CONTROL OF A QUADCOPTER APPLICATION WITH DEVS

Cristina Ruiz-Martin
Ala'a Al-Habashna
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Dr.
Ottawa, ON, Canada
{cristinaruizmartin;alaaalhabashna;gwainer}@sce.carleton.ca

Laouen Belloli

Faculty of Natural and Exact Sciences
University of Buenos Aires
Intendente Güiraldes 2160
Buenos Aires, Argentina
laouen.belloli@gmail.com

ABSTRACT

Embedded systems are increasingly used to control different devices ranging from toys, up to vehicles and spaceships. Each machine has special hardware designed to perform its tasks optimally, and there are constrains that emerge from the relation between hardware, embedded software and the environment. Discrete-Event Modeling of Embedded Systems (DEMES) is a formal methodology to develop software for embedded systems using a discrete event formalism to implement the system software as a model, abstracting the software from the hardware and facilitating verification and validation. The objective of this paper is to show how to use DEMES to develop controllers for a quadcopter and to deploy them on the target hardware.

Keywords: DEVS, RT-DEVS, Modeling and Simulation, Embedded Systems, Quadcopter.

1 INTRODUCTION

An embedded system can be defined as "an engineering artifact involving computation that is subject to physical constraints" (Henzinger and Sifakis 2006), that is, a system involving some hardware and software in relation with an environment. Nowadays, these embedded systems are ubiquitous: in vending machines, cars, fridges, medical devices, factories, etc. They demand more features every day, which increases software complexity. The triple relation between the components of an embedded system (hardware, software and environment) is common in most computing systems. However, there is a main difference with embedded systems: a reaction to the physical environment is needed, and constrains in this reaction may imply specific deadlines, throughput and jitter, as in Real-Time Embedded Systems (RTES) (Q. Li and Yao 2003). The execution constrains may involve processors speed, power, hardware failure rates, etc. (Henzinger and Sifakis 2006). 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, the hardware and the environment.

Traditional approaches for embedded systems are divided into two different design phases: hardware and software. Then, both designs are merged doing a translation at least on some parts of the model. Moreover, these approaches do not cover hardware-software co-design, and the final design is hard to verify against the initial specification. Therefore, the development process usually involves long testing phases, error-prone products, and increased time to market (Niyonkuru 2015).

Most design methodologies for embedded systems are ad-hoc. This makes it hard to scale them up for larger systems. Additionally, they require complex testing efforts without guaranteeing a bug-free product. These deficiencies come from two main weak areas: the development cycle (different artifacts

©2019 Society for Modeling and Simulation (SCS) International

and tools are used) and the verification process (because of the discontinuities in the development cycle) (Niyonkuru and Wainer 2015a).

Lately, formal methods showed great potential in dealing with the above-mentioned issues (Wainer and Castro 2011). However, they remain hard to scale up and they usually do not take into account the physical environment that the embedded system controls. Model-based design techniques handle heterogeneity well and solve the scaling problem, but they lack formal modeling and effective model transformation (Niyonkuru and Wainer 2015a). Our proposed 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. Discrete-Event Modeling of Embedded Systems (DEMES) 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 (Zeigler, Praehofer, and Kim 2000), is a hierarchical and modular formalism for modeling Discrete Event Systems (DES). This formalism provides DEMES with the incremental-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 simulate them, without the need of the hardware and/or the need of the real environment.

Niyonkuru and Wainer developed two kernels based on DEMES: Embedded CD++ (E-CD++) and Embedded CDBoost (E-CDBoost) (Niyonkuru and Wainer 2015b; Niyonkuru 2015; Niyonkuru and Wainer 2015a) that allow users to run models directly on bare-metal without the need of an operating system. Both kernels differ in the communication mechanism used between model execution engines.

The objective of this paper is to show how to use DEMES to develop a controller for a quadcopter. We use E-CDBoost because it has shown better overhead performance than E-CD++ (Niyonkuru 2015). To achieve this objective, we will use, as case study, the development of a PID controller for Crazyflie2.0 (Bitcraze Wiki, Bitcraze.io Crazyflie 2.0), a small quadcopter mainly used for research, education and development. The main advantage of this model development technique is that the model 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 just matching the inputs and outputs of the model with the outputs and inputs needed by the hardware. It is important to highlight that only a preliminary version of the controller is deployed in the quadcopter due to proprietary firmware constraints.

The rest of the paper is organized as follows: in Section 2, we present the background related to this work. We also describe the methodologies used in this work and the tools used to implement and embed the model (CDBoost and E-CDBoost). In Section 3, we present the case study; the development of a PID controller for Crazyflie 2.0. Finally, in Section 4, we remark the conclusions and future research lines.

