** logical and timing correctness rely on DEVS system theoretical roots and sound mathematical theory.
- **Model reuse:** DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition.
- **Hybrid modeling and knowledge reuse:** DEVS is the most general discrete event formalism (i.e., every other method can be expressed as DEVS), and many techniques used for embedded systems have been mapped into DEVS. Hence, we can use different the most appropriated method for modeling and then translate it into DEVS, while keeping independence at the level of the executive.
- **Process flexibility:** hybrid modeling capabilities are transparent for the executive, which is defined by an abstract mechanism that is independent from the model itself.
- **Testing:** defining experimental frames can be automated.

3.3. CDBoost and Embedded CDBoost

In this project we used a new extended version of the CD++ toolkit [16] called CDBoost [15][12], a DEVS simulator implemented in C++11. This simulator is cross platform and it can also run in embedded platforms. The simulator is implemented only as a header library that can be included in any C++11 project without any previous compilation. It only uses the C++ Boost library [6]. CDBoost simulator is easy to install and fast to include in any project.

As described in [12], the CDBoost architecture defines the PDEVSAtomic and PDEVSCoupled classes that extend the Model class. PDEVSAtomic has the virtual methods internal(), advance(), out(), external() and confluence(). We can then, implement the behavior of an atomic model in CDBoost extending PDEVSAtomic and implementing these virtual methods. To implement a coupled model we need to create a new object of PDEVSCoupled using

vectors of models to define the EIC, IC and EOC. At the same time, PDEVSSimulator is in charge of running simulations.

The CDBoost simulator runs in any OS that has a C++11 compiler as gcc and clang. Instead, the Embedded CDBoost simulator (E-CDBoost simulator) is prepared with a compiler that compiles the model and the simulator directly into a single piece of software that can be embedded in platforms without the need of any OS.

One of the differences between the E-CDBoost simulator and the CDBoost simulator is the capability of handling Real-Time (RT) simulation. CDBoost simulator runs the simulation using the model current time only as a way to determine which models from the list of models are the next ones to execute a transition. In the other hand, the E-CDBoost simulator provides a wall-clock time that communicates with an on board clock to check execution deadlines and thus, run the simulations in a RT-mode. Then, the model transitions will occur in coordination with the time of the on board clock. Because of this, the models advance time must be well defined not to generate scheduling problem by setting a transition time that cannot be reached on time according with the on board clock.

Another difference between the E-CDBoost and the CDBoost simulator is the model input/output system. The CDBoost simulator is only used for simulation purposes and then, the model input/output does not need to interact with any external device. Because of this, the CDBoost simulator has no external input. All the inputs are generated by generator models, which are scheduled to send messages to the top model. These generators are actually part of the model even if they are not considered as part of it. Also, the output can be shown, saved or processed as desired but it is only semantic information. Instead, the E-CDBoost simulator can interact with different devices as motors, sensor and radio transmitters and it needs a way to do so. Because of this, the Simulator implements external ports and drivers that communicate the input/output of the top model with the devices.

The E-CDBoost simulator uses the external ports to communicate with the external devices. It will use these ports each time an output message is generated by the model. These ports are implemented extending the new port class and specifying a pDriver (port driver) that translate the message value into specific hardware component commands. The same system is used for the model input. In this case, the pDriver gets the components signal and translate it into correct port values that will be read by the simulator.

4. CASE STUDY – CRAZYFLIE 2.0

The Crazyflie 2.0 [20,21] is a 27g quad-copter easy to assemble. Its advanced functionality and its easiness to fly in doors makes the Crazyflie 2.0 specially designed for research purposes. The Crazyflie 2.0 board is equipped with an EEPROM memory to store the configuration parameters.

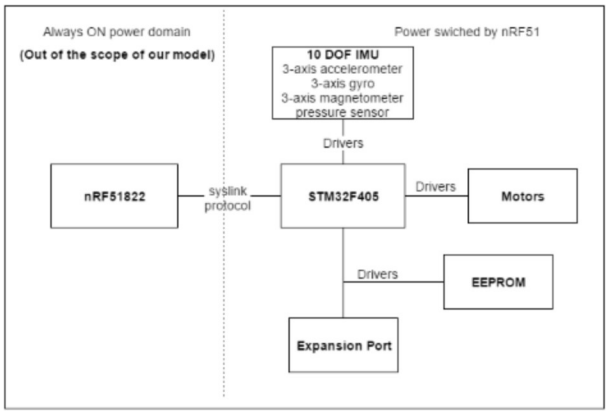

Figure 3. Crazyflie 2.0 architecture simplified (based on [22])

It is also equipped with a 10-DOF IMU with accelerometer, gyro, magnetometer and a high precision pressure sensor. The hardware comes with a development environment already installed in a virtual machine. That virtual machine also works just for flying the quad-copter.

One of the main advantages of the Crazyflie 2.0 is that it is open source under the GNU General Public License. This is especially important for our project. It allows us to study the software implemented inside, reutilize the parts we need, make changes and run away from the parts we do not need, such as the FreeRTOS OS. Remember that Embedded CDBoost do not need and OS to run the models in the platform. It also allows us to embed in the hardware our DEVS model using Embedded CDBoost.

The Crazyflie 2.0 has two microcontrollers, a NRF51822 Cortex-M0 and a STM32F405 Cortex-M4@160MHz.

The NRF51 is in charge of the radio communication and the power management. The STM32F405 runs the main firmware. It handles the sensor reading, the flight control, the motor control and other features such as the expansion ports. The RAM memory of the STM32F405 is 196kB.

The communications between the two microcontrollers are handled by the syslink protocol [23]. In this work, we are not going to work with the NRF51822 microcontroller. That is why how this protocol works is out of our scope.

A schema of this architecture is shown in Figure 3.

Our project focus on developing a controller for a quad-copter using DEVS, it is not about power management and radio communications. For this reason, in this work we are not going to modify the NRF51 microcontroller. We will work with the STM32F405 as it in charge of the sensor reading, the flight control and the motor control. However, in future steps we might have to focus on this microcontroller too if we want, for example, flight the quad-copter with a joystick connected to a computer and send the desired values via radio communications.

The Crazyflie 2.0 sotfware has 3 main components: the real-time operation systems (FreeRTOS) [24], the drivers and the modules. The FreeRTOS runs the different tasks of the software, such as radio communication, stabilization,

power management, etc. The drivers are the functions designed to get and send data to the hardware. Finally, in the modules, the behavior of the controller is implemented: the treatment of the data from the sensor, the PID calculations for the quad-copter's axis, the calculation of the power to be sent to the motors, the behavior of the led, etc. The source code of the original project is available in the Bitcraze Virtual Machine [25].

