Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

BRIDGING SIMULATION FORMALISMS AND EMBEDDED TARGETS: A POWERDEVS-DRIVEN IOT/ROBOTICS WORKFLOW FOR ESP32

Ezequiel Pecker-Marcosig[1,3], Sebastián Bocaccio[2], and Rodrigo Castro[1,2]

[1]Instituto UBA-CONICET de Ciencias de la Computación, Buenos Aires, ARGENTINA
[2]Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, ARGENTINA
[3]Lab. de Automática y Robótica, Facultad de Ingeniería, Universidad de Buenos Aires, Buenos Aires, ARGENTINA

ABSTRACT

This work presents a methodology for developing embedded applications in Internet-of-Things (IoT) and robotic systems through Model and Simulation (M&S)-based design. We introduce adaptations to the PowerDEVS toolkit's abstract simulator to enable embedded execution on resource-constrained platforms, specifically targeting the widely used ESP32 development kit tailored to IoT systems. We present a library of DEVS atomic models designed for simulation-environment interaction, enabling embedded software development through sensor data acquisition and actuator control. To demonstrate the practical utility of the embedded PowerDEVS framework, we evaluate its performance in real-world discrete-event control applications, including a line-follower robot and an electric kettle temperature regulator. These case studies highlight the approach's versatility and seamless integration in IoT and robotic systems.

1 INTRODUCTION AND RELATED WORKS

The continuous upsurge of embedded hardware platforms in the market at affordable prices and the pressure of users for new and innovative applications demand modern and efficient development cycles.

The design of embedded software for cyber-physical applications, which implies the interaction with the surrounding environment, is usually a complex task. Due to their complexity, different systematic and incremental design methodologies have been proposed with the aim of shortening development cycles. Modeling-and-Simulation (M&S)-based design techniques have proven a successful approach for the development of software applications, including embedded software. However, M&S is mostly used only during the early stages of the design process to build and test the software under development (SUD), while models are ultimately discarded when switching into the real target environment, and replaced by a new software implementation from scratch for the target hardware.

In contrast, innovative design methodologies based on the *model continuity* approach (Hu and Zeigler 2005) provide an incremental M&S-based design methodology. The process starts with conventional simulation of the models for the SUD and the environment. Ultimately the simulated environment becomes replaced by the real environment using some Hardware Abstraction Layer (HAL). Therefore, the model for the software remains the same throughout the different design stages, including its embedded execution in real-time (strictly speaking, an embedded real-time simulation). Model continuity has been successfully applied in a wide range of examples for the design of embedded applications for unmanned aerial vehicles (Horner et al. 2022; Ruiz-Martin et al. 2019), mobile robots (Hu and Zeigler 2005; Moallemi and Wainer 2013; Pecker-Marcosig et al. 2018), data network controllers (Wainer and Castro 2011), and power management systems (Cicirelli et al. 2018), to name a few.

Implementing the model continuity approach necessitates a unified framework capable of representing both the software being developed and the dynamic environment with which it interacts. The model for

the latter depends on the application and may be split into the model for the hardware platform (e.g. sensor aspects -such as uncertainty and periodical acquisition- , communication aspects -such as random delays or dropped messages- , physical aspects -such as robots' kinematics- , etc.) and the model of the environment itself. Anyway, this model usually ranges from simple abstractions of the real environment (e.g. pure discrete models) to complex detailed hybrid representations (e.g. involving non-linear differential equations). The Discrete-EVents System specification (DEVS) formalism can act as a common ground to represent and combine a wide variety of mathematical formalisms (Vangheluwe 2000), making it suitable to represent hybrid dynamical systems. DEVS also features some convenient advantages (Wainer and Castro 2011): (a) *reliability* due to logical and timing correctness (based on the DEVS theoretical roots and sound mathematical theory); (b) *model reuse* due to the modular and hierarchical structure of DEVS models fostering model composition; (c) *process flexibility* since a user does not need to worry about the simulation engine common to all models, and (d) *testing* since different test scenarios can be easily incorporated. Furthermore, the execution of an embedded simulation to interact with the environment in real-time claims for lightweight simulations, another aspect where DEVS stands out.

Different formal methodologies for the development of embedded systems making use of DEVS and model continuity have been proposed, such as Robot-in-the-Loop (RiL) (Hu and Zeigler 2005), Discrete-Event Modeling of Embedded Systems (DEMES) (Wainer and Castro 2011) or DEVS-over-ROS (DoveR) (Pecker-Marcosig et al. 2018) (the list is not exhaustive). They all have in common a *pure virtual simulation stage*, where the model of the SUD is designed, supplied with artificial data, and the simulation is executed in virtual simulation time, and a *real-time embedded simulation stage*, where the model for the embedded software is executed on the embedded system and the models representing the environment are replaced by interfaces with the real world. Nonetheless, they differ in the number of intermediate stages used to bring simulation one step closer to reality and consequently refine the design of the SUD, increasing the designers' confidence in the functioning of the final software. It is worth mentioning that the model continuity approach is not exclusive of DEVS, as shown in Cicirelli et al. (2018).

This paper is organized as follows. Section 2 provides the background. Section 3 presents the changes introduced to the PowerDEVS engine, the adaptations addressed in some atomic models to run on an ESP32 development kit and the toolchain to cross-compile models to run embedded. Section 4 introduces the library of DEVS atomic models developed to interact with the environment to be used in robotic and IoT applications, while Section 5 makes use of this library in the development of embedded software for some real applications. Finally, Section 6 concludes the paper.

2 PRELIMINARY CONCEPTS

2.1 Discrete Events Systems specification (DEVS) formalism

We adopted DEVS (Zeigler et al. 2018) for the modeling and simulation of hybrid dynamical systems. A remarkable aspect of DEVS is its modular and hierarchical structure, based on structural (coupled) and behavioral (atomic) models, which promotes the incremental construction of complex models, the robust reuse and replacement of modules, and a hierarchical building of systems. This fact fosters the definition of model libraries.