2 BACKGROUND

In this section, we present some related work on the development of quadcopter controllers. We also explain the methodologies and tools used in this work. We present DEVS as the formalism to model and simulate dynamic systems. We also introduce DEMES as the development methodology of real-time systems. Finally, we describe the main features to the tools we use to implement the model: CDBoost simulator and Embedded CDBoost.

2.1 Quadcopter controllers

A quadcopter controller is an embedded system used to control, in real time, the movement of an Unmanned Aerial Vehicle (UAV). It consists of four independent propellers, located orthogonally, as in Figure 1. Changing the speed of these propellers, the quadcopter can be moved in three different axes: roll, pitch and yaw.

There are many different control algorithms for quadcopters, and the selection of a specific controller depends on the combination of the hardware specifications and the development purposes for the quadcopter. An analysis and comparison between different algorithms is presented in (Argentim et al. 2013). In this research, we have used a Proportional–Integrative–Derivative (PID) controller (J. Li and Li

2011). A PID controller for a quadcopter is a closed-loop controller, where the main goal is to minimize the error between the current position of the quadcopter (yaw, pitch, roll and thrust) and the desired one.

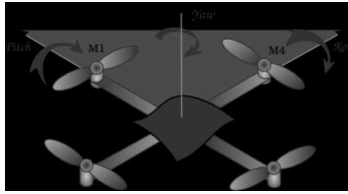

Figure 1. Quadcopter description.

The verification of embedded systems in general and of quadcopter controllers in particular is a hard task. Language-based (software-centric) or synthetic-based (hardware-centric) methodologies do not consider a hardware-software co-design and thus, it is difficult to compare the obtained product against the specifications. More recent methodologies, such as 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 MATLAB/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. (Salih et al. 2010) provides an example of abstract modeling in MATLAB/Simulink for quadcopter controller. Nevertheless, this kind model-based design normally lacks of precise semantic. Thus, the abstract model can be interpreted in several ways, making it hard to verify the final product against the abstract model.

Quadcopter controllers developed with traditional methodologies deal with the above-mentioned drawbacks. Example of these issues are explained in (Gibiansky 2012; J. Li and Li 2011; Salih et al. 2010; Argentim et al. 2013), where mathematical models of quadcopters are built, and validation is achieved by running the controller algorithms in the simulated quadcopter models. However, as soon as we need to embed those controllers in real quadcopter platforms, they must be translated to lower-level software and we also need to repeat the validation process.

Therefore, a formal method is needed to achieve correct verification between the abstract model and the final product. To do this, DEMES uses DEVS as the formal modeling formalism and by implementing computable models, the abstract model can be embedded, avoiding new bugs that may appear in the model translation. 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.

2.2 DEVS formalism

DEVS (Zeigler, Praehofer, and Kim 2000) is a hierarchical and modular formalism for modeling discrete-event systems. The hierarchical and modular structure of DEVS allows defining multiple models that are coupled to work together by connecting inputs and outputs. 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 defines the behavior of the system, and coupled, which describes the structure 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 decides 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 send an output through an output port.

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

Another advantage of DEVS for modeling the controller of a quadcopter 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.

2.3 Discrete-Event Modeling of Embedded Systems (DEMES)

DEMES is an M&S-based development methodology based on DEVS. It allows modeling embedded systems, the hardware and the environment. Moreover, it has the advantage that the model developed with be the embedded software in the final product. DEMES development cycle is depicted in Figure 2.

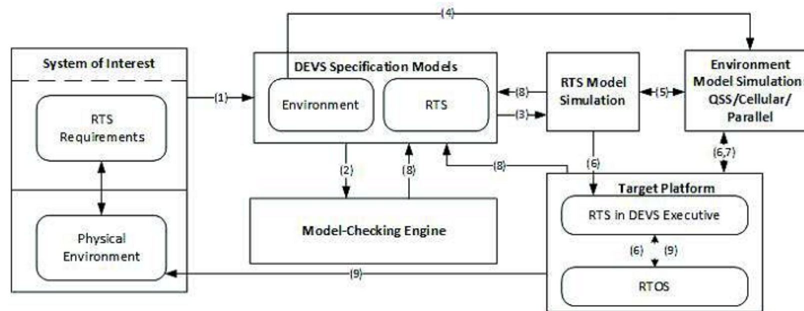

Figure 2. DEMES development cycle (taken from (Niyonkuru and Wainer 2015b)).