It is worth to mention that as any OS, the FreeRTOS functions are called from the modules' functions in order to provide a task scheduler that the controller uses to execute its tasks. The drivers' functions are also called inside the modules functions to interact with the hardware.

The model we have developed for the Crazyflie 2.0 is a simplification of the original software. We have only included the basic features to get the quad-copter fly. Our controller takes the data from the sensors (actual values) and the commander (desired values) and calculates the power to be send to the motors based on this data. We threat the data from the sensors in the same way as in the original project. We have also implemented the same PID calculations, without the hold-attitude mode feature (this feature keeps the quad-copter at a certain altitude fixed by the commander).

The controller that is implemented inside the Craziflie 2.0 works, basically, by initializing a group of tasks that are triggered by the FreeRTOS OS. The Main modules are the stabilizer, the controller, the PID, the command and the imu modules. There is also some debugging modules as the console and the log modules. The rest of the modules are not part of the quad-copter controller itself, but they are needed to communicate with the OS and to control different hardware as the led. We will explain the responsibilities of these main modules in order to understand the parts of the controller needed in our model to get the Craziflie 2.0 fly.

The PID module: This module is in charge of handling PidObject variables. The PidObject data type stores the information that is obtained from the error between the desired value and the actual value and the derivative, proportional and integrative values of this error. Then, the PID module takes a PidObject variable, a desired value and an actual value and calculates the new PID value. It also updates the PidObject with the new error obtained from the actual and the desired values.

The Controller module: This module is in charge of the rate and attitude PID calculations (both calculations are explained later on). For this purpose, it uses the PID module and handles the PidObject variables for each axis with their semantic, this mean, which kind of error are each PidObject storing.

The command and imu modules: These modules are in charge of the communication of the controller with the commander and the sensor device. The data used from the sensor are the accelerometer, the gyro and the magnetometer.

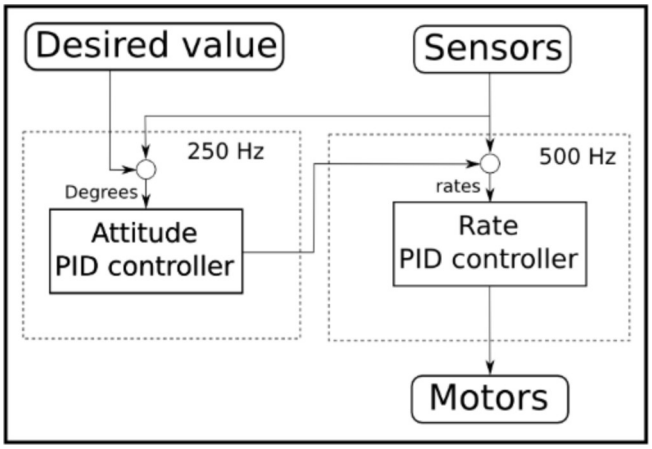

Figure 4. Original Crazyflie 2.0 PID controller (based on [26])

The stabilizer module: This is the main module of the controller. Once its task is triggered by the FreeRTOS, it stays running until the quad-copter is turned off. The stabilizer task is a loop that executes its iterations at a predefined frequency (currently 500Hz). In each iteration, it read the data from the commander and from the sensor to get the desired and the actual values, then it uses the controller module to calculate the new PID values and once it get this values it calculate the correct amount of power to send to the motors.

The sensors are used for two main purposes: The first one is to obtain the desired rate and the second one is to obtain the final correct rate to send to the motors. Now we will explain these two purposes better:

Using the sensors to obtain the desired rate: The desired values are updated using the data from the commander. This data is Euler orientation values. Euler orientations describe the orientation of a rigid body. For this purpose, it uses a 3 dimensional coordinate system and gives for each axis the angle that describes the rigid body rotation in that axis [2]. The Euler orientation values must be transformed into rate values in order to be used by the PID controller which calculates motor rates in degree/s. For this purpose, it uses the gyroscope and accelerometer data and calculates the actual Euler orientation to compare against the desired Euler orientation (this is done at a frequency of 250 Hz). The comparison between the de desired Euler and the actual Euler orientations is made using PID calculations and obtaining as result the desired rates.

Calculating the correct rate to send to the motors: The gyroscope data is used along with the desired rate in order to calculate the correct PID rate to send to the motors (this is done at a frequency of 500 Hz).

A diagram of the relation between the sensors, the desired values and the controller is shown in Figure 4.

Most of the modules of the original Crazyflie 2.0 controller are used to communicate with the hardware through the FreeRTOS. We use an embedded kernel for DEVS that does not require OS, and thus, we do not need any OS

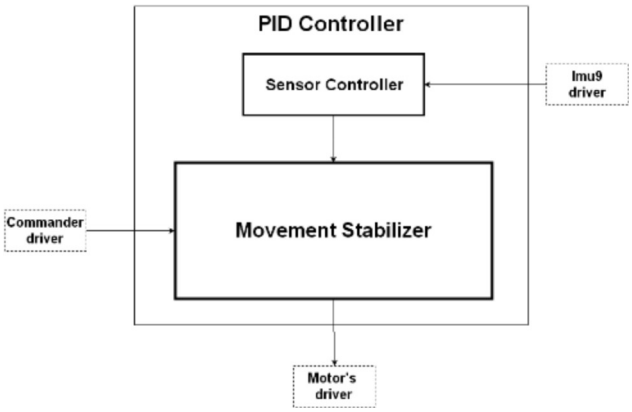

Figure 5. Pid controller top model architecture

related module. Our model only need to handle the main modules mentioned above where the motors' power is calculated. For this purpose, our model receives as input, the data from the sensors and the commander. It uses this data to start the PID calculation and the power to be sent to the motor based on the difference between the actual values and the desired ones. Once the calculations are over, the model sends the new motor power values as output. It is worth to notice that in contrast with the original Crizyflie 2.0 controller, our model does not implement a loop, instead it wait for the sensor to send the data.

In order to achieve a close-loop at a predefined frequency, the driver in charge to communicate the sensor with the model, implement a polling mechanism with the correct frequency to read sensor data and translate the data into correct port value and thus, the model receives input from the sensor periodically. The model output is send to the motors also through a driver and the motors update their power. The generated cycle achieved by polling is what replaces the loop of the original Craziflie 2.0 controller.

The Crazyflie 2.0 comes with a set of led that can be programmed to be turned on and off as desired. This is very useful to allow a visual communication between the user and the controller, but is not essential to fly the quad-copter and we are not implementing any controller for them.

