Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

DEVS-OVER-ROS (DOVER): A FRAMEWORK FOR SIMULATION-DRIVEN EMBEDDED CONTROL OF ROBOTIC SYSTEMS BASED ON MODEL CONTINUITY

Ezequiel Pecker Marcosig
Juan I. Giribet

Departamento de Ingeniería Electrónica
FI, UBA and CONICET
Av. Paseo Colón 850
C1063ACV, Buenos Aires, ARGENTINA

Rodrigo Castro

Departamento de Computación
FCEyN, UBA and ICC, CONICET
Ciudad Universitaria, Pabellón 1
C1428EGA, Buenos Aires, ARGENTINA

ABSTRACT

Designing hybrid controllers for cyber-physical systems raises the need to interact with embedded platforms, robotic applications being a paradigmatic example. This can become a difficult, time consuming and error-prone task for non-specialists as it demands for background on low-level software/hardware interfaces often falling beyond the scope of control designers. We propose a simulation-driven methodology and tool for designing hybrid controllers based on a model continuity approach. The simulation model of a controller should evolve transparently from a desktop-based mocking up environment until its final embedded target without the need of intermediate adaptations. DEVS-over-ROS relies on the DEVS framework for robust modeling and real-time simulation of hybrid controllers, and on the ROS middleware for flexible abstraction of software/hardware interfaces for sensors and actuators. We successfully tested DoveR in a case study where a custom-made crafted robotic system is built concurrently with the design of its controller.

1 INTRODUCTION AND MOTIVATION

The efficient development of embedded controllers for Cyber-Physical Systems (CPS) is under unprecedented pressure, driven by an upsurge of markets combining the Internet of Things and flexible production systems (Marwedel 2018). Salient properties in this setting are the uncertainty and rapid evolution of requirements, pushed by a pervasive availability of cheap, yet powerful technology for sensors, actuators, and embedded computing platforms.

For decades the software engineering community has spent a tremendous effort in creating formal methods and tools to develop controllers for embedded systems, in particular for those of hybrid nature and with real-time constraints. Modern design methods for hybrid controllers tend to rely on unified modeling frameworks, which capture and combine together the expressive power of well-known modeling techniques such as hybrid automata (see, for instance, Branicky et al. 1998 and references therein). Yet, most existing methods are still heavy, expensive and hard to scale up for real applications. Model- and simulation-based techniques offer increasingly attractive capabilities to allow for rapid, yet robust prototyping and final delivery of embedded controllers (Wainer and Castro 2011).

In particular the Discrete EVent Systems Specification (DEVS) (Zeigler et al. 2000) combines discrete event, discrete time, and continuous dynamics under a mathematically sound way. DEVS models are block-based units of self-contained behavior, which can be interconnected through input/output ports to create modular and hierarchical topologies of blocks. DEVS-based methodologies for M&S-driven engineering have been successfully implemented by integrating software development best practices to offer product life-cycle control. This becomes particularly relevant in scenarios where it is difficult to predict systems behavior as changes are introduced very frequently (Bonaventura et al. 2016). The main challenges faced

are system complexity, tight delivery times, the quality and flexibility of the developed models and tools, communication of results in interdisciplinary teams, and big data-scale analysis.

Our approach aims at an end-to-end methodology for designing hybrid controllers based on *model continuity* for DEVS (Hu and Zeigler 2004). Our main idea is that a DEVS simulation model for a controller can evolve transparently from a desktop-based mocking up environment until its final embedded target without the need of intermediate recoding nor reimplementation. In this context, the DEVS simulation engine plays the role of a "virtual simulation machine" for the DEVS model. Model continuity is a sound approach to mitigate the introduction of errors in the development process of controllers. Model continuity is becoming an increasingly recognized strategy (Cicirelli et al. 2018) where the DEVS framework is also seen as a candidate technology for CPSs (recently validated by Pecker Marcosig et al. 2017).

Previous works in the DEVS community (Yu and Wainer 2007; Castro et al. 2009; Castro et al. 2012; Moncada et al. 2013; Niyonkuru and Wainer 2015) dealing with embedded simulation adopted varied approaches to solve the real-time management or the coordination between the simulation executive and the underlying hardware (e.g., with or without an underlying operating system). Yet, we believe that the current weakest link in these efforts supporting model continuity lies in the connection between models and lower level sensors and actuators. This link represents a key interface layer between software abstractions and the physical platform. Typically, device drivers should cope with this requirement. Yet, being low-level platform-specific software artifacts, reprogramming or replacing device drivers can be a heavy process that demands specialized skills (often beyond those found with control designers).

In this work we propose and develop DoveR, a conceptual and practical framework helping to remove this obstacle, by combining real-time DEVS model simulation with the Robot Operating System (ROS) (Quigley et al. 2009), a massively adopted middleware library for developing robotic applications.

This paper is organized as follows. Section 2 introduces the DEVS formalism and the ROS middleware. Section 3 details the proposed methodology. The robot where we have conducted the experiments is presented in Section 4. Section 5 presents the DEVS simulation toolkit along with DoveR. Experiments conducted on the robot are depicted in Section 6. Finally, Section 7 provides concluding remarks and future work.

2 BACKGROUND

2.1 The Discrete EVent Systems Specification (DEVS) Framework

Modeling and Simulation of hybrid systems is central for understanding the behavior of discrete and continuous dynamics interacting with each other. This is because in most cases of practical interest hybrid systems don't accept analytical solutions.

DEVS is a formal model description framework equipped with an abstract simulation algorithm that is independent of the nature of the described system. It has been shown that DEVS can describe exactly any discrete system and approximate continuous systems with any degree of desired accuracy, therefore being capable to simulate all kinds of hybrid systems that undergo a finite amount of changes in finite intervals of time (Zeigler et al. 2000).