The process starts with the definition and modeling of the system of interest, both the Real Time (RT) System and its environment. These models can be done directly on DEVS or using another technique and then translated to DEVS (step-1). Once the DEVS models are developed, model checking is used for verification and validation of the model properties (step-2). At the same time, the RT-DEVS models are used to run the simulations in the simulated DEVS environment (step-3). We can also simulate the physical environment together with the RT-DEVS models (step-4). Sometimes the number of tests needed to do model checking grows exponentially; in this case, we can simulate the individual behavior of DEVS submodels using different scenarios of the environment (step-5). The tested submodels can be incrementally deployed in the target platform (step-6). A real-time executive executes the models on the hardware (step-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 (step-7) can be done using simulation, under a risk-free testing environment. DEMES allows doing design changes incrementally in a spiral cycle (step-8). The development cycle finishes with the RT System fully tested and every model deployed on the target platform.

DEMES combines the advantages of M&S with the methodological rigor provided by DEVS, and enables incremental construction of 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:** DEVS is the most general discrete event formalism and many techniques for embedded systems have been mapped into DEVS. Hence, we can use many modeling methods and translate them 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.

In order to implement a quadcopter controller using the DEMES methodology, we have used a kernel (Embedded CDBoost) that allows the users to run their DEVS models directly on bare-metal without the need of an operating system (Niyonkuru 2015) and a DEVS simulator (CDBoost)

2.4 CDBoost and Embedded CDBoost

In this project we used a new extended version of the CD++ toolkit (Wainer 2002) called CDBoost (Vicino 2015; Vicino et al. 2015), a DEVS simulator implemented in C++11. This simulator is cross

platform and it can also run in embedded platforms. The simulator is implemented as a header library that can be included in any C++11 project without any previous compilation. The only dependence is C++ Boost library (Karlsson 2005). CDBoost simulator is easy to install and fast to include in any project.

CDBoost architecture (Vicino, 2015) defines the PDEVSAtomic and PDEVSCoupled classes that extend the Model class. PDEVSAtomic has the virtual methods internal(), advance(), out(), external() and confluence(). We use these methods to implement the behavior of an atomic model. To implement a coupled model, we create a new object of PDEVSCoupled using vectors of models to define the EIC, IC and EOC. PDEVSSimulator is in charge of running simulations.

The Embedded CDBoost (E-CDBoost) engine is an extended version of CDBoost 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. E-CDBoost handles Real-Time (RT) execution of DEVS models. It provides a wall-clock time that communicates with an on board clock to check execution deadlines and thus, run the models in RT-mode. The model transitions occur in coordination with the time of the on-board clock. Therefore, time advance of models must be well defined not to generate scheduling problems by setting a transition time that violates the deadlines of the on-board clock.

A main difference between E-CDBoost and CDBoost is the model input/output system. CDBoost is only used for simulation purposes and then, the model input/output does not interact with any external device. 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 so. The output can be shown, saved or processed, but it is only semantic information. Instead, E-CDBoost interacts with sensors and actuators. To handle these interactions, E-CDBoost implements external ports and drivers (pDriver) that enable communication between the input/output of the top model and the devices. The pDrivers translate the message value of the model 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 simulation model.

3 CASE STUDY: A PID CONTROLLER FOR CRAZYFLIE 2.0

The Crazyflie 2.0 (Bitcraze Wiki, Bitcraze.io Crazyflie 2.0) is a 27g quadcopter easy to assemble. Its advanced functionalities and its easiness to fly indoors make it suitable for research purposes.

In this section, we present the characteristics of the Crazyflie, the model development and how to embed it on the hardware. Although in DEMES we have a step that is model checking, for this research project, we have implemented another testing strategy: we simulate the components of our model and we compare the results against the ones provided using the original firmware in the Crazyflie 2.0

3.1 Hardware Description

The Crazyflie 2.0 is an open source board equipped with:

- An EEPROM memory to store the configuration parameters.
- A 10-DOF IMU with accelerometer, gyro, magnetometer and a high precision pressure sensor.
- 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. The communications between the two microcontrollers are handled by the syslink protocol (Crazyflie syslink protocol).

A full description of the Crazyflie 2.0 architecture simplified can be found in (Bitcraze Wiki Crazyflie2 Architecture). The Crazyflie 2.0 software has three main components: the real-time operation systems (FreeRTOS), 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, the modules implement the behavior of the controller, which involves the treatment of the data from the sensor, the PID calculations for the quadcopter'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 (Bitcraze Wiki Virtual Machine).

The controller that is implemented inside the Crazyflie 2.0 works, basically, by initializing a group of tasks that are triggered by the FreeRTOS OS.