5. MODEL DESIGN AND IMPLEMENTATION

In this section, we explain the architecture of our PID controller for the Crazyflie 2.0, and how it was implemented.

5.1. System Architecture: DEVS Top Model

Our PID controller top model is composed by two coupled models: a Sensor Controller and a Movement Stabilizer Figure 5.

The PID Controller receives data from two different sources: the sensor and the commander. The sensors input data is represented in the figure as Imu9 driver, as it is the name of the driver that sends the data from the sensors to our model. The commander is represented in the figure as

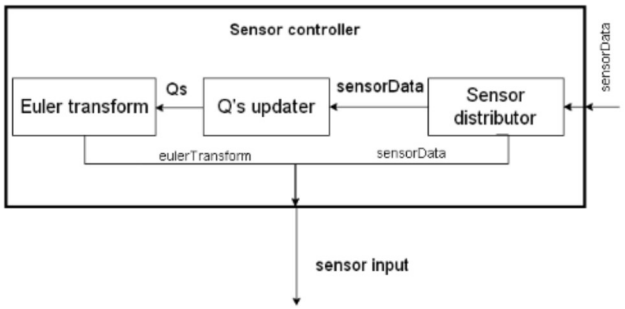

Figure 6. Sensor controller coupled model.

Commander driver, as it is the name of the driver that connects the hardware to the model input.

The Imu9 driver sends to the model the actual data that the sensors from the quad-copter are sensing. That is the positions x, y, z from both the gyro and the accelerometer. The Commander driver sends to the model the desired roll, pitch and yaw from the quad-copter and the desired thrust. It is worth to mention that the desired roll, pitch and yaw can be and angle or an angle rate.

Based on the above mention data, our model calculates the power to be sent to the motors to reach the desire values set by the Commander. The output of the PID Controller is send to the Motor's drives, which connects the model to the motors of the quad-copter. The output of our model is the power to each of the 4 motors of the quad-copter.

As we have mentioned above, the desired values from the commander are Euler orientation values that must be transformed into rate values in order to be used in the PID controller. For this purpose, we use the sensor controller to transform the sensor data into Euler orientation values.

The Sensor controller coupled model is depicted in Figure 6. It is composed by 3 atomic models: Sensor distributor, Q's updater and Euler transform (These models are explain below). The Sensor controller model input is the data from the sensor (positions x, y, z from both the gyro and the accelerometer). The sensor controller uses this data to calculate the Euler roll, pitch and yaw from the quad-copter. The output of the model is alternant: once the raw data from the sensor, once the Euler roll, pitch and yaw.

The movement stabilizer model is depicted in Figure 7. The movement controller model is the one in charge of coordinate the 6 PID models and processes the results to send to the power calculator model (each sub-component will be explained later on).

Each time a new command is received, the input will arrives from the commander input and the information carried by the message is used to set new desired values to be used in the PID calculations. In addition, each time there is an incoming message in the sensor input, a new iteration of the closed loop start and the sensor data is used as the actual values to be combined with the desired values and thus start the PID calculations. Once the PID calculations are done, the results are sent to the power calculator model

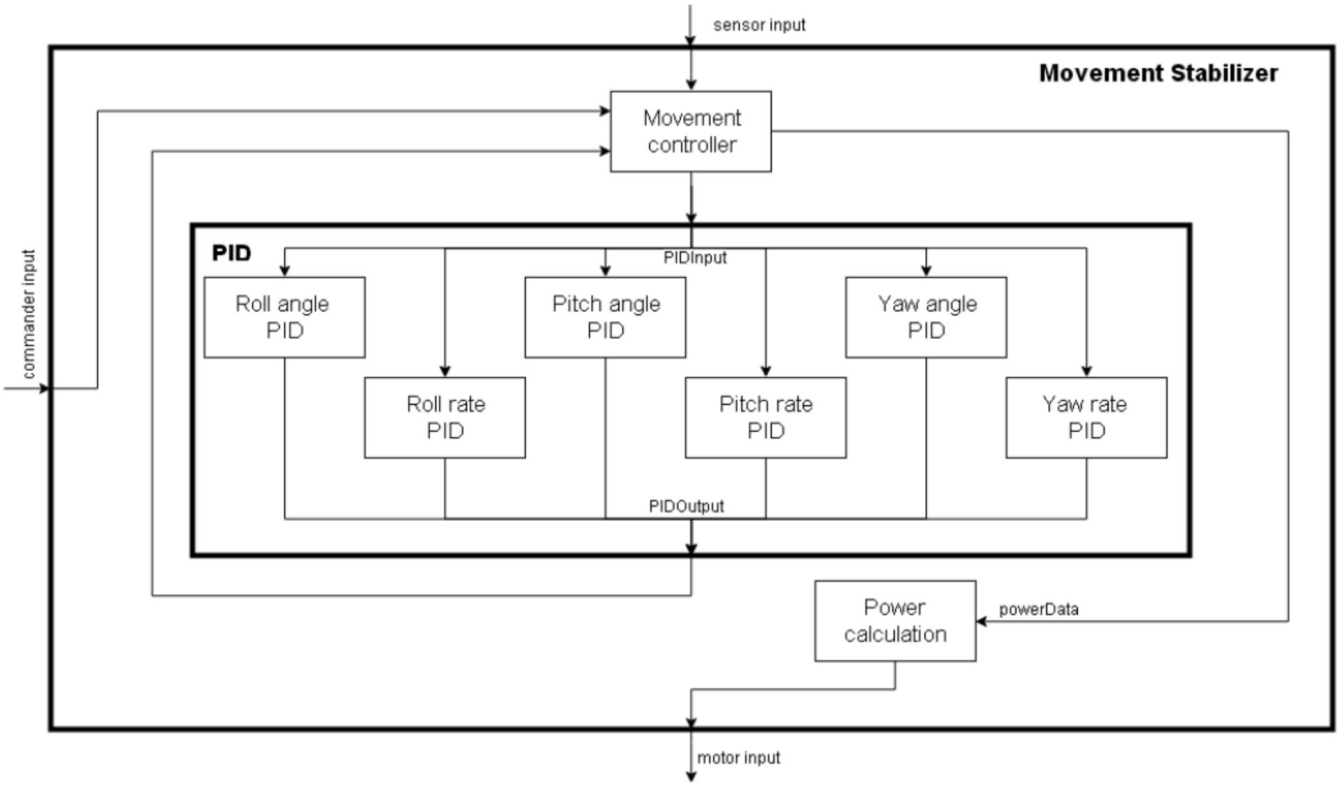

Figure 7. Movement stabilizer coupled model

where the amount of power to send to each motor is calculated using the PID results.