A system modeled with DEVS is defined as a modular and hierarchical composite of submodels, which can be either behavioral (Atomic) or structural (Coupled). Submodels interact by means of messages sent through input/output ports, in a block-oriented fashion. In terms of **behavior**, a DEVS **Atomic** model **A** is defined by the following tuple: $A = \{S, X, Y, \delta_{int}, \delta_{ext}, ta, \lambda\}$, where S is the set of internal states, X is the set of accepted input messages, and Y is the set of available output messages. Four dynamic functions define behavior: time advance $ta(s) : S \to \mathbb{R}_0^+$ is the lifetime of each state $s \in S$. After $ta(s)$ units of time an internal transition $\delta_{int} : S \to S$ is triggered (assuming no external input events arrived). In case an external event arrives, $\delta_{ext}(s, e, x) : S \times \mathbb{R}_0^+ \times X \to S$ (the external transition function) is triggered, with e being the elapsed time for a given state s and $0 \leq e < ta(s)$. Every time a new state s' is calculated (either by invoking δ_{int} or δ_{ext}), a new lifetime $ta(s')$ is calculated and the elapsed time e is reset to 0. Finally, $\lambda(s) : S \to Y$ is the output function that can be invoked to emit output messages.

In the sections below we show the combination of these dynamic functions and timing scheme with the proposed DoveR framework.

In terms of **structure**, a DEVS **Coupled** model **C** interconnects Atomic and Coupled components together through their input/output ports. It can be described by the following tuple: $C = \{X, Y, D, EIC, EOC, IC, Select\}$, where: X and Y are sets of input and output messages respectively, D is the set of components names, IC is the set of internal couplings among members of D, EIC is the external inputs coupling relation (links between external input ports and internal components) and EOC is the external output coupling relation. $Select$ is a tie-breaking function to assign execution priorities when several internal or external transition functions are scheduled for the same simulation time. The DEVS formalism is closed under coupling: any hierarchical composition of DEVS atomic and coupled models defines an equivalent atomic DEVS model. DEVS is by definition an asynchronous formalism, where each atomic model controls its own clock (time advance ta(s)). The composition of several atomic models into a larger coupled model preserves this timing independence.

This flexible coupling will permit a straightforward procedure for connecting or reconnecting pieces of the control-based model in the DoveR strategy. For practical modeling and simulation, we adopted the PowerDEVS toolkit (Bergero and Kofman 2010) which will be described below.

2.2 The Robot Operating System (ROS) Middleware

Despite its name, ROS is not an operating system. In fact, it is a middleware library that provides an abstraction layer on top of an operating system (Quigley et al. 2009). ROS allows both experts and non-experts for developing software for (but not limited to) robotic applications in a way that makes its use agnostic of low-level hardware layers. ROS provides common interfaces fostering reusability and code sharing. Even when there exist other middleware options, ROS is a largely accepted library, with a vast and growing open-source community.

A system running ROS is composed of several processes called *nodes*. ROS nodes are software applications that perform different functions. Owing to its modular structure, a node can be started or stopped at any time facilitating debugging. ROS allows nodes for communicating over networks in a seamless way. It provides two communication mechanisms between nodes: *Topics* and *Services*. Topics follow a publisher/subscriber architecture, allowing many-to-many one-way communication. Publishers and subscribers are not aware of the existence of each other. Unlike topics, services are suitable for synchronous request/reply interaction between nodes. Both topics and services use ROS *messages* which are strictly typed data structures.

ROS relies on a master node that keeps track of all the nodes on the network, and available services and topics to which a node can subscribe. If a node is interested in particular data it must subscribe to the corresponding topic. Then, every time a message is published on that topic, the callback function of every listening node is executed. ROS has a varied and growing set of tools that facilitate data and network topology visualization, managing node parameters and message transformations, measure topic bandwidth, just to mention a few. ROS is provided with bags used for data storage, thus real data can be later played back. We adopted the *kinetic* flavor of ROS due to its massive adoption and the fact that it is a long-term support version.

3 MODEL-CONTINUITY-BASED METHODOLOGY

We propose a reference blueprint to apply a simulation-driven methodology. The goal is to guide an end-to-end development process for embedded controllers relying on model continuity: the model shall never leave its simulation executive, from the mocking up stage until the final delivery stage, and no single line of code should be modified for the model when landing in its target robotic embedded platform.

Key assumptions made are: a) The target system is a robotic platform which can be itself under construction, i.e., the model of the plant itself is a potential source of errors, b) the robot is controlled

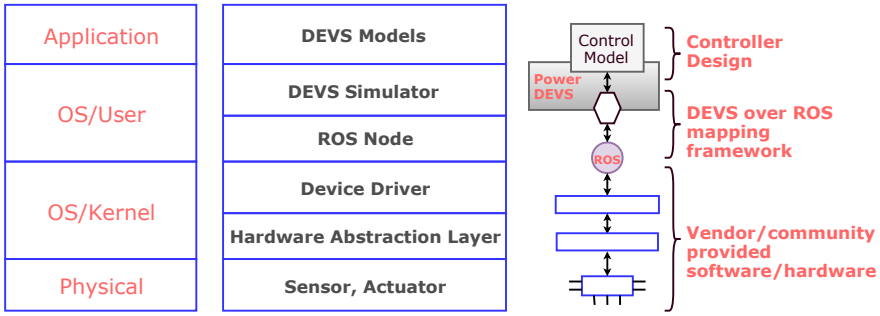

Figure 1: Abstraction layers.

by means of a single-board computer (SBC) capable of running a standard embedded Linux OS, c) the model development platform is a regular PC system running standard desktop Linux, and d) there is a local network infrastructure for the communication of the PC with the SBC using standard technology (Ethernet/WiFi).