Formally, a DEVS *Atomic* model is a minimal building block defined as $M_A = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where X is the set of input events, Y is the set of output events and S is the set of state values. Four dynamic functions define the behavior of the model: δ_{int} (internal transition, for autonomous behavior), δ_{ext} (external transition, for receiving events and reactive behavior), λ (output, for emitting events) and ta (time advance, for autonomous self clocking). Each possible state $s \in S$ has an associated *time advance* $ta(s)$ that determines for how long the model will remain in s in the absence of external input events.

A DEVS *Coupled* model is a network of models defined as $M_C = \langle X_{self}, Y_{self}, D, \{M_d\}, \{I_d\}, \{Z_{d,j}\}, Select \rangle$ where $self$ is the coupled model itself, X_{self} and Y_{self} are the sets of input and output values of the coupled structure (respectively), D is the dictionary of connected components belonging to $self$, and M_d ($d \in D$)

is any other model, coupled or atomic. For each $d \in D \cup \{self\}$, I_d is the set of influencees models of subsystem d. For each $j \in I_d$, $Z_{d,j} : Y_d \to X_j$ is the d to j translation function, while $Select : 2^D \to D$ is a tie–breaking function for simultaneous events.

To graphically visualize the behavior of a DEVS atomic model we rely on the DEVS-Graph notation (Song and Kim 1994). The *states* $s \in S$ are represented by circular nodes including the state identifier and the time advance $ta(s)$, and the double circle is the initial state. Dashed edges between states denote internal transitions, labeled with the associated output ports and output events separated by an exclamation mark (`OutPort!OutValue`), while continuous edges represent external transitions, labeled with the input port that fires this transition and the event value separated by a question mark (`InPort?InValue`). Although there is a way to represent coupled models in DEVS-Graph, we will exploit the graphical representation provided by the PowerDEVS simulator (see next section).

A key aspect of DEVS is the strict separation between a DEVS model and the simulation engine used to run the model. Since the engine is common for any model, any modification introduced to it will have an impact on all new and previously developed models without any additional effort.

DEVS features a generalized abstract simulator algorithm, with two main classes of objects: *simulators* and *coordinators*. The execution of each atomic model is controlled by its *simulator*, while *coordinators* manage the coordination of coupled models and are in charge of synchronizing their children simulators and coordinators. At the top of this hierarchy, an overseeing *root-coordinator* is in charge of triggering new simulation cycles and handling the advancement of the global simulation time.

2.2 PowerDEVS simulation toolkit

The PowerDEVS toolkit (Bergero and Kofman 2011) is a DEVS simulator particularly suitable for the simulation of hybrid dynamical systems. PowerDEVS features a Graphical User Interface (GUI), which hides away the internal aspects of DEVS models facilitating its use by modelers unfamiliar with DEVS. A model is then built by interconnecting graphical representations of DEVS atomic and coupled models (blocks with input/output ports that are dragged, dropped and wired) including the setting of their parameters.

The core of PowerDEVS is written in C++. The building mechanism is based on a series of Makefiles for compiling the atomic models, the engine, the utilities and some PowerDEVS extensions, and for linking all together to create a standalone executable that comprise the model and simulator to run the simulation.

Beyond the discrete-events internally generated by a PowerDEVS model, the abstract simulator was adapted to receive external events from outside PowerDEVS (Pecker-Marcosig et al. 2018). Every time an atomic model requests for listening a given communication port, a *listener* is launched in a secondary thread which is continuously monitoring that port. Incoming events arriving at any time are enqueued in a message queue common to all listeners. During the idle time, the *root-coordinator* checks the queue and processes incoming messages by mapping them as external events for the listening atomics. A similar mechanism is available for an atomic model to send events outside PowerDEVS. Currently listeners listen to UDP sockets, while other IoT communication protocols are being incorporated.

2.3 Embedded Hardware Development Kits for Internet-of-Things (IoT) Applications

Several embedded kits for the development of IoT applications can be found in the market. DEVS simulation toolkits have already been tested on the most popular hardware platforms nowadays for embedded applications, such as STM32 Nucleo (Ruiz-Martin et al. 2019; Earle et al. 2020), ESP32 (Govind and Wainer 2024) and RED-V Things Plus (Cárdenas et al. 2024). Furthermore, DEVS simulators are also used to run simulations in real-time on PCs and single board computers (SBC) on top of general purpose operating systems (OS), as in Pecker-Marcosig et al. (2018), Horner et al. (2022), Govind and Wainer (2024). One distinctive feature of PowerDEVS is its ability to build controllers by visually wiring graphical blocks in a modular and hierarchical manner. This is a highly valued feature of established simulation toolkits in control engineering.

In general, the execution of embedded applications on an embedded platform relies on an underlying real-time operating system (RTOS) which, among other things provides device drivers, a precise time management and a Hardware Abstraction Layer (HAL). Consequently, the same application on top of an RTOS can be run on different hardware platforms without any change.

The ESP32-Wroom-32 development kit by Espressif stands out for IoT applications due to its versatility (thanks to its multiple peripherals and communication interfaces), availability, and performance at an affordable price. Moreover, its high processing power in a small footprint and lightweight platform makes it a good choice for autonomous vehicles. The ESP32 is a dual-core Xtensa 32-bit LX6 system-on-chip (SoC), with built-in Wi-Fi and Bluetooth communication, and multiple I/O and peripheral ports. The two identical cores are connected to a single shared main memory, allowing data exchange between tasks running on different cores. A custom version of FreeRTOS is mantained by Espressif specifically targeted to the ESP32 development kit to exploit its features (ESP-IDF FreeRTOS).

An application for the ESP32 is developed on a PC and cross-compiled for the Xtensa architecture. The executable is then copied to the onboard flash memory via a UART port accessible through a serial-to-USB converter mounted onboard. The same UART is connected to the standard output for logging purposes.