The Main tasks/modules are:

- **Stabilizer**: It executes a loop that iterates at a predefined frequency (currently 500Hz). On each iteration, it reads the data from the commander and from the sensor to get the desired and the actual values, and then it uses the controller module to calculate the new PID values. Once it gets these values, it calculates the correct amount of power to send to the motors.
- **Controller**: It calculates the rate and attitude of the PID calculations using the PID module
- **PID**: It handles the error between the desired value and the actual value, and the derivative, proportional and integrative values of this error. It uses these values, the desired value and the actual value to calculate the new PID value sent to the Stabilizer.
- **Command and IMU**: They are in charge of the communication of the controller with the commander (which set the desire value using the Euler orientation values (Diebel 2006)) and the sensor device (which gives raw data to calculate the actual position). The data used from the sensor are the accelerometer, the gyro and the magnetometer. The Euler orientation values must be transformed into rate values in order to be used by the PID controller.

There are also some debugging modules as the console and the log modules. The rest of the modules are not part of the quadcopter controller itself, but they are needed to communicate with the OS and to control different hardware as the led.

3.2 PID Model for the Controller using DEVS: TOP MODEL

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 related module. 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 quadcopter to fly.

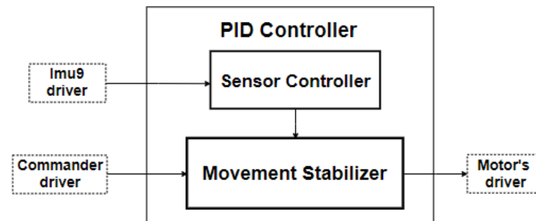

Figure 3. PID controller top model architecture.

Our DEVS model (Figure 3) takes the data from the sensors (actual values) using the Imu9 driver, and from the commander (desired values) using the Commander driver. It calculates the power to be send to the motors based on this data. The data is sent to the motors using the Motor's driver. We treat the data from the sensors in the same way as in the original project. This is defined in the Sensor Controller DEVS coupled model. We also implement the same PID calculations, without the hold-attitude mode feature (this feature keeps the quadcopter at a certain altitude fixed by the commander). This is defined in the Movement Stabilizer coupled model.

In contrast with the original Crazyflie 2.0 controller, our model does not implement a loop. Instead, it waits for the sensor to send the data. To achieve a close-loop at a predefined frequency, the driver that communicates the sensor with the model implements a polling mechanism with the correct frequency to read sensor data and translate the data into correct port value (i.e. the model receives input from the sensor periodically).

The Imu9 driver sends to the model the actual data that the sensors from the quadcopter are sensing. That is the x, y, z coordinates from both the gyro and the accelerometer. The Commander driver sends to the model the desired roll, pitch and yaw from the quadcopter and the desired thrust. Based on the above-mentioned 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 Motors' drivers, which connects the

model to the motors of the quadcopter. The output of our model is the power to each of the four motors of the quadcopter.

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.

Figure 4 shows the *Sensor controller* coupled model. It is composed by 3 atomic models: *Sensor distributor*, *Q's updater* and *Euler transform*. The *Sensor controller* model input is the data from the sensor (x, y, z coordinates from both the gyro and the accelerometer) and uses it to calculate the Euler roll, pitch and yaw from the quadcopter. The output of the model alternates between the raw data from the sensor and the Euler roll, pitch and yaw.

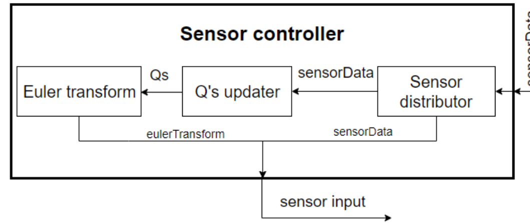

Figure 4. Sensor controller coupled model.

The *Movement Stabilizer* model is depicted in Figure 5. The *Movement controller* model is in charge of coordinating the six *PID* models and processes the results to send to the *Power calculation* model. Each time a new command is received, the desired value is updated. Additionally, each time there is an incoming message in the sensor input, a new iteration of the closed loop starts, 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 finished, the results are sent back to the *Movement Controller* that sends the information to the *Power calculation* model where the amount of power to send to each motor is calculated.

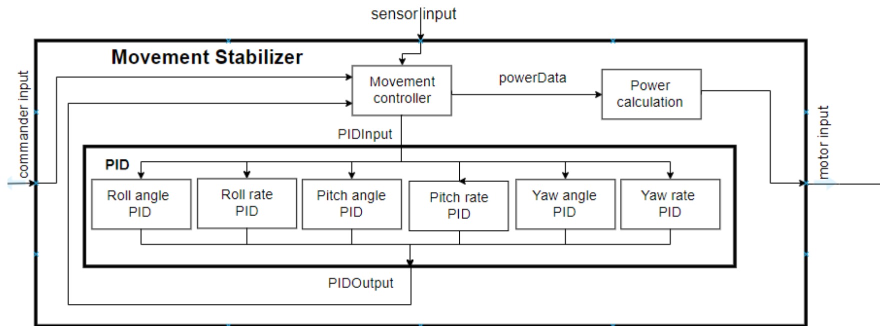

Figure 5. Movement stabilizer coupled model.