In this work the concept of real-time is taken as soft real-time in a relaxed sense. There are several alternatives to tighten up this approach with more strict treatments of real-time. For instance, by adopting a DEVS simulator that treats explicitly the concept of message deadlines (Yu and Wainer 2007), or a DEVS simulator that relies on an RTOS (Bergero and Kofman 2010) providing worst case timing guarantees. They should only bring along better timing features to the final product. In this sense, we purposely stand on a "Commercial Off-The-Shelf" scenario, and study what can be achieved under this vanilla setting. In other words, we accept that the real-time simulation performs only a best-effort treatment of timing, where overrun situations could arise (i.e., the simulation clock falls behind the wall clock, see Cellier and Kofman 2006), possibly affecting the quality of the controller.

In order to apply our model-continuity-based methodology to problems arising in M&S of embedded systems, we rely on the ROS middleware to abstract away subtle issues of hardware and subsystems communication. Indeed, there exist a myriad of ROS packages freely available, developed by an active community for a variety of sensors and actuators. Typically, software systems depend on well-defined layers of abstraction (Figure 1), each one devoted to a particular duty and providing neighbor layers with a standard interface. ROS together with the Hardware Abstraction Layer (HAL) (provided by the OS kernel) can efficiently abstract lower-level layers from the user/application level. Meanwhile, hybrid controllers lie on the application layer. A DEVS-based simulation engine fills the gap in-between. We refer to the composition of DEVS over ROS as the DoveR middleware. In our work, DoveR is implemented by extending the PowerDEVS toolkit.

The overall solution can provide the control automation community with a simulation platform adequate to develop hybrid systems and capable of linking transparently with the underlying hardware. Even when there exist other projects which add an abstraction layer on top of ROS (Crick et al. 2017), we rely on PowerDEVS as it features modular interconnection of blocks for continuous and discrete models, an attractive and natural paradigm in the automation community.

Stage 1: Simulation-based Controller Design. We assume that we count on a model of the system to be controlled. It comes either from applying first principles from physics or from applying black-box identification techniques. The System Model represents the target real robotic system. Based on this System Model, our aim is to design a Controller Model in charge of controlling the plant inputs so that it behaves in a given desired way. Figure 2 shows the closed-loop between plant and controller models. This simulated scenario allows the engineer for testing different control strategies in a risk-free setting.

Stage 2: ROS and Communication Network in the Loop. The previous step could be thought of as an ideal stage where we are considering a perfect network between the plant and the controller. Not only delays, but also congestion, package discard, and any other non-idealities are introduced into the closed loop with the addition of a real communication network (Figure 2). Moreover, we are also incorporating the

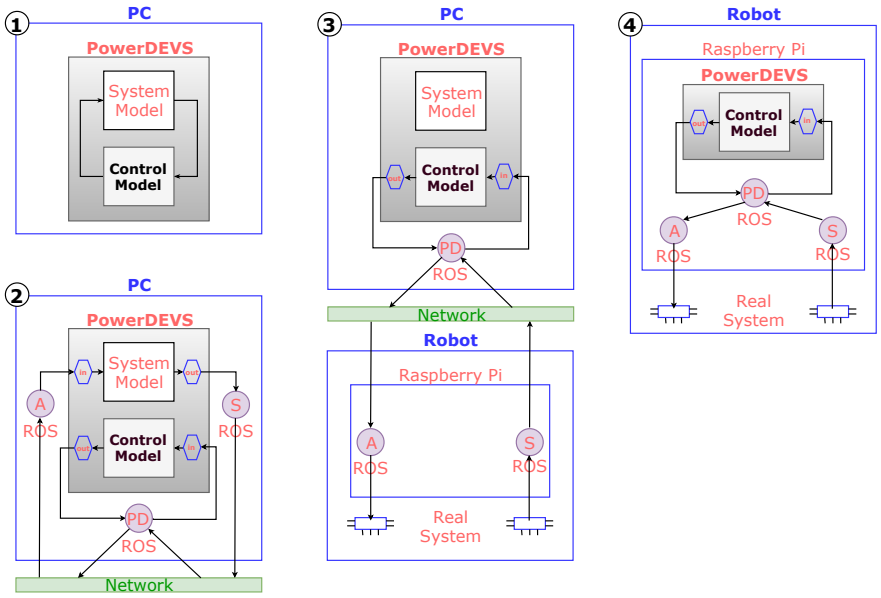

Figure 2: End-to-end model continuity. *Stage 1:* Standalone simulation-based controller design (DEVS toolkit), *Stage 2:* Adding DoveR: DEVS over ROS and network-in-the-loop, *Stage 3:* Connecting to target: robot-in-the-loop simulation, *Stage 4:* Embedded simulation: embedded controller with DoveR framework.

ROS middleware above the network, where PD, A and S refer to PowerDEVS, Actuator and Sensor ROS nodes respectively. Applied in this scenario the previously designed controller will probably need some adjustments in order to take these additional issues into account and go back to the preceding performance.

Stage 3: Remote Simulation of the Controller. The obvious next step is to get rid of the System Model and replace it by the robot itself (Figure 2). Then, we have to deal with non-modeled behavior that can strongly affect closed-loop performance. Even though we try to work with acceptable models, there will always exist some not-considered effects. As it was previously stated, we are providing a methodology for rapid prototyping. Therefore, it is not advisable to spend large amounts of time trying to develop a perfect model.

Stage 4: Embedded Simulation of the Controller on the SBC. Finally, the same Controller Model used throughout these steps and implemented on the simulator is run on the final embedded platform (Figure 2). For the simulated controller nothing has changed, since it sees the same interfaces as in the previous stages, as if it were already running on the PC. It can be thought of as a virtual machine running on the SBC.