DEVS is a formalism used to model phenomena and each DEVS model (atomic and coupled) represent in some abstraction level the behavior of a phenomenon. Since in this work we are modeling a controller, our phenomenon is a set of algorithms and the models represent the behavior of those algorithms. The algorithms that we are modeling in this work are written using Object Oriented Programming (OOP) and then, there are function and method calls, assignations and control flow sentences (as conditionals and loops).

Some of our models such as the PIDs, the Power Calculator, the QS updater and the Euler Transform models represent functions or methods. For them, external transitions represent the function or method call and it must then be executed. For this purpose, each time an external input arrives, the model execute the function or method algorithm in the external transition and once the result is ready, the output is returned through a message using the output function and after the result is sent, the internal

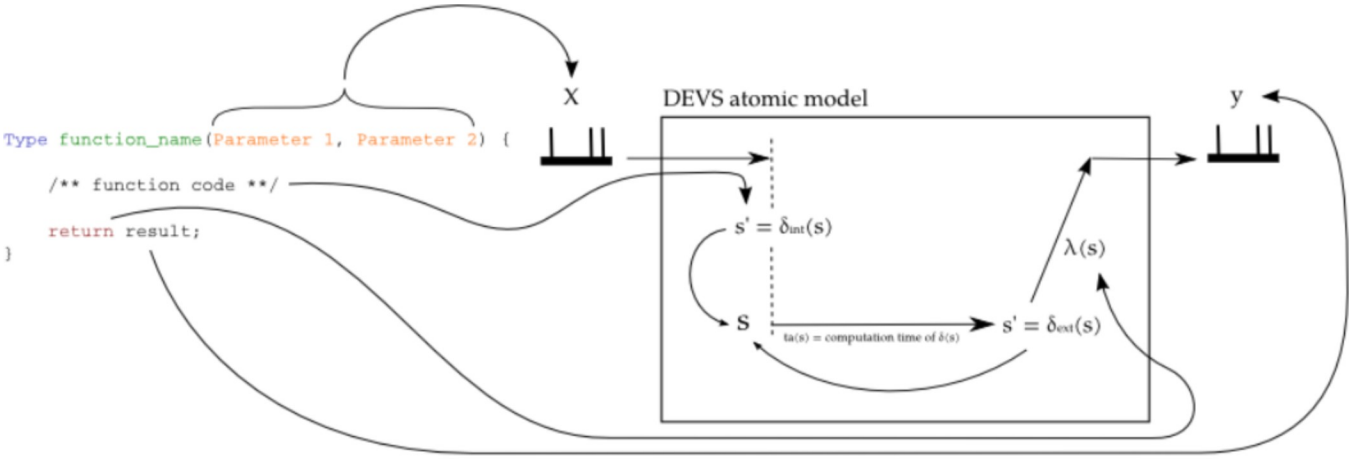

Figure 8. Mapping between a C++ function and a DEVS atomic model.

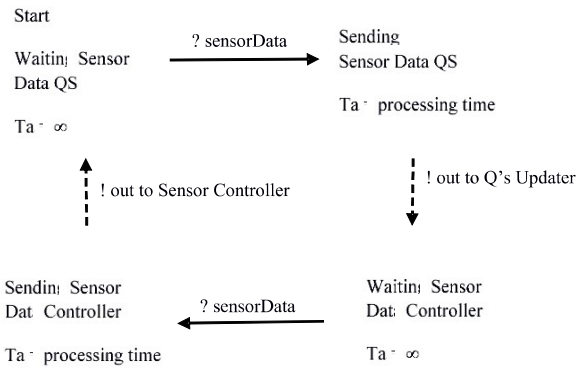

Figure 9. Sensor distributor DEVS graph.

transition passivates the model to represent the end of the function or method execution. Figure 8 Shows how is the mapping between a function wrote in C++ and a DEVS model that is representing the function.

The Sensor Distributor and the Movement Controller are modeling the algorithm control flow. The external transition represents either the evaluation of conditions with the subsequent decision or the assignments of a code block. In the case of conditionals, the output function serves to communicate the decided action to the responsible of execute it. In order to evaluate the condition, the incoming message values are combined with the model state. For the code block, the model internal state stores the partial assignments while executing, and the output function is used to trigger function calls. Each time an external event arrives, the incoming messages carry the value of the previously called functions, the external transition updates the internal state, and depending on the ending of the code block, it can execute more function calls or send the result to the next stage of the algorithm.

5.2. DEVS model specifications

In this section, we explain the behavior of each of the atomic models that compose our PID controller.

Sensor Distributor Atomic Model

The sensor distributor is in charge of managing the data from the sensor. Its DEVS graph is depicted in Figure 9.

The model is initially passivate in the "Sensor Data QS" state. When an input is received (Table 1), the Sensor Distributor remains in the same state, but the time advance of the state is actualize to a processing time value and the information received is stored. After this time, the information stored is send to the QS Updater and the model passivate in a new state "Sensor Controller Data". Again, when an input is received, the Sensor Distributor remains in the same state, but the time advance of the state is actualize to a processing time value and the information received is stored. After this time, the information stored is send to the Sensor Controller output and the model passivates in the initial state "Sensor Data QS". This process continues every time an external event arrives.

Input	Output	
gyro_xgyro_ygyro_zacc_xacc_yacc_z	To Sensor Controller output portgyro_xgyro_ygyro_zacc_xacc_yacc_z	To Qs Updater output portgyro_xgyro_ygyro_zacc_xacc_yacc_z

Table 1. Sensor distributor model input and output messages.

We use the Sensor Distributor in our PID Controller because the PID controller implemented in the Crazyflie 2.0 uses alternatively the raw data from the sensor and the

Euler roll, pitch and yaw to calculate the error function. Thus, in our model, we need to distribute this data: once it is sent to the Sensor controller output directly and in the next step, it is sent internally to the atomic model the distributor is connect to.

Qs Updater Atomic Model

The Qs Updater atomic model implements the orientation algorithm we are using to estimate the orientation of the quad-copter in a computable efficient way. It uses the quaternion representation. Our Qs Updater model can be initialized with the two algorithms implemented in the Crazyflie 2.0 the Mahony algorithm [3] and the Madgwick algorithm [9]

The difference between the Mahony and Madgwick algorithms implement in the Qs Updater is the algorithm it uses to calculate the values from the quaternion based on the sensor data. The only difference in the behaviour of the Qs updater model is when the model receives an input and the external function is trigger. If the model is instantiated with the Mahony algorithm, the external function uses this algorithm to process the data. When it is instantiated with the Madgwick algorithm, it uses this other algorithm.