3 ADAPTATIONS PERFORMED TO POWERDEVS

This section covers the adaptations performed to PowerDEVS and the modified toolchain to build and run embedded simulations over FreeRTOS on an ESP32 development kit. A priori the adaptations should only be necessary on the engine and they will have an impact on all new and previously developed models without any additional effort. However, some atomic models make use of particular features that are not useful nor compatible for embedded simulations and need to be removed or adapted.

3.1 Adaptations of the PowerDEVS Simulation Engine for Embedded Execution

The PowerDEVS simulation toolkit was originally released in 2011 (Bergero and Kofman 2011), and across more than a decade many features were developed in the engine to support different research lines. For the embedded version of PowerDEVS, we only keep the features that find a potential use for embedded applications, while removing all the unnecessary features.

Any M&S-based design methodology calls for a single PowerDEVS version to be used to run simulations on different platforms (PC or embedded) and in different time settings (virtual simulation time or real wall-clock time). Therefore, to keep the PowerDEVS model (`pdm` file created with the GUI) exactly the same (or as untouched as possible) we make use of conditional compilation directives to indicate the corresponding toolchains the code to build for different platforms, for example, removing unnecessary features and replacing some pieces of code.

The PowerDEVS engine was adapted to run in the user application layer on top of ESP-IDF FreeRTOS. Thanks to the FreeRTOS' HAL, PowerDEVS would also run on any of the large numbers of supported boards. Moreover, FreeRTOS provides functions for the management of time simplifying the timely execution of discrete-events by the abstract simulator in real-time simulations, and a large set of device drivers that will be extensively used throughout this paper. Furthermore, C++ libraries offer another source of abstraction providing a common interface despite having a different implementation for each OS (e.g. for the management of time and UDP sockets). However, some C++ libraries are not fully ported to FreeRTOS, such as the Boost C++ set of libraries.

To foster adaptability and modularity, functions used by the PowerDEVS engine that depend on specific features of the underlying hardware platform and/or OS are separated from the rest. This is why we have the file `pdevslib.linux.cpp` for simulations on Linux and `pdevslib.esp32.cpp` for embedded simulations on ESP32. For example, the function `printLog()` used to manually print log messages for debugging writes in a text file in the Linux version, while on the embedded version it prints to standard output. This file is also used to launch UDP listeners requested by atomic models to listen to UDP ports.

In Linux, the listeners run as separate threads independent from the main thread (see Section 2.2), while on the embedded version listeners run as independent FreeRTOS tasks running on the second core. This way, we decided to use one core for the execution of PowerDEVS handled by its task scheduler, while the second core handled by another task scheduler is left to execute the listener tasks. Finally, some functions available in the Linux version are not yet implemented in the embedded version.

While waiting for the next imminent event in real-time simulations, PowerDEVS performs a *busy waiting* instead of putting the embedded device in sleep mode as in other embedded DEVS toolkits. This is particularly relevant in embedded systems with stand-alone power supply. However, during this period, the *root-coordinator* monitors the arrival of UDP messages carrying external events generated outside PowerDEVS. Alternative mechanisms based on hardware interrupts (IRQs) and independent cores sleeping management will be explored as future work.

3.2 Adaptations of PowerDEVS Atomic Models for Embedded Execution

The model continuity approach, central to M&S-based design methodologies for software development, requires some atomic models to run on both PC and embedded platforms. For example, general purpose atomic models, such as those for signals generation and UDP communication.

Most of the atomic models in PowerDEVS use *parameter readers* to abstract multiple sources of model parameters, and *simulation loggers* for various destinations of simulation results. However, they rely heavily on C++ libraries not fully supported by FreeRTOS. We used conditional compilation directives to remove them from the code to run embedded. Currently, the only way to supply model parameters in embedded PowerDEVS is via the model file (`pdm`), and the only mechanism for debugging and logging is the `printLog` function (see Section 3.1).

3.3 Build Mechanism and Simulation Execution

The typical workflow with PowerDEVS is as follows. A PowerDEVS model is developed with the GUI, which consists of a `pdm` text file containing the list of the atomic and coupled models, the model parameters, and structural and graphical data. Then it is passed to the PowerDEVS's *Pre-processor* which translates the `pdm` file into a C++ header file (`model.h`). Finally, the file `model.cpp` (common to all models) and the `model.h` header file for the current PowerDEVS model are built and linked with the particular atomic models used, the engine and the needed utils, to create the simulation standalone executable file `model`. This workflow changes slightly when working with embedded applications. Espressif provides its own framework for building, flashing and monitoring embedded applications for ESP32 called *Espressif IoT Development Framework* or ESP-IDF for short. The whole process is handled by a Python script called `idf.py` provided by Espressif. Given the growing community of users of the ESP32 development platform, there is a lot of code available online. Espressif itself maintains a server with application components that can be freely downloaded and used out-of-the-box in our own applications (see Section 4.3).

For the embedded software running on the ESP32, the entry point for a user's application is the function `app_main()` which is automatically invoked on startup. This function is then used to create a FreeRTOS task called `create_model()`, which calls the `main()` function of the `model` executable and passes the simulation parameters.

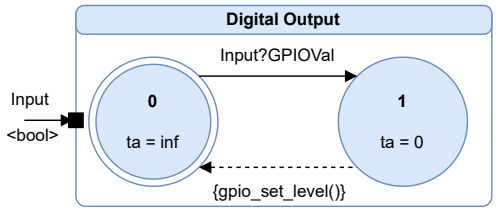
The ESP-IDF toolchain uses `CMake`, so all folders and files to compile –including PowerDEVS' atomics, engine, and utilities– are listed in a `CMakeLists.txt` file. This file also excludes folders and files not compiled for running embedded (e.g., those with unsupported libraries).