4 ROBOT DESCRIPTION

TachoBot, the *Test Assembly for the Control of Hybrid Objectives* (Figure 3) is a custom-made experimental platform, crafted with off-the-shelf components. TachoBot is a 3-degree-of-freedom (3-DOF) vehicle designed for studying hybrid control strategies in cyber-physical systems. The robot features 8 lateral propellers to manage the position on the plane and the rotation angle around its vertical axis. Said actuators are mounted in pairs, thus resulting in 4 independent forces acting on the robot (Figure 4). A central propeller is in charge of considerably reducing the friction with the underlying surface, allowing for displacements even with extremely small control efforts. This actuator must lift a weight of around 3 kg.

4.1 Continuous Robot Model and Discrete Regulation Controller

The robot can be seen as a rigid body with 3-DOF subject to forces $\mathbf{F} = [f_1, f_2, f_3, f_4]^T$ (Figure 4). Therefore, dynamic equations describing its movement can be derived by applying Euler-Newton laws (1). The state

Figure 3: TachoBot.

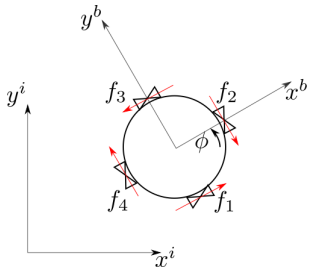

Figure 4: TachoBot model.

vector $\mathbf{x}(t) \in \mathbb{R}^6$ specifies the *pose* of the vehicle: $\mathbf{x} = [x, v_x, y, v_y, \phi, \omega]^T$, where x, y and ϕ are the linear position and the yaw angle respectively, and v_x, v_y, ω their corresponding speeds. Parameters M, R and I_z are respectively the mass, lever arm and moment of inertia of the TachoBot, whereas S and C terms stand for $\sin(\phi)$ and $\cos(\phi)$. $\dot{\mathbf{x}}$ is the time derivative of state vector $\mathbf{x}$.

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ C/M & S/M & -C/M & -S/M \\ 0 & 0 & 0 & 0 \\ S/M & -C/M & -S/M & C/M \\ 0 & 0 & 0 & 0 \\ R/I_z & -R/I_z & R/I_z & -R/I_z \end{bmatrix} \mathbf{F} \tag{1}$$

$$\mathbf{u}^{pid} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0.5 & -0.5 & 0.5 & -0.5 \end{bmatrix} \mathbf{F} \tag{2}$$

These quantities are more easily described in the non-inertial vehicle frame $(x^b - y^b)$ attached to the robot in Figure 4. However, given the yaw angle ϕ the velocity of the robot and its position in (1) are referred to the inertial frame $(x^i - y^i)$.

To control the state dynamics we rely on a controller to generate forces in $\mathbf{F}(t)$ such that $\mathbf{x}(t)$ tends to the reference $\mathbf{x}^{ref}(t)$. Without loss of generality, we consider 3 PID controllers, for the linear and angular positions, made up of the sum of terms proportional to the error $\mathbf{e} = \mathbf{x} - \mathbf{x}^{ref}$, its integral and derivative. The three PID outputs $\mathbf{u}_{pid} = \left[u_x, u_y, u_\phi\right]^T$ relate to $\mathbf{F} = [f_1, f_2, f_3, f_4]^T$ in (1) by means of transformation matrix T in (2). Since T is rectangular, in order to convert controller outputs to forces we use the Moore-Penrose pseudoinverse $T^\dagger$, which provides the control action with minimum energy.

4.2 Hardware Description

The control board is a Raspberry Pi 3 (Model B) computer which includes a quadcore 64 bit processor, 1 GB RAM, 32 GB external flash and a built-in WiFi module. It runs an Ubuntu Xenial (16.04) and communicates with sensors and actuators via an I^2C bus. Each force f_i in (2) is the product of a pair of DC coreless motors attached with propellers. They are controlled by individual pulse width modulation (PWM) signals generated with a PWM controller PCA9685. Zero thrust corresponds to both motors on a pair working at 50 %. DC motor drivers consist of power MOSFETs, while the central propeller uses a commercial electronic speed controller (ESC).

The angle and angular speed measurements are provided by an Inertial Measurement Unit (IMU) comprised of an accelerometer and a gyroscope. MPU9250 was selected due to the availability of its ROS node. Finally, two lithium polymer (LiPo) batteries are attached to the robot to supply energy.

5 POWERDEVS SIMULATOR

PowerDEVS is an open-source DEVS-based simulator appropriate for hybrid system modeling and real-time simulation (Bergero and Kofman 2010). A salient feature is its graphical design interface, where atomic and coupled models are represented by blocks that can be wired and interchanged by simple drag-and-drop actions. Therefore, it is suitable for a non-DEVS expert, hiding away the internals of the formalism.

PowerDEVS is also the flagship tool for the Quantized State System (QSS) family of numerical methods (Cellier and Kofman 2006) that can solve ordinary differential equations efficiently in the context of a discrete-event simulation. This way, by means of QSS methods PowerDEVS provides a unified framework to represent discrete and continuous dynamics in a hybrid system with a single coupled model, while featuring real-time simulation capabilities.

PowerDEVS is composed of various independent programs: the Model Editor, the Atomic Editor, the Preprocessor and the Simulation Interface. The first one is an IDE to edit and configure the simulation model, the second is a graphical editor for DEVS atomic models, the preprocessor is in charge of translating the model editor files to a standalone executive, and finally the graphical simulation control interface allows for user interaction with the simulation execution.

5.1 DoveR: Communication Between DEVS and ROS Using the PowerDEVS Simulation Toolkit

In order to establish a bridge between PowerDEVS and ROS we choose UDP network sockets. We avoid TCP because its connection-oriented nature imposes extra overhead of no value in robotic applications, where retransmission of lost messages is typically useless (new samples should preempt old ones).