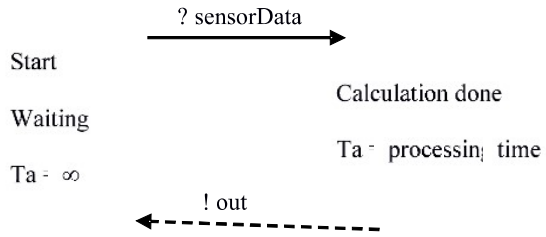

Figure 10. Qs Updater DEVS graph.

Input	Output
gyro_xgyro_ygyro_z acc_xacc_yacc_z	gyro_xgyro_ygyro_z q0q1q2q3

Table 2. Qs Updater model input and output messages.

So, the Qs Updater is in charge of calculating the values of the quaternion to later on (in the Euler Transform atomic model) estimate the position of the quad-copter. It also resends the data from the gyro. Its DEVS graph is depicted in Figure 10.

The atomic model is initially passivate. When an input is received (Table 2), the external function is trigger, the quaternion values are calculated (either with the Mahony or Madgwick algorithm) and stored. The values from the gyro are also store. The model state changes to active with a processing time advance. After this time advance the values stored (quaternion and gyro) are sent (Table 2) and the model passivates again.

Euler Transform Atomic Model

The Euler Transform atomic model calculates the Euler roll, pitch and yaw based on the input data. Its DEVS graph is depicted in Figure 11.

The atomic model is initially passivate. When and input is received (Table 3), the external function is trigger and the Euler yaw, pitch and roll are calculated and stored using the values of the quaternion and the gyro. The values from the gyro are also store. The model state changes to active with a processing time advance. After this time advance, the values stored (Euler angles and gyro) are sent (Table 3) and the model passivates again

Movement Controller Atomic Model

The movement controller model is the model in charge of managing the interaction between the 6 PIDs models of the movement stabilizer and the input of the movement stabilizer. In one hand, the commander input is the desired

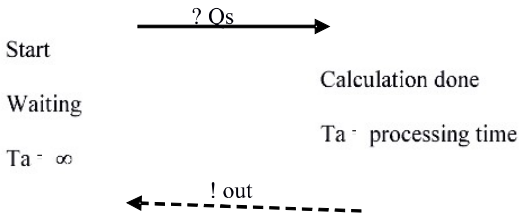

Figure 11. Euler Transform DEVS graph.

Input	Output
gyro_xgyro_ygyro_zq0 q1q2q3	gyro_xgyro_ygyro_zeul er_rolleuler_pitheuler_ yaw

Table 3. Euler Transform model input and output messages.

Euler orientation of the quad-copter and it must be transformed into desired rate values for the motors. For this purpose, the movement controller uses the processed sensor data from the sensor controller as the actual Euler orientation of the quad-copter and using the 3 Angle PID models it calculates the desired rate that represent the desired Euler orientation from the commander. If the desired Euler orientation from the commander is the same

that the actual Euler orientation from the sensors, the desired rate is zero. In other hand, the the movement controller uses the obtained desired rate and the gyroscope sensor data to calculate the axis actuator to send to the powerCalculator model explained later on.

Each axis (Pitch, Yaw and Roll) is treated independently from the others and their final calculated PID values are joined only in the final step handled by the powerCalculator model and send to the motors. The movementController model internal state store the state and the current values of each axis and their PID calculations are made asynchronously. i.e. there is any coordination between the axes while calculating their PID values. Each time the state of one axis is updated in the movementController model, the movementController model checks if all the axes have their state in ready and when that happens, the movementController model send the axes values to the powerCalculator model.

Figure 12 shows the behavior of the movementController atomic model. When the model start, the model waits the commander to get the desired values, onces, the commander send the desired values (Table 4), the model wait for the sensor data to arrive (Table 4). Once the sensor data arrives, the model start processing the axis data sending the desired and the actual values to the PID models and receiving the PID model result until all the axes data is ready and then, the axes power data is sent to the powerCalculation model. Depending if the sensor controller sends Euler orientation values or just the sensors data, the movementController will use the PID angles or not. If Euler values are received from the sensor controller, then the desired rates can be calculated and before start calculating the next rate values of each axis to send to the motors (Table 4), the desired rate is updated using the PID angle models. If the sensor controller does not send Euler orientation values and just

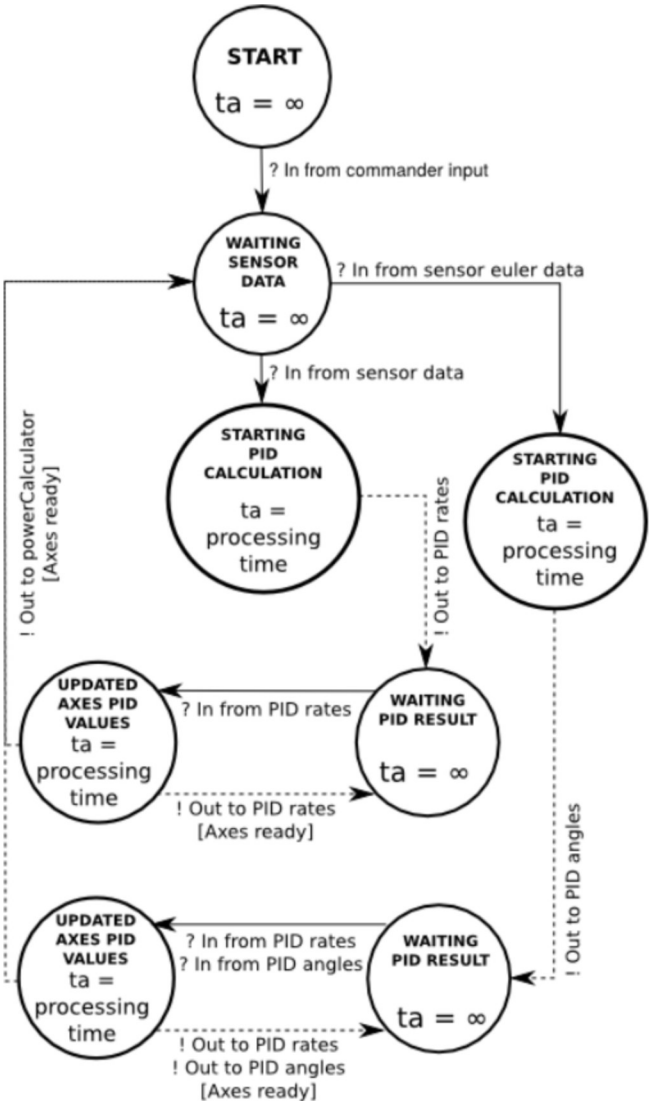

Figure 12. Movement Controller DEVS graph.

send the sensor data, the desired rates can be updated and the movementController model uses the same desired rate values than the last time. The sensor controller normally sends Euler orientation values at a frequency of 250 Hz while it send the sensor data at a frequency of 500 Hz. In Figure 12 we have only include the first commander input which is important to be able to work, but the commander input can arrive at any moment and when that happens, its value in the movementController internal state will be updated to be ready to use the next time the sensor send Euler orientation data. Figure 13 shows the representation of the state of each axis in the movementControler model internal state and the possible values.