4 LIBRARY OF EMBEDDED MODELS TO INTERFACE INPUT/OUTPUT DEVICES

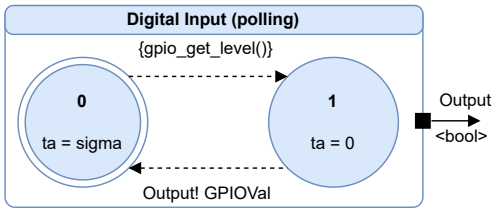
This section presents a PowerDEVS library of atomic models named `esp32` aimed at developing embedded applications to monitor and interact with the environment. The ESP32 board provides a variety of peripherals ready to use ranging from general purpose input/output ports (GPIO) to LCD and SD card interfaces, covering

several serial communication protocols including UART, SPI, 1-Wire and I2C. The most popular hardware available off-the-shelf for data acquisition and actuation in IoT and robotic applications can be interfaced using GPIOs and serial communication ports.

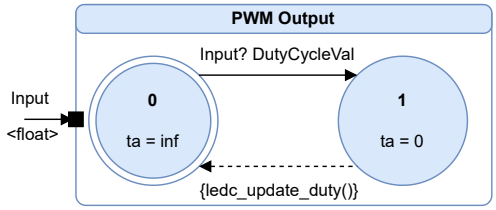
This section presents the atomic models to interface general purpose inputs (Figure 1(b)) and outputs (Figure 1(a)), including handlers for multiple digital inputs and outputs, pulse-width modulation (PWM) outputs (Figure 1(c)) and a temperature sensor through an I2C communication port (Figure 1(d)). These models can be combined to create interfaces for more complex hardware, such as H-bridges (Section 5.3) and stepper motor drives.



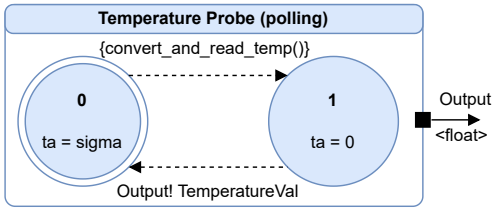
(a) Digital Output Atomic Model.



(b) Digital Input Atomic Model.



(c) PWM Output Atomic Model.



(d) Temperature Probe Atomic Model.

Figure 1: Description of the functioning of the atomic models in the `esp32` library for data acquisition and actuation using DEVS-Graph notation. Curly braces enclose actions executed during state transitions.

4.1 Digital Input/Output atomic models

The *Digital Input* atomic model periodically polls the state of a given GPIO of the ESP32 board. During the initialization, the GPIO port is set as input using the ESP32 GPIO API. This atomic model periodically schedules an internal transition to read the value in the input port (with the API function `gpio_get_level()`), and fires an output event with this value through the atomic's output port `Output`. Figure 1(b) illustrates this behavior in DEVS-Graph notation. The input pin and the polling rate are supplied as parameters of the atomic model.

The *Digital Input Handler* atomic model is used in connection with multiple *Digital Input* atomic models to produce a single output event after having new readings from all the digital inputs it is attached to. The number of inputs is passed as a parameter.

The *Digital Output* atomic model is used to write the value of an input event on a GPIO of the ESP32. During the initialization, the GPIO port is set as output and initialized in low level using the ESP32 GPIO API. Every time an input event is received in the atomic's input port `Input`, an internal transition is triggered in zero time and the value carried by the input event is written in the GPIO (with the API function `gpio_set_level()`) during the output function λ. This behavior is shown in Figure 1(a). The pin is supplied as parameter of the atomic model.

In case N values need to be written simultaneously in N GPIOs, we should use N *Digital Output* atomic models which will schedule N internal transitions at the exact same time instant. Although the events are scheduled at the same time, given that PowerDEVS features DEVS Classic, they will be executed sequentially one after the other. Consequently, there will be an inevitable drift (measured in wall-clock

time) due to the overhead needed by the *root-coordinator* to determine the imminent atomic model, make the time advance (in this case, zero time) and write the value arriving to each *Digital Output* on the corresponding GPIO. This drift might lead to the malfunctioning of the hardware. The simplest way to solve this issue is to use a single atomic model that receives an input event carrying the *N* values to write on all the GPIOs during the same transition function. This atomic model is named *Multiple Digital Outputs* and accepts parameters to set the number of outputs *N* and the corresponding GPIO pins.

4.2 PWM Output atomic model

Typical actuators for the interaction with the environment in embedded applications are small DC motors and stepper motors. The speed of a DC motor is proportional to the supply voltage and it is typically managed with PWM signals characterized by a frequency (or period) and a duty cycle.

The *PWM Digital Output* atomic model makes use of the LEDC peripheral of the ESP32 to generate PWM signals. During the initialization, the LEDC peripheral is configured using the ESP32 LEDC API, setting the PWM frequency, the resolution for the duty cycle (13 bits) and its initial value is set to zero. The duty cycle must be between 0 and 100 % which correspond to 0 and $(2^{\text{resolution}} - 1)$.

Every time this atomic model receives an input event in port `input` carrying the value for the duty cycle, an internal transition is scheduled in zero time to update the corresponding value for the LEDC peripheral (with the API function `ledc_update_duty()`) during the output function λ. Figure 1(c) illustrates this behavior. This atomic model has a single parameter for the GPIO pin. It is responsibility of the user to check that this GPIO actually features a PWM.

4.3 Temperature Probe atomic model

Some common input devices to acquire data from the environment rely on serial communication rather than on parallel interfaces through multiple GPIOs. These interfaces are usually found on devices that need to exchange complex dataframes. This is the case of the DS18B20 temperature probe, a widespread off-the-shelf 1-Wire digital thermometer. Given the extensive use of this device in embedded applications with the ESP32, we rely on third-party drivers for the DS18B20 and the 1-Wire communication based on the Remote Control Transceiver (RMT) peripheral. For the 1-Wire communication, the ESP32 plays the role of master, while the DS18B20 device acts as slave.

The DS18B20 is usually in idle state and every time a new temperature measurement is queried, the master must issue a convert command. Once the measurement is ready, it is stored in the 2-byte temperature register and the DS18B20 returns to idle state. The device indicates that a new measurement is available and then the master issues a read command (with the driver function `convert_and_read_temp()`).