3.3 PID Model for the Controller using DEVS: Atomic Model Specifications

In this section, we explain the behavior of some of the atomic models that compose our PID controller. Due to lack of space, we only discuss the behavior of the *Sensor Distributor* atomic model.

The sensor distributor is in charge of managing the data from the sensors. Its DEVS graph is depicted in Figure 6. The model is initially passive in the "Waiting Sensor Data QS" state. When an input is received (i.e. gyro_x, gyro_y, gyro_z, acc_x, acc_y and acc_z), the *Sensor Distributor* move to the state "Sending Sensor Data Qs", the time advance of the state is updated to a processing time value and the information received is stored. After the processing time, the information stored (i.e. the input values) is sent through the output port Q's Updater. The model is update to a new state "Waiting Sensor Controller Data" and it passivates. When an input is received, the *Sensor Distributor* move to the state "Sending Sensor Data Controller", the time advance of the state is updated to a processing time value and the information received is stored. After this time, the information stored is send through the Sensor Controller output

port and the model passivates in the initial state "Waiting Sensor Data QS". This process continues every time an external event arrives.

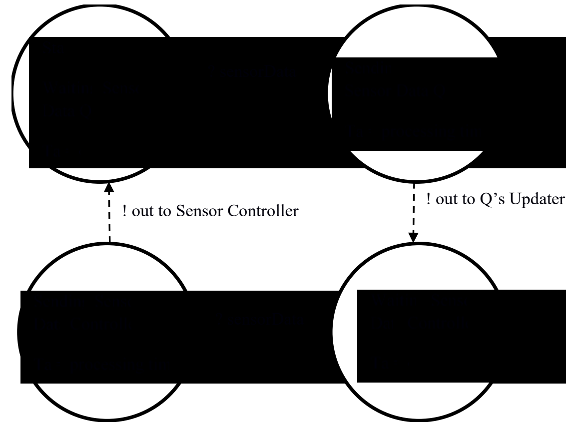

Figure 6. Sensor distributor DEVS graph.

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 the data sent to the sensor controller is first received by the sensor distributor for intermediate processing. Thereafter, sensor data is sent to the Qs Updater atomic model, which implements the orientation algorithm to estimate the orientation of the quadcopter.

The rest of the atomic models of our controller are defined in a similar way also using DEVS.

3.4 Implementation in CDBoost

Once all the atomic and coupled models are defined, we implement them in CDBoost to have a computer model to test and verify. To be able to use the model in simulation model, we also need to define the PDrivers imu9 and commander as DEVS atomic models. We define them as *input_event_stream* models that read the inputs from text files. The *input_event_stream* models are used only for computational simulation purposes to simulate the sensors and the commander as stated in the DEMES M&S methodology.

3.5 Results in Simulation Mode

As a part of the development phase of our model, we have designed and run several tests to verify our model although we did not performed model checking as suggested by DEMES. Instead, we have designed unit tests to verify that the behavior of our models is the expected one.

To automatize the testing stage we have used the BOOST test module. The main advantage of this approach is that, once the test is implemented and executed, its results will be automatically analyzed and the component verified. Moreover, if a test fails, this implementation approach tells us which part of the test did not pass.

We have built unit tests for the following submodels (which represent the behavior 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 corresponding Crazyflie 2.0 functions. The *Sensor distributor* and the *Movement controller* are tested in the integration-testing stage while testing the top model.

We discuss the Qs Updater Unit Test as an example. In the Qs Updater unit test, we verify three aspects of the model:

- First we check if the output message type is the one expected (i.e. if the output message is the type QS).

- Second, we verify the implementation of the Mahony quaternion sending several input messages with different values. We manually call the external, the output and the internal functions and store the output message. Each output value is compared against the result provided by the Crazyflie 2.0 function we are modeling. If the generated results are always equal to the expected ones, our model passes the test. Otherwise, the test fails.
- Finally, we repeat the same procedure to test the Madgwick quaternion implementation.