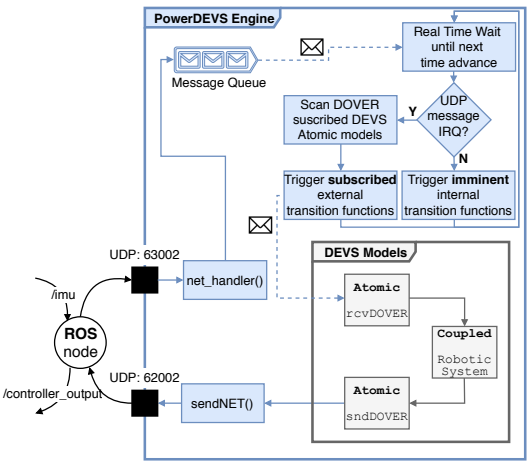

Figure 5: Real-time UDP message handling in the ROS/PowerDEVS interface.

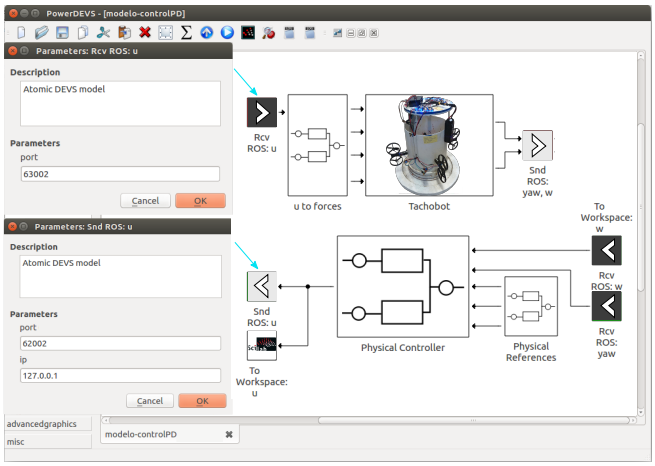

Figure 6: PowerDEVS screenshot. Dark gray (receive) and light gray (send) DoveR atomic models.

When performing real-time simulation (Cellier and Kofman 2006), internal events stick to wall-clock time. The root-coordinator in a DEVS framework keeps track of the atomic model presenting the earliest time advance about to expire (also known as the **imminent** models). While waiting for said earliest imminent time stamp, the DEVS root-coordinator is in idle state and has the chance to process messages incoming from the ROS middleware.

When the simulation starts, a DoveR atomic model requests for listening to a particular UDP port, and it is registered as a *listener* of that port. Each UPD port is assigned to an individual thread that remains listening throughout the entire lifetime of the simulation. Each UDP port and its associated thread can be shared by several atomic models that, upon request, are added to the list of listeners for that port. For interprocess communication between listener threads and the main PowerDEVS thread we rely on Linux IPC Message Queues. A single FIFO queue enqueues messages arriving to PowerDEVS, identifying the

UDP port number they come from. Once a message is popped out from the queue, the associated port is read and each atomic model subscribed as a listener is notified of the arriving message by triggering its DEVS external transition function (see Figure 5). Therefore, ROS messages play the role of external events in a DEVS framework, which consist of a value and a time stamp arriving at an atomic model.

```
void   sndDOVER::init(double t,...) {
va_list parameters;
va_start(parameters,t);
port = atoi(va_arg(parameters, char*));
...
sigma = INF;}
double sndDOVER::ta(double t) {return sigma;}
void   sndDOVER::dint(double t) {sigma = INF;}
void   sndDOVER::dext(Event x, double t) {
double *xv;
char msg[200];
xv = (double*) (x.value);
sprintf(msg,"%f",xv[0]);
sndNET(port, ip, msg, strlen(msg));
sigma = 0;}
Event sndDOVER::lambda(double t) { return ;}
```

Figure 7: sndDOVER atomic model functions.

```
void sndNET(int port, char* ip, char* data, int size
    ){
...
/* Create new socket. */
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
bind(sockfd, (struct sockaddr *) &local_addr,sizeof(
    struct sockaddr_in));
int result = sendto(sockfd,data,size,0,(struct
    sockaddr*)&remote_addr,sizeof(remote_addr));
close(sockfd);}
```

Figure 8: sndNET method (PowerDEVS Engine).

```
def main():
...
rospy.init_node('PowerDEVS_Control') #ROS node
    declaration
udpport_read_yaw=62002 #Port to read the Control
    Action for the yaw angle
udp_server_socket_yaw=socket.socket(socket.AF_INET,
    socket.SOCK_DGRAM) #UDP socket
udp_server_socket_yaw.bind(("",udpport_read_yaw))
#New ROS topic (where controller output will be
    published)
ROSpublisher = rospy.Publisher('controller_output',
    Twist,queue_size=10,latch=True) #ROS Publisher
    object
msg_read_list=[server_socket_yaw] # UDP ports to
    listen to
while True:
    read,_,_=select.select(read_list,[],[],0.001) #
    timeout = 0.001
    for s in read:
        yaw_ctrl_msg,addr = s.recvfrom(18) # UDP
    message with controller output from PowerDEVS
        rospy.loginfo("[PowerDEVS-Ctrl] Message
    received from PowerDEVS.")
        yaw_ctrl_msg = Twist() # create empty ROS
    message
        yaw_ctrl_msg.angular.z = float(yaw_ctrl) #
    fill in ROS message
        ROSpublisher.publish(yaw_ctrl_msg) # publish
     message to ROS topic
        rospy.loginfo("[PowerDEVS-Ctrl] Message
    succesfully published in ROS topic.")
continue
```

Figure 9: Excerpt of a ROS node for DoveR.