We developed the *DS18B20 Probe* atomic model to interact with this device. During the initialization, the RMT peripheral is set and the device is set to use the maximum resolution to take 12-bit measurements using the ESP32 RMT API. This atomic model works similarly to the *Digital Input* atomic model by polling the GPIO pin where the probe is attached to (Figure 1(d)). This pin is supplied as a parameter.

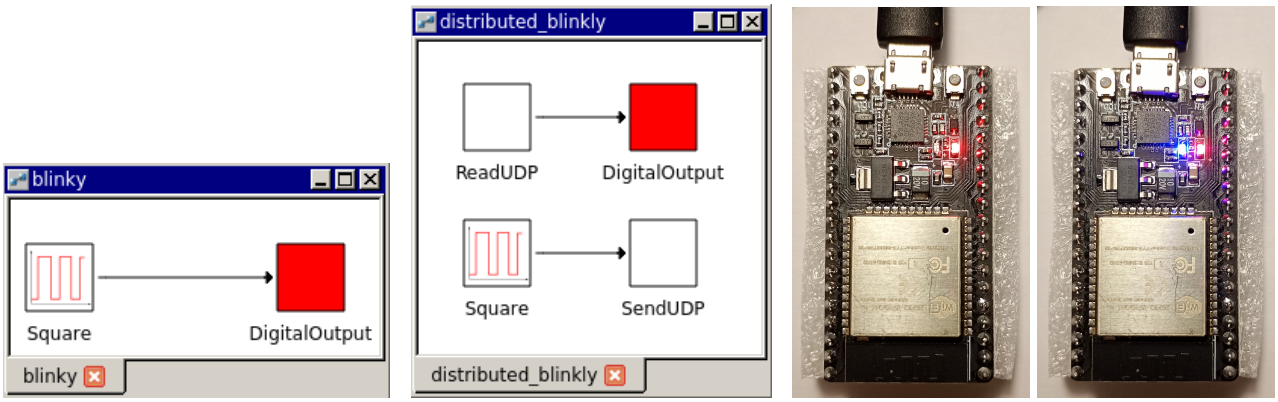
5 CASE STUDIES

This section covers some case studies using M&S to develop embedded applications with the `esp32` library introduced in Section 4. The design methodology starts with a *pure virtual simulation stage* on PC to verify correct functioning via simulation logs. Then, the same model is deployed and tested in hardware during the *real-time embedded simulation stage*, replacing models representing the environment by interfaces to the real hardware. The first example is a *Blinky LED* application, where the onboard LED of the ESP32 microcontroller blinks at a configurable frequency –serving as a canonical `Hello world!` for embedded systems. This example is further extended to show how to use the `udpcomm` library alongside with the `esp32` to communicate simulations running concurrently on different platforms (embedded and PC) via UDP sockets. The next two examples illustrate M&S-based design of discrete-event controllers

for: (a) a temperature controller and (b) a line-follower robot. All the examples are available in a git repository SEDLab (2025a), including the controller for a stepper motor that is not shown here due to space restrictions. All these embedded applications (strictly speaking, PowerDEVS models) were tested in real-time on an ESP32-Wroom-32 development kit with 38 pins.

5.1 Blinky LED and Distributed Blinky LED

This model is built by simply combining a *Square* atomic model from the `sources` library, which periodically generates output events alternating values 0 and 1 at a frequency set by the user, and a *Digital Output*, which receives those events and writes their values on the corresponding GPIO pin. Figure 2(a) illustrates the *Blinky LED* coupled model. In this example, we make use of the ESP32 onboard LED connected to the GPIO pin 2 (see Figure 2(d)), therefore the parameter of the *Digital Output* was set to 2. An excerpt of the simulation logging is shown in Listing 1.



(a) Stand-alone Blinky LED model. (b) Distributed Blinky LED model. (c) LED Off. (d) LED On.

Figure 2: Snapshots of PowerDEVS' GUI for the Blinky LED examples: (a) Stand-alone Blinky LED, (b) Distributed Blinky LED. ESP32 Onboard LED: (c) Off, (d) On.

The Blinky LED example was extended to test inter-simulation communication, where two (or more) independent executions of PowerDEVS running on different platforms exchange data in runtime. In this case, there are two *Blinky LED* coupled models running in two different ESP32 boards. The *Square* atomic model in one simulation periodically generates events for the *Digital Output* atomic model in the other simulation using the *Send UDP* and *Read UDP* atomic models in-between, and vice versa. *Send UDP* and *Read UDP* atomic models belong to the `udpcomm` library and use the UDP communication mechanism described in Section 2.2. Figure 2(b) shows a snapshot of this model. Both boards must be connected to the same WiFi network. An excerpt of the simulation logging is shown in Listing 2 (Device 1, listens to UDP port: 6000 and sends to IP: 192.168.234.94 and UDP port: 5000) and Listing 3 (Device 2, listens to UDP port: 5000 and sends to IP: 192.168.234.37 and UDP port: 6000). However, this example is not limited to embedded simulations, and it can also be tested to communicate two simulations running on an ESP32 board and a PC. In this last case, the *Digital Output* atomic model should be replaced by a *GNU Plot* atomic model to plot the incoming events.

Listing 1: Simulation logging from the Blinky example.

```
[DigitalOutput] Simu Time: 1693.00, Dext function
[DigitalOutput] Simu Time: 1693.00, Output function, Write Value: 0 in GPIO pin: 2
[DigitalOutput] Simu Time: 1693.00, Dint function
[DigitalOutput] Simu Time: 1694.00, Dext function
[DigitalOutput] Simu Time: 1694.00, Output function, Write Value: 1 in GPIO pin: 2
[DigitalOutput] Simu Time: 1694.00, Dint function
```

Listing 2: Simulation logging from Device 1.