Once we finished the unit test, and before moving to the tests on the embedded version, we have run simulations using artificial input values to check that our top model runs properly and generates the input for the motors. Figure 7 shows the results from this simulation.

```
model input sensor:
1/1 SENSOR_DATA 1 1 1 1 1 1
2/1 SENSOR_DATA 2 2 2 2 2 2
3/1 SENSOR_DATA 3 3 3 3 3 3
4/1 SENSOR_DATA 4 4 4 4 4 4
5/1 SENSOR_DATA 5 5 5 5 5 5
6/1 SENSOR_DATA 6 6 6 6 6 6
7/1 SENSOR_DATA 7 7 7 7 7 7
8/1 SENSOR_DATA 8 8 8 8 8 8
9/1 SENSOR_DATA 9 9 9 9 9 9
10/1 SENSOR_DATA 1 1 1 1 1 1
11/1 SENSOR_DATA 2 2 2 2 2 2
12/1 SENSOR_DATA 3 3 3 3 3 3
13/1 SENSOR_DATA 4 4 4 4 4 4
14/1 SENSOR_DATA 5 5 5 5 5 5
15/1 SENSOR_DATA 6 6 6 6 6 6
16/1 SENSOR_DATA 7 7 7 7 7 7
17/1 SENSOR_DATA 8 8 8 8 8 8
18/1 SENSOR_DATA 9 9 9 9 9 9

model input commander:
1/1 COMMANDER_INPUT 11 11 11 500 ANGLE ANGLE RATE
5/1 COMMANDER_INPUT 15 15 15 500 ANGLE ANGLE RATE
10/1 COMMANDER_INPUT 20 20 20 500 ANGLE ANGLE RATE
15/1 COMMANDER_INPUT 30 30 30 500 ANGLE ANGLE RATE

Creating atomic model pf_sensor
atomic model pf_sensor created
Creating atomic model pf_commander
atomic model pf_commander created
Preparing runner
Starting simulation until time: 500/1seconds
10000153/10000000 type: MOTOR_INPUT
M1: 504
M2: 290
M3: 496
M4: 710
20000131/10000000 type: MOTOR_INPUT
M1: 507
M2: 223
M3: 493
M4: 777
30000153/10000000 type: MOTOR_INPUT
M1: 511
M2: 161
M3: 489
M4: 839
40000131/10000000 type: MOTOR_INPUT
M1: 515
M2: 161
M3: 485
M4: 839
50000153/10000000 type: MOTOR_INPUT
M1: 518
M2: 112
M3: 482
M4: 888
60000131/10000000 type: MOTOR_INPUT
M1: 521
M2: 113
M3: 479
M4: 887
70000153/10000000 type: MOTOR_INPUT
M1: 525
M2: 113
M3: 475
M4: 887
80000131/10000000 type: MOTOR_INPUT
M1: 529
M2: 113
M3: 471
M4: 887
90000153/10000000 type: MOTOR_INPUT
M1: 532
M2: 114
M3: 468
M4: 886
100000131/10000000 type: MOTOR_INPUT
M1: 504
M2: 114
M3: 496
M4: 886
110000153/10000000 type: MOTOR_INPUT
M1: 509
M2: 53
M3: 491
M4: 947
120000131/10000000 type: MOTOR_INPUT
M1: 512
M2: 54
M3: 488
M4: 946
130000153/10000000 type: MOTOR_INPUT
M1: 516
M2: 54
M3: 484
M4: 946
140000131/10000000 type: MOTOR_INPUT
M1: 519
M2: 53
M3: 481
M4: 947
150000153/10000000 type: MOTOR_INPUT
M1: 522
M2: 0
M3: 478
M4: 1068
160000131/10000000 type: MOTOR_INPUT
M1: 526
M2: 0
M3: 474
M4: 1068
170000153/10000000 type: MOTOR_INPUT
M1: 530
M2: 0
M3: 470
M4: 1068
180000131/10000000 type: MOTOR_INPUT
M1: 534
M2: 0
M3: 466
M4: 1068
Finished simulation with time: infsec
Simulation took: 0.00219148sec
```

Figure 7. Simulation results of the developed DEVS model with an artificial input.

3.6 Implementation in Embedded CDBoost

The ultimate goal of this work is to employ the DEVS model discussed above as the actual controller of the Crazyflie 2.0. We faced various hardware-related challenges (e.g., reading the sensor data) while trying to employ our DEVS model. As such, we have decided to start with a simpler DEVS model (referred to as flightDEVS – see figure 8) that takes as input the desired values each time and use them to control the motors of the quadcopter. Despite the simplicity of the model, employing such model as the control system of the Crazyflie 2.0 would verify that a DEVS model can be deployed (using ECDBoost) as a firmware that runs directly on the target quadcopter hardware without the need for an operating system. The use of more advanced models (such as the model discussed in the previous sections) with our bare-metal kernel will be implemented in future work.

Figure 8. The flightDEVS atomic model.