In contrast, outgoing messages from PowerDEVS (Figure 7) depart directly to a corresponding UDP port (Figure 8). ROS then receives a UDP datagram, conforms a proper ROS message, and publishes it on a corresponding topic (Figure 9). Note that `ta`, `dint`, `dext` and `lambda` functions in Figure 7 correspond to the time advance, internal and external transition and output functions defined in Section 2.1.

5.2 Embedded Simulation On a Raspberry Pi

We installed PowerDEVS on the Raspberry Pi board running Ubuntu Xenial 16.04. We generate the simulation model files on a PC and deploy them over the network into the SBC using `ssh`. Only the Preprocessor and the Simulation Interface modules of PowerDEVS are needed in the SBC. Likewise, we installed ROS-Base (without GUI tools). The PowerDEVS installation on the SBC proved quite easy after downloading it from the SourceForge repository. However, some precautions need to be taken: some libraries in the repository are precompiled for PC architecture (amd64 and i386) and must be re-compiled to run on an Armv7l architecture. Also the `qmake` package from the Qt library must already be installed. Finally, PowerDEVS uses the Scilab package as a numerical back end for complex mathematical operations (available for Debian and Ubuntu). After a successful installation, the BackDoor toolbox is required to open a communication channel between PowerDEVS and the Scilab workspace.

6 RESULTS

Henceforth, we will follow the steps described in Section 3 to develop a PID controller for the TachoBot using PowerDEVS running embedded in the Raspberry Pi. Experiments were conducted with the robot

mounted on a rotating platform in order to evaluate only the closed-loop angle behavior. The goal is to stabilize the angle around a reference value of 1 rad.

In Step 1 we close the loop in a purely simulated setting. We count on the model from Section 4 entirely derived from first principles. Since the working range of lateral propellers is around PWM at 50 %, it is valid to assume a linear relation between the PWM and the thrust and adopt a candidate value. Taking the model as is, we obtain Figure 10 and the resulting PID gains are registered.

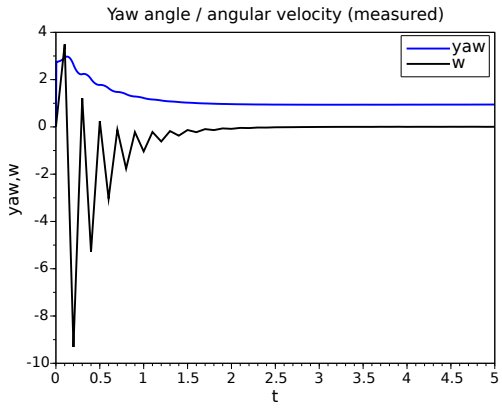

Figure 10: Simulation-based designed controller.

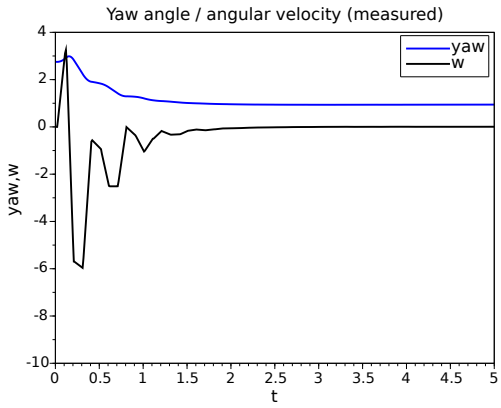

Figure 11: ROS and network in the loop.

Introducing a wireless network and the ROS middleware into the loop might demand additional tuning for PID in Step 2. A screenshot of the simulation model for this second step is shown in Figure 6. Resulting curves are in Figure 11 which differs from the ones in Figure 10 due to the delay introduced by adding two ROS nodes, one for PID controller and another connected to the robot model, running on the PC.

Indeed the delay introduced by the ROS middleware is in the order of 7 ms and can be appreciated in Figure 12, where a ramp signal generated by a PowerDEVS source block was sent back to PowerDEVS through a ROS loop. In this case, two nodes were considered: one attached to PowerDEVS and another acting as a loopback, both running on the PC. Since this delay is almost constant it can be compensated with classical techniques, such as a Smith predictor. In a next step, we moved both ROS nodes to the Raspberry Pi, incorporating a WiFi network in between. Consequently a bigger and more varying delay, in the order of 80 ms, can be seen in Figure 13.

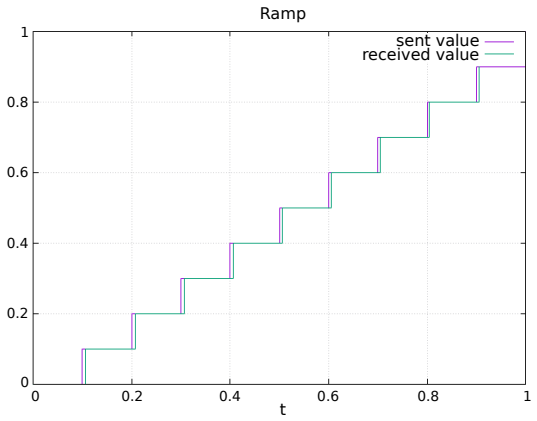

Figure 12: Measured local loop delay.

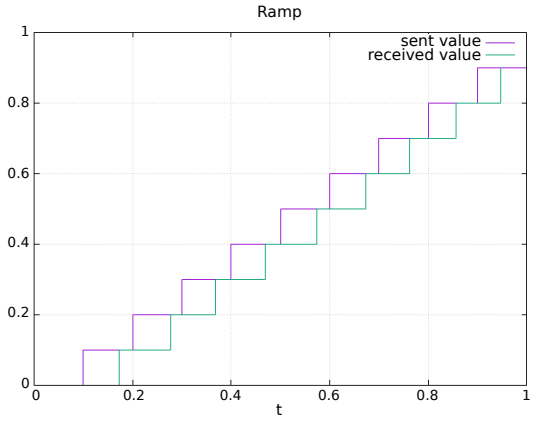

Figure 13: Measured remote loop delay.