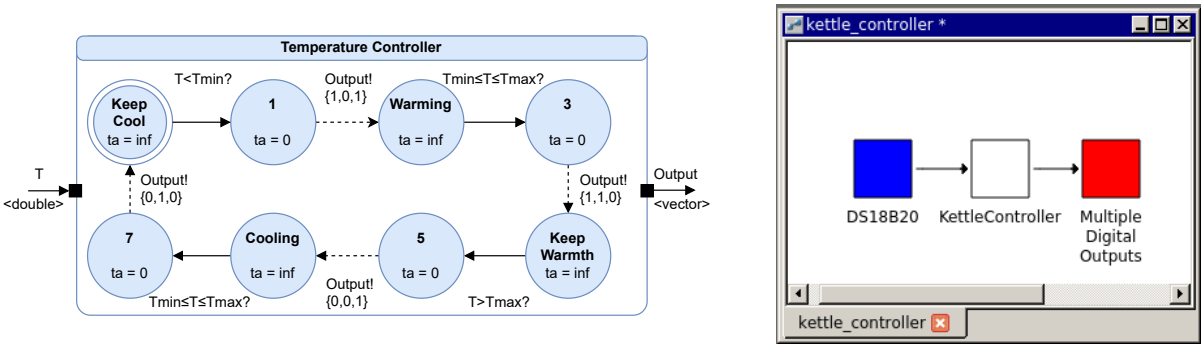
```
[ReadUDP] Requesting UDP port: 6000
[SendUDP] Init UDP:5000, IP:192.168.234.94
[SendUDP] Simu Time: 15.00, Msg:0, IP
    ↪ :192.168.234.94, UDP:5000
[sendNET] Sent data: 0 to IP:192.168.234.94,
    ↪ UDP:5000
[netHandler] Received: 0 on UDP:6000
[ReadUDP] Simu Time: 15.21, Rcv Msg: 0
[SendUDP] Simu Time: 16.00, Msg: 1, IP
    ↪ :192.168.234.94, UDP:5000
[sendNET] Sent data: 1 to IP:192.168.234.94,
    ↪ UDP:5000
[netHandler] Received: 1 on UDP:6000
[ReadUDP] Simu Time: 16.38 Rcv Msg: 1
```

Listing 3: Simulation logging from Device 2.

```
[ReadUDP] Requesting UDP port: 5000
[SendUDP] Init UDP:6000, IP:192.168.234.37
[SendUDP] Simu Time: 15.00, Msg: 0, IP:
    ↪ 192.168.234.37, UDP: 6000
[sendNET] Sent data: 0 to IP: 192.168.234.37,
    ↪ UDP: 6000
[netHandler] Received:0 on UDP:5000
[ReadUDP] Simu Time: 15.15, Rcv Msg: 0
[SendUDP] Simu Time: 16.00, Msg: 1, IP:
    ↪ 192.168.234.37, UDP: 6000
[sendNET] Sent data: 1 to IP: 192.168.234.37,
    ↪ UDP: 6000
[netHandler] Received: 1 on UDP: 5000
[ReadUDP] Simu Time: 16.28, Rcv Msg: 1
```

5.2 Design of a Temperature Controller

This section covers the design of a closed-loop discrete-event controller to maintain a heating system's temperature within predefined high and low thresholds. Such on-off controllers with hysteresis are common in HVAC and level control systems, where a variable of interest must remain between set limits.



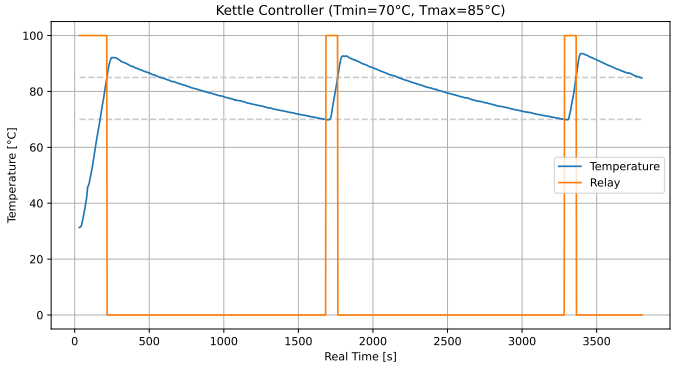
(a) Temperature Controller atomic model.

(b) Kettle Controller coupled model.

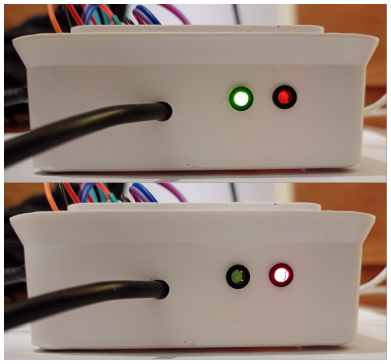
Figure 3: Temperature controller: (a) DEVS-Graph for the *Temperature Controller* atomic model, (b) Snapshot of the PowerDEVS' GUI for the *Kettle Controller* coupled model.

The temperature controller in this section will be tested on an electric kettle. This controller is implemented as an atomic model named *Kettle Controller*. To close the loop, we need measurements of the temperature from the system under control. To interface with a DS18B20 probe installed on the electric kettle, we make use of the *DS18B20 Probe* atomic model. Upon receiving an input event carrying the current temperature, the *Kettle Controller* checks if it is inside the desired range set by the user and triggers an output event with the controller action for a heating device (actuator). The heating device in this case is a relay to turn on/off the electric kettle. In addition, a visual indication is provided by two additional outputs of the controller connected to *Digital Output* atomics, which write on two GPIO pins wired to a green LED (temperature is in-range) and a red LED (temperature is out-of-range), see Figure 4(b).

The *Kettle Controller* atomic works as follows: if the temperature T is below the lower threshold T_{min} the heating device and the red LED are turned on; if the temperature T is above the upper threshold T_{max} the heating device is turned off and the red LED is also turned on; if the temperature T is in range but rising the heating device and the green LED are turned on; while if the temperature is in range but falling the heating device is turned off and the green LED is turned on. This behavior is shown in Figure 3(a) using DEVS-Graph notation. To simplify this notation, we assume that: (a) the temperature value received



(a) Logged data.