PID Atomic Model

The PID models are in charge of doing the PID calculations of the closed loop controller. Since the PID calculations are made using the derivative and integrative parts of the error function between the actual values and the desired values, the PID models keep in their internal state the needed

Figure 13. Axes (Pitch, Roll and Yaw) representation in the movement controller internal state.

Input commander	Input sensor controller	output to PID	output to PowerCalculator
Desired roll Desired pitch Desired yaw Desired thrust roll type pitch type yaw type	Euler roll Euler pitch Euler yaw Gyro x Gyro y Gyro z Acc X Acc Y Acc Z	Command Desired value Actual Value PID id	Roll Pitch Yaw Thrust

Table 4. Movement Controller model input and output messages.

information from previous calculations. The proportional part is obtained from the current error, the integrative part is obtained by adding all the previous errors and the derivative part is obtained doing finite derived with the current error and the last error. The PID models are abstracted from any semantic the error have (rate errors, angle errors, acceleration errors, etc). Then, the 6 PID models of the movement stabilizer (Roll angle PID, Pitch angle PID, Yaw angle PID, Roll rate PID, Pitch rate PID and Yaw rate PID) have the same behavior.

A difference between the original Crazyflie 2.0 controller and our approach, is the fact that in the original controller, the closed-loop frequency is hardcoded and the elapsed time used in the derivative part of the PID calculations are a fixed value obtained from the hardcoded information. In our approach, the elapsed time of the external function is used instead, and if some error occur in the frequency in which the sensors send data to the model or the frequency is changed, the PID model are not affected.

Once the goal of minimizing the error is achieved (with some error), the derivative and integrative part are no more valid because they refer to a previous error function. This is why, the PID models can receive as message a command to reset the values to zero, which is their initial state until they get the first error.

The Figure 14 is the DEVS graph of the PID models behavior. The model stay passivated until a request arrives. If the incoming request is a calculation request, then in the internal state, the PID calculation will be done and the model internal state integrative and derivative part will be updated for the next calculation request. Once the calculation is done, the result is send through the output function and the internal transition gets the model

passivated. If a reset request arrives, the model reset the integrative and derivative parts to zero and is passivated. The input and output messages of the PID models is shown in Table 5.

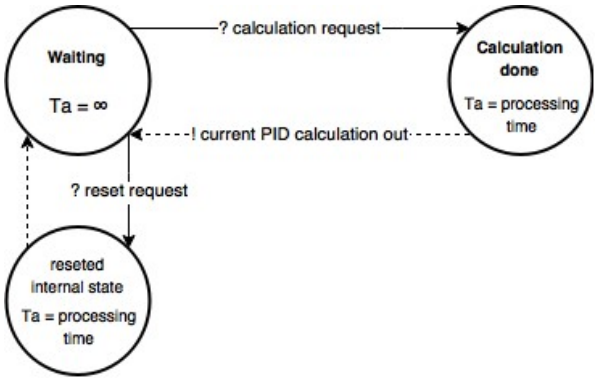

Figure 14. PID models DEVS graph.

Input	output
Command Desired value Actual Value PID id	PID id value

Table 5. PIDS model DEVS graph.

Power Calculator Atomic Model

As explained in the section two "Related works", there is a relation between the axes movements and the motor power. For each desired axes movement, a relation between the motors' power must be done. An axis movement can be seen as an axis rate, if there is a certain rate for a certain time in an axis, there is an axis movement. The powerCalculator model is in charge of combining the axis actuator values (the calculated PID rates) and the thrust to obtain the correct amount of power to send to the quad-copter motors.

As in the original Crazyflie 2.0 controller we have implemented two different relation systems between the actuators and the motor power, the QUAD_FORMATION_X and the QUAD_FORMATION_NORMAL, depending on the powerCalculator model initialization, it will use one or the other. Equation 1 and Equation 2 shows the relation system between the actuators and the motor power of both implementations.

The powerCalculator behavior shown in Figure 15 is similar of the PID models behavior. The only differences are that the powerCalculation has not reset request, and the fact that it does not update its internal state each time a calculation request arrives. Table 6 shows the input and output message of the powerCalculator model.

5.3. Implementation In CDBoost

The CDBoost simulator does not implement ports as is usual in DEVS simulators. Ports are filters that are used each time a model send a message to other models, when this happens, the sender, send the message through a port "p" and then the simulator dispatch the message to all the

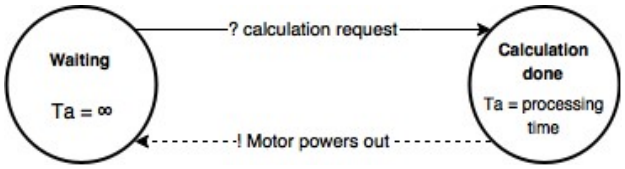

Figure 15. Power Calculator DEVS graph.

Equation 1. Relation system between the actuator values and the motor power using QUAD FORMATION X.

Equation 2. Relation system between the actuator values and the motor power using QUAD FORMATION NORMAL.

models that are connected to the sender as receptors and are listening through the port "p". The receptors also can implement different behavior for the incoming messages depending in which port the message has arrived. Since we do not have port, the messages sent by the sender model are broadcasted to all the receptors models connected to the sender. Because of this, the receptor models must filter and classify the incoming messages to determine if they must or must not process each message and to know how to process each message depending on its source.

The CDBoost simulator is implemented as a C++11 header library that is included in the project. The implementation of atomic models is done by extending the virtual class pdevs::atomic and implementing its virtual methods internal, advance, out, internal and confluence. Then, the new obtained class must be instantiated to get the atomic model objects that are handled by the simulator calling the implemented virtual methods in order to run the simulation.

In order to create the coupled models, we must instantiate the pdevs::flattened_coupled using as the constructor

parameter the models list, EIC, IC and the EOC. The model list is a vector of the models involved in the coupled model we are creating. The EIC is the External Input Coupling, which is the list of models that will receive messages from the coupled model input. The IC is the Internal Coupling, a list of pairs representing the connections between the models in the coupled model. The EOC is the external Output Coupling, similar to the EIC but connecting the models output to the coupled model output.