Step 3 can be split into, at least, two substeps. First we need to test the validity of the model with the addition of network latencies, the ROS middleware and potential issues introduced by the hardware

and the operating systems. With this in mind, we supplied the robot actuators with a square wave and measured the resulting yaw angle and angular speed (Figure 14). Then, we fed the simulation model of the robot with the same square wave and recorded the resulting data. Then, we made additional adjustments on model parameters (such as the robot mass) such that the simulation in Figure 15 and the measurements looked quite similar. The model parameter adjustments led, in turn, to re-tuned PID control gains.

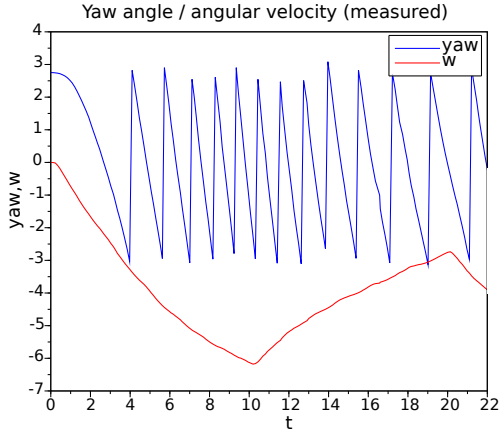

Figure 14: Measurements.

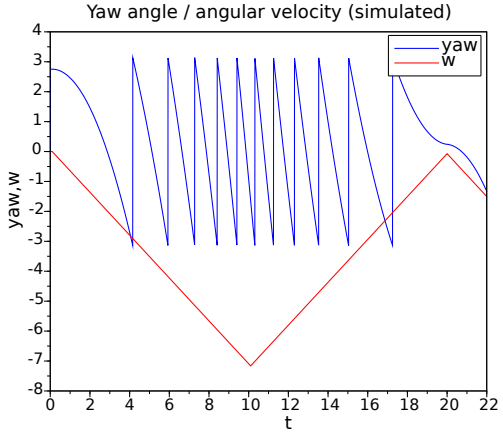

Figure 15: Simulation.

Finally, we embedded the PID controller on the Raspberry Pi. The resulting curves for Step 4 can be seen in Figure 16. At the beginning, a delay can be noticed of about 2 s due to initialization issues. It is followed by a non-minimum phase response where the robot starts moving in the opposite direction. This is likely due to mechanical issues not captured by the original model (later, a malfunctioning motor was detected). From around 4 s onwards the robot response behaves as expected.

Between 8.3 s and 10.4 s the yaw angle remained stuck with zero angular speed due to the effect of static friction of the rotating platform. At the same time, the control effort (Figure 17) started to increase with a positive slope because of the PID's integral term. Then, the robot released as soon as it was able to overcome the friction. From then on, the angle curve appears to be underdamped. Around 20.5 s the angle got stuck again with zero speed close to the reference, where neither the error nor the control action are null.

This process showed that we managed to successfully apply the methodology proposed based on the DoveR middleware, obtaining in a very straighforward way a yaw angle controller embedded in the final platform. The process also helped to learn that the rotating platform affects robot behavior since it adds a static friction term not considered in the model in Section 4. However, the PID controller showed its robustness against unmodeled effects. From this point on, a second iteration of the full methodology can be applied from Step 1, if desired, in order to enrich the system model for capturing more dynamics and develop incrementally a more robust controller.

7 CONCLUDING REMARKS AND FUTURE WORK

We presented DEVS-over-ROS (DoveR), a software framework that fosters simulation-model continuity to develop hybrid embedded controllers for cyber-physical systems, with a focus in robotic applications. The framework is accompanied with a sample end-to-end methodology where the simulator plays the role of a real-time virtual machine for the simulation models. Thus, the desired controller model can evolve transparently from a desktop-based mocking up environment until its final embedded target without the need of intermediate recoding nor reimplementation.

We adopted the DEVS framework for modeling and simulation of hybrid systems, that has been proposed before as a technology for model continuity in embedded systems. Yet, the issue of dealing with low

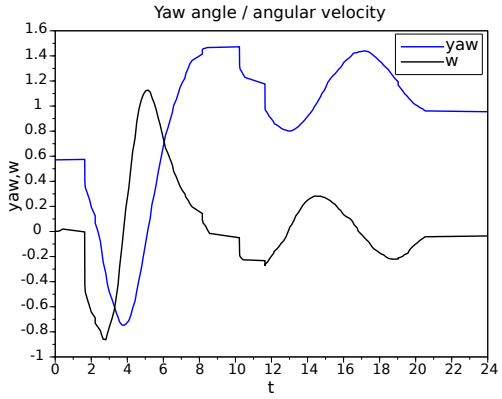

Figure 16: Embedded controller.

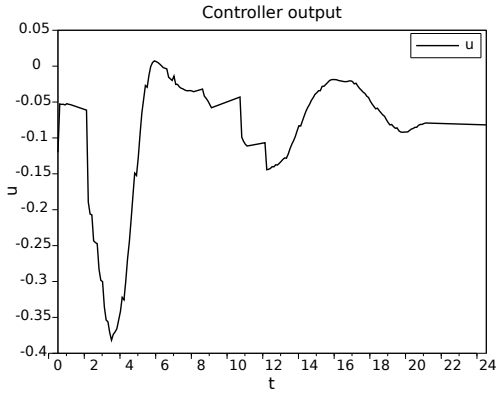

Figure 17: Controller output.

level intricacies at the hardware/software interface remained as a bottleneck for flexible simulation-driven development. We selected the ROS middleware to mitigate this problem, and make the hardware/software interface as modular, reusable, and transparent as possible to both a DEVS simulator and a DEVS modeler.