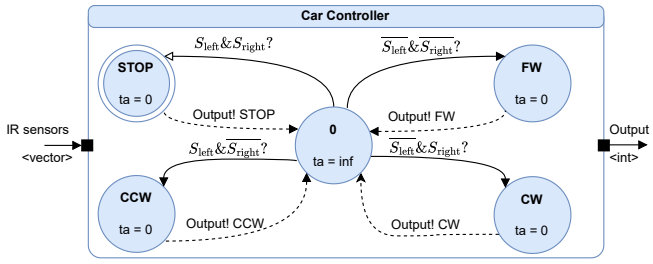
(b) LEDs.

Figure 4: Embedded execution of the *Kettle Controller* (with $T_{\min} = 70\,^{\circ}\text{C}$ and $T_{\max} = 85\,^{\circ}\text{C}$): (a) Measured temperature and actuated relay, (b) green LED (T is in-range) and red LED (T is out-of-range).

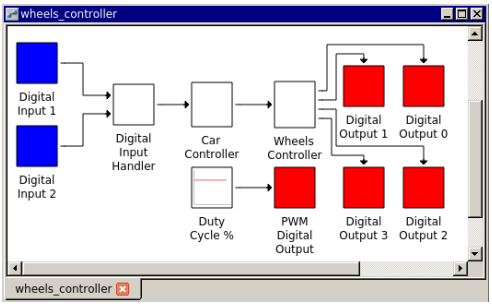
by the controller is pre-processed to generate a discrete event every time one of the following conditions is met: $T < T_{\min}$, $T > T_{\max}$ and $T_{\min} \leq T \leq T_{\max}$, and (b) the value of an output event is a 3-tuple `<heating device, green led, red led>`. The *Kettle Controller* takes $T_{\min}$ and $T_{\max}$ as parameters. The *Kettle Controller* coupled model is shown in Figure 3(b), which corresponds to a snapshot of the PowerDEVS' GUI. Figure 4(a) shows the logged data from the embedded execution of *Kettle Controller*.

5.3 Design of a Line-follower Robot Controller

This section covers the design of a closed-loop discrete-event controller to make a 2-Wheel Drive (2WD) robot follow a black line in the ground. This application is commonly known as line-follower robot.



(a) Car Controller atomic model.



(b) Line-Follower Controller coupled model.

Figure 5: Line-follower robot controller: (a) DEVS-Graph for the *Car Controller* atomic model, (b) Snapshot of the PowerDEVS' GUI for the *Line-Follower Controller* coupled model.

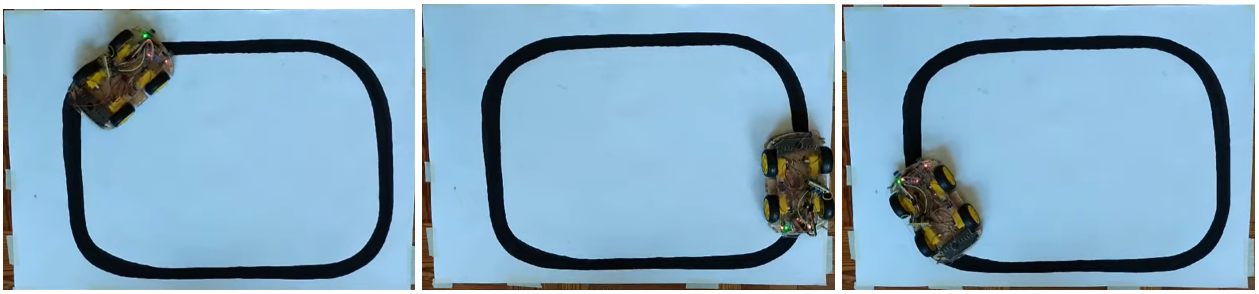
In this case the control loop is closed with measurements of the relative position of the 2WD robot with respect to the dark line in the ground. To this end we use two off-the-shelf TCRT5000 infrared (IR) sensors, where an IR LED emits infrared light and a photosensitive device receives the reflected light in an object placed in front. Both sensors are placed beneath the robot facing ground. The object to be detected is the black line contrasting with a white background. If the sensor is over the line there is no light reflected and its output will be a high level (1), while it will be a low level (0) if it is over the white background.

The discrete-event controller for the line-follower robot works as follows. If the two IR sensors are placed over the white background with the black line in-between the robot will move forward. If the right sensor is over the black line and the left sensor is over the white background the robot will turn clockwise,

while in the opposite case it will turn counter-clockwise. Finally, if both sensors are on the black line the robot will stop. The *Car Controller* atomic model implements this behavior (see Figure 5(a)).

The two wheels of the 2WD robot are managed with two independent small DC motors attached to the ESP32 through an off-the-shelf double H-bridge driver (L298n). Using the drive/coast operation mode, this driver is managed by three digital inputs per motor, one for setting the speed (with a PWM signal) and two for setting the direction of rotation. To make the 2WD robot move forward, both wheels must rotate clockwise with same speed. To make the robot rotate clockwise, the left wheel has to rotate clockwise while the right wheel has to remain stopped. To make the robot rotate counter-clockwise, it is the other way around. The *Wheels Controller* atomic model features a look-up table which receives the action to be applied to the 2WD robot (in {`FW`,`CW`,`CCW`,`STOP`}) and generates the values for the four digital output pins connected to the input pins of the L298n driver to set the wheels' direction of rotation.

Figure 5(b) shows the coupled model for this application. Two *Digital Input* atomic models are used to read the IR sensors, a *Digital Input Handler* atomic model is used to generate a single output event after the arrival of new measurements in both IR sensors, the *Car Controller* atomic model receives the measurements of the IR sensors and decides how the 2WD robot has to move, then the *Wheels Controller* atomic model translates the controller action to rotation commands for the wheels, and four *Digital Output* atomic models are used to write on the inputs of the motors driver. There is also a *PWM Output* atomic model to generate a PWM signal for the motors driver to set a fixed speed on both motors. Figure 6 shows some snapshots of the behavior of the line-follower robot in closed-loop, taken from SEDLab (2025b).