We have implemented a modelGenerator class in charge of instantiating the atomic models explained above and generating the coupled models. The modelGenerator is also in charge of reading the commander input and sensorData files (Figure 6 and Figure 7) and create the input_event_stream models. The input_event_stream models are used to implement the top model input for

Input	output
Roll Pitch Yaw Thrust	M1 M2 M3 M4

Table 6. Power Calculator model input and output messages.

computational simulation purposes, they read and parse a string and generate a timed sequence of events to send to the top model.

5.4. Implementation In Embedded CDBoost

In order to run the model in in the embedded simulator, ports and drivers must be implemented. This is part of the future work. Other people in the lab is currently working on it.

6. RESULTS

While developing the model, we have run and designed several test to verify our model. First, we have run compilation test for all our atomic models and for the top model. Once the models compile, we have run unit test to verify that the behaviour of our models were the expected.

To automatize the testing stage we have used the BOOST test module [27]. The main advantage of this approach is that, once the test is implemented, just running it, we directly know if the model has passed the test or not. We do not need to analyze anything manually. Moreover, if there is a failure in the test, this implementation approach tells us which part of the test did not success.

We have built unit test for the models that represent the behaviour of C++ functions in the Crazyflie 2.0: Q's updater, Euler transform, PID and Power calculation atomic models. The purpose of these unit tests is to compare the results provided by our atomic models against the ones provided by the Crazyflie 2.0 functions they are modelling. The Sensor distributor and the Movement controller are tested in the integration testing stage while testing the top model. All our atomic models has passed the following test

6.1. Qs Updater Unit Test

In the Qs Updater unit test we verify 3 things. Fist we check if the output message type is the one expected (i.e. if the output message is the type QS). Second, we check the proper implementation of the Mahony quaternion. For this purpose, we send several input messages with different values. We manually call the external, the output and the internal functions and store the output message. The output message is compared against the results provided by the Crazyflie 2.0 function we are modeling, when its parameters are the same as the ones our model receives. If the result is the same, our model passes the test. Otherwise, the test fails. Finally, we repeat this same procedure to test the Madgwick quaternion implementation.

6.2. Euler Transform unit test

In the Euler Transform unit test we verify 2 things. Fist we check if the output message type is the one expected (i.e. if the output message is the type EulerData). Second, we check the proper implementation of the Euler Transform function. For this purpose, we send several input messages with different values. We manually call the external, the output and the internal functions and store the output message. The output message is compared against the results provided by the Crazyflie 2.0 function we are modeling, when its parameters are the same as the ones our model receives. If the result differs in an amount less than 10^{-4}, our model passes the test. Otherwise, the test fails. We have set this error in the calculation to be flexible for numeric approximation errors while calculating the values.

6.3. PID unit test

In the PID unit test we verify 2 things. Fist we check if the output message type is the one expected (i.e. if the output message is the type PIDOut). Second, we check the proper implementation of the PID when calculation the angle and the rate. For this purpose, we send several input messages with different values. We manually call the external, the output and the internal functions and store the output message. The output message is compared against the results provided by the Crazyflie 2.0 function we are modeling, when its parameters are the same as the ones our model receives. If the result is the same, our model passes the test. Otherwise, the test fails.

6.4. Power Calculator unit test

In the Power Calculator unit test we verify 3 things. Fist we check if the output message type is the one expected (i.e. if the output message is the type MotorInput). Second, we check the proper implementation of the Power Calculator for the Quad_formation_X. For this purpose, we send several input messages with different values. We manually call the external, the output and the internal functions and store the output message. The output message is compared against the results provided by the Crazyflie 2.0 function we are modeling, when its parameters are the same as the ones our model receives. If the result is the same, our model passes the test. Otherwise, the test fails. Finally, we repeat this same procedure to test the Quad_formation_normal implementation.

Finally, before testing using real data from the Crazyflie 2.0, we have run a simulation using a mocked input values to check that our top model runs properly and generated the input for the motors. Figure 16 shows the results from this simulation.