The DEVS flight control (flightDEVS) is composed of a single atomic model called flightDEVS. The model has one input port (Command port) and four output ports (e.g., Motor Port1). Here, the input values are fed to the input ports from a set of predefined values. The behavior of the model is as follows; it receives an input event holding a thrust power levels for the rotors, stores the received values, and sends them out through the output ports. In more detail, when an input event/value is received via the Command

port, the model's external function is called. The external function reads the payload carried by the input event, containing the thrust power levels, stores the values and sets the internal time advance to a very short timeout value. This is to simulate a near-instant reaction time. When the time advance expires, the output function is called which sends the thrust power levels designated for each rotor of the quadcopter through the output ports. Then, the internal transition function is called, which passives the model, to wait for the next input event.

In order to run our flightDEVS model on the Crazyflie 2.0 quadcopter, the DEVS-based system was integrated into the Crazyflie 2.0 firmware solution. To implement that, the code of our DEVS model was first employed as part of the Crazyflie 2.0 firmware. Then, the Crazyflie 2.0 flight control system initialization was modified to bypass the calls that start-up the RTOS and schedule the flight control and any other application as tasks. The system initialization was re-written to initialize the DEVS-based model on ECDBoost instead. This way, our DEVS model replaces the Crazyflie 2.0 flight control system, and ECDBoost replaces the functionality of FreeRTOS in running the DEVS simulation without any RTOS scheduling intervention. Figure 9 shows the new Crazyflie 2.0 system architecture.

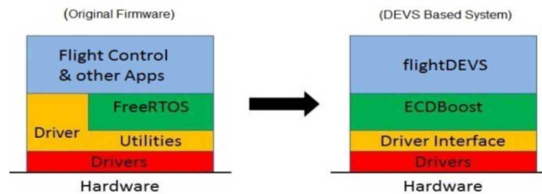

Figure 9. The old system architecture vs. ECDBoost-based system.

The figure shows that a Driver Interface has been added. It serves as a bridge between the DEVS-based system's port objects and the hardware drivers from/to which data is sent. For instance, the MotorPort object calls to the Driver Interface in order to activate the rotors of the Crazyflie 2.0 at the requested thrust power level. The input thrust levels were provided by an array of values that were fed into the Command Port.

3.7 Results in Embedded Mode

The DEVS-based system made of our flightDEVS model running on ECDBoost was simulated. Simulation results are shown in Table 1.

Table 1. Simulation results of the flightDEVS model.

Input	Output
[TIME 00:00:00:203:000] [INPUT 10]	
	[TIME 00:00:00:213:000] [MOTOR 3] [THRUST 10] [TIME 00:00:00:214:000] [MOTOR 2] [THRUST 10] [TIME 00:00:00:215:000] [MOTOR 1] [THRUST 10] [TIME 00:00:00:216:000] [MOTOR 0] [THRUST 10]
[TIME 00:00:00:406:000] [INPUT 20]	
	[TIME 00:00:00:416:000] [MOTOR 3] [THRUST 20] [TIME 00:00:00:417:000] [MOTOR 2] [THRUST 20] [TIME 00:00:00:418:000] [MOTOR 1] [THRUST 20] [TIME 00:00:00:419:000] [MOTOR 0] [THRUST 20]
[TIME 00:00:00:609:000] [INPUT 30]	
	[TIME 00:00:00:619:000] [MOTOR 3] [THRUST 30] [TIME 00:00:00:620:000] [MOTOR 2] [THRUST 30] [TIME 00:00:00:621:000] [MOTOR 1] [THRUST 30] [TIME 00:00:00:622:000] [MOTOR 0] [THRUST 30]

The new Crazyflie 2.0 full flight system solution was also executed on the quadcopter. The same sequence of events used in the simulations was also employed in the experimentation phase. The only difference is that larger thrust values were used when running the ECDBoost firmware on the quadcopter.

Results have shown that the ECDBoost-based firmware can be used to operate the quadcopter successfully. A video of the experiment was taken (ARSLab 2018). As can be seen in the video, the quadcopter takes off and mid-air change of thrust events occur during the experiment.

4 CONCLUSIONS AND FUTURE WORK

In this work, we explain how to use DEMES to develop a controller for a quadcopter and to embed it on the Crazyflie 2.0. The quadcopter controller was tested in simulation mode (both unit test and integrated tests) to verify their generated behavior. When deploying the model on the hardware, we faced issues because the gyroscope sensor was embedded on the firmware, and it cannot be moved to the hardware. We faced this issue because we choose hardware with firmware inside that we need to remove to run out of the OS. Due to these issues, a preliminary version of the model (flightDEVS) that does not include reading data from the sensors was deployed.

Using DEMES for quadcopter controllers has the following advantages:

- The sensors can be modeled and simulated 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.

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

Future work involves exploring how to move the sensors from the firmware to the hardware. If we do not success in this process, as the model is hardware independent, we will select other hardware to deploy.