(a) Time = 01:20:00 (b) Time = 01:30:00 (c) Time = 01:38:00

Figure 6: Snapshots of the Line-Follower Robot in closed-loop with the embedded controller model simulated in real-time. Timestamps in min:sec:millisec.

6 CONCLUDING REMARKS AND FUTURE WORK

This paper presented the adaptations addressed on the PowerDEVS simulation toolkit to run embedded simulations on top of the FreeRTOS real-time operating system on an ESP32 development kit. It also introduced the `esp32` library of atomic models for the development of embedded applications targeting robotics and IoT applications. The availability of atomic models for interacting out-of-the-box with the environment promotes the rapid development of embedded applications based on M&S.

A series of simple but representative case studies of increasing complexity have been developed using this library, covering applications in both areas. The applications presented in this paper showed that the adaptations were effective and allowed the execution of the same DEVS model in both PCs and embedded environments and in both virtual and real-time.

Next steps include the development of more complex real-world-sized applications which will require the growth of the `esp32` library with atomic models to interface with typical hardware found in robotics and IoT applications. Moreover, the models need to be posed under exhaustive testing to ensure the robustness and reliability of DEVS models running in real-time on embedded hardware.

Available ESP32 sleep modes need to be analyzed to replace the use of busy waiting by the *root-coordinator* to wait for the next imminent event in real-time simulations compatible to the monitoring of the arrival of external messages to PowerDEVS. Moreover, specific *parameter readers* and *simulation loggers* targeted to embedded applications, complementary to those available for Linux simulations, need to be developed.

REFERENCES

Bergero, F., and E. Kofman. 2011. "PowerDEVS: a Tool for Hybrid System Modeling and Real-Time Simulation". *Simulation* 87(1-2):113–132.

Cárdenas, R., P. Malagón, P. Arroba, and J. L. Risco-Martín. 2024. "xDEVS no_std: A Rust Crate for Real-Time DEVS on Embedded Systems". In *Proc. of the 2024 Annual Modeling and Simulation Conference (ANNSIM)*. May 20th-23th, Washington, DC, USA, 1-13.

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.

Earle, B., K. Bjornson, C. Ruiz-Martin, and G. Wainer. 2020. "Development of A Real-Time DEVS Kernel: RT-Cadmium". In *Proc. of the 2020 Spring Simulation Conference (SpringSim)*. May 19th-21th, Fairfax, VA, USA, 1-12.

Govind, S., and G. Wainer. 2024. "Handling Asynchronous Inputs in DEVS Based Real-Time Kernels". In *2024 Winter Simulation Conference (WSC)*, 2277–2288 https://doi.org/10.1109/WSC63780.2024.10838981.

Horner, J., T. Trautrim, C. R. Martin, G. Wainer, and I. Borshchova. 2022. "Discrete- Event Supervisory Control for the Landing Phase of a Helicopter Flight". In *2022 Winter Simulation Conference (WSC)*, 441–452 https://doi.org/10.1109/WSC57314.2022.10015393.

Hu, X., and B. Zeigler. 2005. "Model Continuity in the Design of Dynamic Distributed Real-Time Systems". *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 35(6):867–878.

Moallemi, M., and G. Wainer. 2013. "Modeling and Simulation-driven Development of Embedded Real-Time Systems". *Simulation Modelling Practice and Theory* 38:115–131.

Pecker-Marcosig, E., J. I. Giribet, and R. Castro. 2018. "DEVS-Over-ROS (DOVER): A Framework for Simulation-Driven Embedded Control of Robotic Systems Based on Model Continuity". In *2018 Winter Simulation Conference (WSC)*, 1250–1261 https://doi.org/10.1109/WSC.2018.8632504.

Ruiz-Martin, C., A. Al-Habashna, G. Wainer, and L. Belloli. 2019. "Control of a Quadcopter Application with DEVS". In *Proc. of the 2019 Spring Simulation Conference (SpringSim)*. April 29th-May 2nd, Tucson, AZ, USA, 1-12.

SEDLab 2025a. "Embedded PowerDEVS Examples". https://git-modsimu.exp.dc.uba.ar/matiasb/powerdevs-CERN/-/tree/create-esp32-engine/examples/ESP32, accessed 11th April 2025.

SEDLab 2025b. "Embedded simulation of a Line Follower Controller implemented in PowerDEVS". https://youtu.be/Ka0sNb7t5UA, accessed 9th April 2025.

Song, H. S., and T. G. Kim. 1994. "The DEVS Framework for Discrete Event Systems Control". In *Proc. of the Fifth Annual Conference on AI, and Planning in High Autonomy Systems*. Dec 7th-9th, Gainesville, FL, USA, 228-234.

Vangheluwe, H. L. 2000. "DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling". In *Proc. of the 2000 IEEE International Symposium on Computer-Aided Control System Design (CACSD)*. Dec 25th-27th, Anchorage, AK, USA, 129-134.

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.

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. 3rd ed. San Diego, CA, USA: Academic Press.

AUTHOR BIOGRAPHIES

EZEQUIEL PECKER-MARCOSIG has been appointed Assistant Researcher in CONICET (National Research Council of Argentina) and holds an Adjunct Professor position at UBA. His email address is emarcosig@dc.uba.ar.

SEBASTIÁN BOCACCIO is an MASc student in Computer Science in the Computing Department, Faculty of Exact and Natural Sciences, UBA, and scholar at the Discrete Event Simulation Lab. His e-mail address is sbocaccio@dc.uba.ar.

RODRIGO CASTRO is Professor with the UBA and Head of the Discrete Event Simulation Lab at the Research Institute of Computer Science (ICC) of CONICET (National Research Council of Argentina). His e-mail address is rcastro@dc.uba.ar.