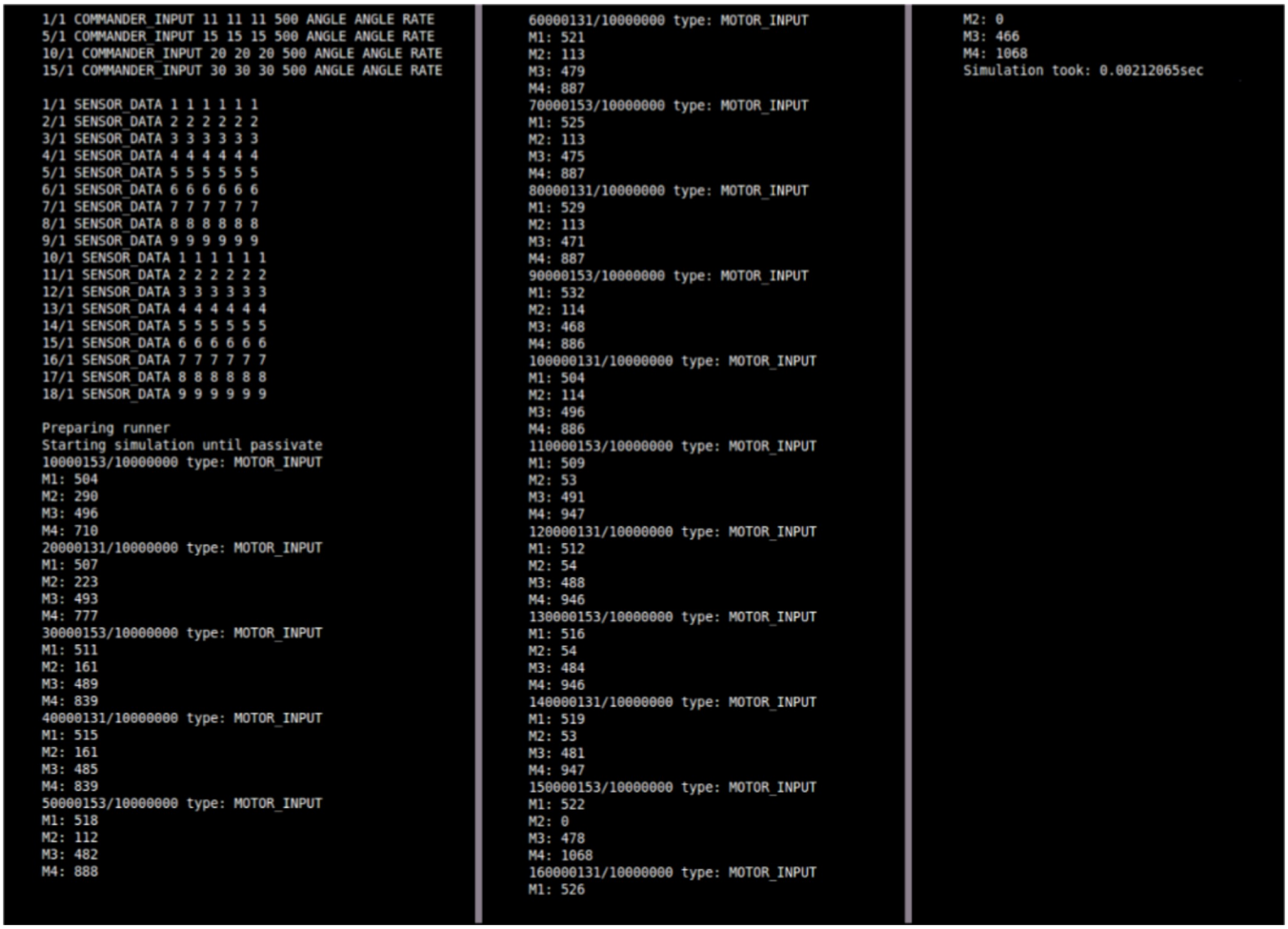

Figure 16. Simulation results using mocked input

As future work, the model needs to be tested using real data from the Crazyflie 2.0. Other people in the lab is working on it. Once the model is fully tested in simulation model, it has to be embedded in the hardware using Embedded CDBoost.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have developed a controller for a quad-copter based on the implemented in the Crazyflie 2.0 using DEVS and the DEMES design approach for embedded system. We have tested our atomic models and run simulations using mocked data to verify the calculations and the controller behavior.

As future work, we need to validate the model using real data from the quad-copter and test it embedded in the real hardware. This work attempts to show how can be DEMES used to implement a quad-copter controller and obtain the advantages of using a formal methodology. We have achieved a controller with a more realistic behaviour as for example, using the elapsed time to calculate the time between the last and the current iteration of the controller instead of hardcoding a value. In terms of validation and correctness, this difference makes our model more robust to errors than the original with hardcoded values. As future work, other people in the ARSLab-Carleton are embedding the model in the ECDBoost simulator, a free-OS kernel for embedded models in order to run the model in the Crazyflie 2.0.

Also the commander communication driver must be implemented in order to allow the quad-copter being controller from a remote controller as a joystick or a computer.

8. REFERENCES

1 L.M. Argentim, W.C. Rezende, P.E. Santos, R.A. Aguiar, PID, LQR and LQR-PID on a quadcopter platform, 2013 International Conference on Informatics, Electronics and Vision, ICIEV 2013. (2013).

2 J. Diebel, Representing attitude: Euler angles, unit quaternions, and rotation vectors, Matrix. 58 (2006) 1–35.

3 M. Euston, P. Coote, R. Mahony, J. Kim, T. Hamel, A complementary filter for attitude estimation of a fixed-wing UAV, 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS. (2008)

340–345.

4 A. Gibiansky, Quadcopter Dynamics , Simulation , and Control Introduction Quadcopter Dynamics, (2012) 1–18.

5 T. Henzinger, J. Sifakis, The embedded systems design challenge, in: FM 2006: Formal Methods, 2006: pp. 1–15.

6 B. Karlsson, Beyond the C++ standard library: an introduction to boost, 2005.

7 J. Li, Y. Li, Dynamic analysis and PID control for a quadrotor, 2011 IEEE International Conference on Mechatronics and Automation. (2011) 573–578.

8 Q. Li, C. Yao, Real-time concepts for embedded systems, CRC Press, 2003.

9 S.O.H. Madgwick, A.J.L. Harrison, R. Vaidyanathan, Estimation of IMU and MARG orientation using a gradient descent algorithm, IEEE International Conference on Rehabilitation Robotics. (2011).

10 D. Niyonkuru, Bare-Metal Kernels for DEVS Model Execution in Embedded Systems by, Carleton University, 2015.

11 D. Niyonkuru, G. Wainer, Towards a DEVS-based Operating System, in: Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, 2015: pp. 101–112.

12 D. Niyonkuru, G. Wainer, O. Dalle, Sequential PDEVS Architecture, in: DEVS '15 Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, 2015: pp. 165–172.

13 D. Niyonkuru, G.A. Wainer, Discrete-Event Modeling and Simulation for Embedded Systems, Computing in Science & Engineering. 17 (2015) 52–63.

14 a. L.L. Salih, M. Moghavvemi, H. a. F. a F. Mohamed, K.S.S. Gaeid, Modelling and PID controller design for a quadrotor unmanned air vehicle, IEEE International Conference on Automation Quality and Testing Robotics (AQTR). 1 (2010) 1–5.

15 D. Vicino, CDBoost simulator, (n.d.).

16 G. Wainer, CD++: a toolkit to develop DEVS models, Software: Practice and Experience. 32 (2002) 1261–1306.

17 G. Wainer, R. Castro, DEMES: a Discrete-Event methodology for Modeling and Simulation of Embedded Systems, Modeling and Simulation Magazine, April. 2 (2011) 65–73.

18 G. Wainer, E. Glinsky, P. Macsween, A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism, in: Model-Driven Software Development, Springer B, 2005: pp. 363–383.

19 B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems, Academic press, 2000.

20 Bitcraze Wiki, (n.d.). https://wiki.bitcraze.io/projects:crazyflie2:index

21 Bitcraze.io Crazyflie 2.0, (n.d.). https://www.bitcraze.io/crazyflie-2/

22 Bitcraze Wiki Crazyflie2 Architecture, (n.d.). https://wiki.bitcraze.io/projects:crazyflie2:architecture:index

23 Crazyflie syslink protocol, (n.d.). https://wiki.bitcraze.io/doc:crazyflie:syslink:index

24 FreeRTOS, (n.d.). http://www.freertos.org/

25 Bitcraze Wiki Virtual Machine, (n.d.). https://wiki.bitcraze.io/projects:virtualmachine:index

26 Bitcraze Wiki Crazyflie 2.0: Sensor and PID Control, (n.d.). https://wiki.bitcraze.io/doc:crazyflie:dev:fimware:sensor%7B/_%7Dto%7B/_%7Dcontrol

27 BOOST test module, (n.d.). http://www.boost.org/doc/libs/1_60_0/libs/test/doc/html/index.html