The control designer is then free to study the performance of a candidate controller by choosing its favorite workflow, without a need of being a DEVS expert or a ROS developer. A typical workflow can include to close the control loop in a purely simulated context (a non real-time PC), or retain the controller at the PC while interacting (now in real-time) over a network with the real robot, or to directly deploy the controller to the target embedded platform for full real-time embedded control of the real system. The ROS nodes can be easily moved back and forth between the platforms to assess timing constraints and the effects of the introduced latencies and data overhead.

We tested the DoveR strategy and tool in a case study where a custom made, crafted robotic system is being built concurrently with the design of its controller. We successfully verified the usefulness and flexibility of our methodology. ROS nodes for sensors and actuators provided the required abstraction from the perspective of the controller model implemented in the PowerDEVS toolkit. The transparent portability of ROS and PowerDEVS between a PC platform and a Raspberry Pi embedded target was a key feature.

The modifications introduced into the PowerDEVS engine abide by the standards of a DEVS real-time abstract simulator, assimilating smoothly the message-based communication with ROS nodes via UDP sockets. This opens up the possibility of reusing previous efforts in DEVS-based automatic control studies and adapt them smoothly to realistic scenarios working with real robots (or physical plants in general).

Next steps include a thorough characterization of real-time performance limits imposed by latencies introduced by DoveR. We will also investigate alternative mechanisms of communication between PowerDEVS and ROS other than network sockets. Additionally, we will leverage the recording capabilities of ROS in order to provide immediate feedback from the robot to the PC (enabling for systematic optimization of the controller parameters). Finally, we will develop a position tracking control for the robot.

REFERENCES

Bergero, F., and E. Kofman. 2010. "PowerDEVS: A Tool for Hybrid System Modeling and Real-Time Simulation". *Simulation* 87(1-2):113–132.

Bonaventura, M., D. Foguelman, and R. Castro. 2016. "Discrete Event Modeling and Simulation-Driven Engineering for the ATLAS Data Acquisition Network". *Comp. in Science & Engineering* 18(3):70–83.

Branicky, M. S., V. S. Borkar, and S. K. Mitter. 1998. "A Unified Framework for Hybrid Control: Model and Optimal Control Theory". *IEEE Transactions on Automatic Control* 43(1):31–45.

Castro, R., E. Kofman, and G. Wainer. 2009. "A DEVS–based End-to-end Methodology for Hybrid Control of Embedded Networking Systems". *IFAC Proceedings Volumes* 42(17):74–79.

Castro, R., I. Ramello, M. Bonaventura, and G. A. Wainer. 2012. "M&S-Based Design of Embedded Controllers on Network Processors". In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation – DEVS Integrative M&S Symposium*. March 26th-30th, Orlando, FL, USA, 32:1–32:8.

Cellier, F. E., and E. Kofman. 2006. *Continuous System Simulation*. New York: Springer Science & Business Media.

Cicirelli, F., L. Nigro, and P. F. Sciammarella. 2018. "Model Continuity in Cyber-Physical Systems: A Control-Centered Methodology Based on Agents". *Simulation Modelling Practice and Theory* 83:93–107.

Crick, C., G. Jay, S. Osentoski, B. Pitzer, and O. C. Jenkins. 2017. "Rosbridge: ROS for Non-ROS users". In *Robotics Research*, edited by H. I. Christensen and O. Khatib, 493–504. Cham, Switzerland: Springer.

Hu, X., and B. P. Zeigler. 2004. "Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example". *Journal of Intelligent and Robotic Systems* 39(1):71–87.

Marwedel, P. 2018. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 3 ed. Cham, Switzerland: Springer.

Moncada, M., E. Kofman, F. Bergero, and L. Gentili. 2013. "Generación Automática de Código para Sistemas Embebidos con PowerDEVS". In *Proceedings of the XV Workshop on Information Processing and Control (RPIC)*. September 16th-20th, Bariloche, Argentina. [in Spanish].

Niyonkuru, D., and G. A. Wainer. 2015. "Discrete-Event Modeling and Simulation for Embedded Systems". *Computing in Science & Engineering* 17(5):52–63.

Pecker Marcosig, E., J. I. Giribet, and R. Castro. 2017. "Hybrid Adaptive Control for UAV Data Collection: A Simulation-based Design to Trade-off Resources Between Stability and Communication". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan et al., 1704–1715. Piscataway, NJ, USA: IEEE.

Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. 2009. "ROS: An Open-Source Robot Operating System". In *Proceedings of the Open-Source Software Workshop of the International Conference on Robotics and Automation (ICRA)*. May 12th-17th, Kobe, Japan, 1-6.

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

Yu, H., and G. A. Wainer. 2007. "eCD++: An Engine for Executing DEVS Models in Embedded Platforms". In *Proceedings of the 2007 Summer Computer Simulation Conference*, 323–330. Piscataway, NJ, USA: IEEE.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2 ed. San Diego, CA, USA: Academic press.

AUTHOR BIOGRAPHY

EZEQUIEL PECKER MARCOSIG is an Electronics Engineer and a PhD student in the Facultad de Ingeniería, Universidad de Buenos Aires and ICC-CONICET. His work is supported with a PhD Fellowship from the Peruilh Foundation. His email address is epecker@fi.uba.ar.

JUAN I. GIRIBET is a Professor at the Universidad of Buenos Aires, and director of the Master degree in Mathematical Engineering. He is also a researcher at CONICET. His research interests are operator theory and its applications to control theory and signal processing. His email address is jgiribet@fi.uba.ar.

RODRIGO CASTRO is a Professor in the Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, head of the Simulation Lab and researcher at ICC-CONICET. His research includes simulation and control of hybrid systems. His email address is rcastro@dc.uba.ar.