REFERENCES

Argentim, L., W. Rezende, P. Santos, and R. Aguiar. 2013. PID, LQR and LQR-PID on a Quadcopter Platform. 2013 *International Conf. on Informatics, Electronics & Vision*, 2013. Dhaka, Bangladesh

ARSLab. 2018. Simple DEVS Based Flight System on Crazyflie 2 0. https://www.youtube.com/watch?v=fFK-hGCas8A.

Bitcraze io Crazyflie 2.0. https://www.bitcraze.io/crazyflie-2/ Last Accessed 16-12-2018

Bitcraze Wiki. https://wiki.bitcraze.io/projects:crazyflie2:index Last Accessed 16-12-2018

Bitcraze Wiki Crazyflie2 Architecture. https://wiki.bitcraze.io/projects:crazyflie2:architecture:index Last Accessed 16-12-2018

Bitcraze Wiki Virtual Machine. https://wiki.bitcraze.io/projects:virtualmachine:index Last Accessed 16-12-2018

BOOST Test Module. http://www.boost.org/doc/libs/1_60_0/libs/test/doc/html/index.html Last Accessed 16-12-2018

Crazyflie Syslink Protocol. https://wiki.bitcraze.io/doc:crazyflie:syslink:index Last Accessed 16-12-2018

Diebel, J. 2006. Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors. *Matrix* 58: 1–35.

FreeRTOS. http://www.freertos.org/ Last Accessed 16-12-2018

Gibiansky, A. 2012. Quadcopter Dynamics, Simulation, and Control. http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics/ Last Accessed 16-12-2018

Henzinger, T., and J. Sifakis. 2006. The Embedded Systems Design Challenge. *FM'06 Proceedings of the 14th international conference on Formal Methods*, 1–15. Hamilton, Canada

Karlsson, B. 2006. Beyond the C++ Standard Library: An Introduction to Boost. Pearson Education,

Boston, Ma

Li, J., and Y. Li. 2011. Dynamic Analysis and PID Control for a Quadrotor. *2011 IEEE International Conference on Mechatronics and Automation*: 573–578. Beijing, China

Li, Q., and C. Yao. 2003. Real-Time Concepts for Embedded Systems. CRC Press.

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

Niyonkuru, D., and G. Wainer. 2015a. Towards a DEVS-Based Operating System. *Proc. of the 3rd ACM SIGSIM Confonference on Principles of Advanced Discrete Simulation*, 101–112. London, UK.

Niyonkuru, D., and G. Wainer. 2015b. Discrete-Event Modeling and Simulation for Embedded Systems. *Computing in Science & Engineering* 17, no. 5: 52–63.

Salih, A.L., M. Moghavvemi, H. A. F. Mohamed, and K.S. Gaeid. 2010. Modelling and PID Controller Design for a Quadrotor Unmanned Air Vehicle. *IEEE International Conference on Automation Quality and Testing Robotics (AQTR)* 1: 1–5. Cluj-Napoca, Romania

Vicino, D. 2015 CDBoost Simulator. https://gforge.inria.fr/projects/cdboost/ Last Accessed 16-12-2018

Vicino D., D. Niyonkuru, G. Wainer, and O. Dalle. 2015. Sequential PDEVS Architecture. *DEVS '15 Proc. of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, 165–172. Alexandria, VA, USA

Wainer, G. 2002. CD++: A Toolkit to Develop DEVS Models. *Software: Practice and Experience* 32, no. 13: 1261–1306.

Wainer, G., and R. Castro. 2011. DEMES: A Discrete-Event Methodology for Modeling and Simulation of Embedded Systems. *Modeling and Simulation Magazine*, April 2: 65–73.

Zeigler, B.P., H. Praehofer, and T.G. Kim. 2000. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. San Diego: Academic Press.

AUTHOR BIOGRAPHIES

CRISTINA RUIZ-MARTIN has obtained a Ph.D. in Industrial Engineering (University of Valladolid) and Systems and Computer Engineering (Carleton University). She is a Postdoctoral Fellow at the Department of SCE at Carleton. Her email address is cristinaruizmartin@sce.carleton.ca.

ALA'A AL-HABASHNA has obtained his PhD in Electrical and Computer Engineering from Carleton University, Ottawa, Canada. Currently, he is a Postdoctoral Fellow at the Department of Systems and Computer Engineering at Carleton University. His email address is alaaalhabashna@sce.carleton.ca

LAOUEN BELLOLI is a Ph.D. student at the Laboratory of applied artificial intelligence (LIAA) at the Department of Computer Science at University of Buenos Aires. His email is laouen.belloli@gmail.com

GABRIEL WAINER is a Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca.