Facilitating the use of Physics-Based Simulations on Embedded Devices by running FMUs from MicroPython

Tom REYNAUD[1] Erfan ENFERAD[2] Maxime LEFRANCOIS[3]

[1]Mines Saint-Étienne, France, tom.reynaud@etu.emse.fr
[2]erfan.enferad@gmail.com
[3]Mines Saint-Étienne, Univ Clermont Auvergne, INP Clermont Auvergne, CNRS, UMR 6158 LIMOS, F - 42023, Saint-Étienne, France, maxime.lefrancois@emse.fr

Abstract

Physics-based simulations (PBS) are increasingly valuable for real-time applications in embedded systems, yet integrating them on resource-constrained devices remains challenging. This paper presents ufmu, a lightweight framework that enables execution of source-code FMI 2.0-compliant Functional Mock-up Units (FMUs) within the MicroPython environment, targeting platforms such as the ESP32. The proposed approach integrates into the MicroPython firmware the FMU source code and the FMU model description translated into C structures, and exposes a minimal Python API for simulation control, enabling model-based computations on-device without cloud dependencies. We evaluate the framework using a standard FMU model, comparing performance across ESP32, Unix, and plain C environments in terms of memory usage, execution time, and firmware size. Despite the ESP32's hardware limitations, the results demonstrate that meaningful simulations can be achieved efficiently, with minimal memory overhead. All code, documentation, and experiment instructions are freely available under an MIT license, supporting reproducibility and adoption in education, prototyping, and embedded research. This work also lays the foundation for future integration with eFMI and the FMI 3.0 standard.

Keywords: Physics-Based Simulation, FMI, FMU, eFMI, MicroPython

1 Introduction

The simulation software market is expected to reach 40.5 billion US dollars in 2030 according to a report (Zoting 2023). In recent years physics-based simulation (PBS) has become a very important tool in different domains of both science and industry. PBS support for the analysis and prediction of events, with applications ranging from breaking systems in vehicles (Magargle et al. 2017) to earthquake prediction (Saad et al. 2023), having the potential to save lives.

Embedded devices support physics-based simulation (PBS) by supplying data. In a cloud–edge architecture, they typically collect sensor data, such as temperature or humidity, and transmit it to the cloud for processing and storage. Still, performing real-time calculations in the cloud is costly, as it increases network traffic and latency, lowers energy efficiency, and can hinder scalability (Savaglio et al. 2019). Recent developments enable embedded devices to run simulations themselves, helping to address these issues while also simplifying the overall architecture (Lenord et al. 2021). However, the skill gap between simulation development and embedded system programming remains a barrier to broader adoption and integration of these capabilities, especially for hobbyists or DIY (Do-It-Yourself) enthusiasts.

The Functional Mockup Interface (FMI), developed by the Modelica Association, is an open standard for sharing and co-simulating dynamic models, supported by tools for monitoring, evaluation, and plotting (Blockwitz et al. 2012). Models are packaged as Functional Mock-up Units (FMUs), which encapsulate code, data, and metadata in a standardized format for easy integration across tools.

Developed as part of the EMPHYSIS project, the embedded Functional Mock-up Interface (eFMI) extends FMI to enable simulation execution on embedded devices (Lenord et al. 2021). eFMI introduces a dedicated programming language, GALEC, with the goal to ensure PBS execution stays within the resource constraints of embedded systems. GALEC guarantees static worst-case execution time and memory usage, and prevents illegal memory access. Additionally, eFMI's container architecture supports multiple model representations, including algorithmic, production, and binary code. These features make eFMI suitable for safety-critical applications and adaptable to various embedded software environments.

Optimized for microcontrollers, MicroPython (George and MicroPython community 2014) is a streamlined implementation of Python 3 designed to compile and execute code directly on microcontrollers, which are at the heart of most embedded devices. MicroPython is easier to learn and use than C (Fangohr 2004) and provides a rich set of libraries, making it well-suited for education and rapid prototyping in embedded systems (Tollervey 2017). To our knowledge, embedding FMI simulations directly within the MicroPython environment is unprecedented and complements the eFMI paradigm.

Therefore, this paper addresses the following research

question: *How can FMUs be integrated into MicroPython for execution on embedded devices ?* We propose the ufmu framework, which enables execution of FMI 2.0-compliant FMUs within MicroPython by:

1. translating the FMU model description into a C structure,
2. integrating it and the FMU source code into the MicroPython build, and
3. exposing the FMU via a lightweight Python API through the ufmu library, allowing simulations to run directly within MicroPython.

This paper makes the following key contributions:

- A framework for exporting and running one source-code FMI 2.0-compliant FMU embedded inside the MicroPython firmware;
- Performance evaluation of the proposed integration;
- Exploration of future perspectives, including the potential use of eFMI for further optimizations and adaptation to the FMI 3.0 standard.

The rest of this paper is structured as follows. Section 2 provides background information. Section 3 outlines the ufmu framework, detailing our methodology, associated challenges, and optimizations. It also validates the implementation through test cases, including running the bouncing ball simulation on an ESP32 microcontroller. Section 4 discusses related work. Finally, Section 5 concludes the paper by summarizing the contributions, discussing the implications of the work, and outlining future directions for research and development.

2 Background

2.1 The need for PBS in embedded devices

PBS are models that represent the physical behavior of a system. Applications include predictive maintenance, where a component's remaining life is estimated based on its current condition to enable timely intervention before failure (Magargle et al. 2017), and fault diagnosis, where the source of faults is identified from observed behavior to support corrective actions and improve system reliability (Aivaliotis et al. 2019).

The use of PBS directly inside embedded systems is an emerging field that holds great promise for improving the accuracy, efficiency, and performances of these systems (Tavella et al. 2016). Accuracy is improved by offering a more precise representation of the system's physical behavior, which is crucial for real-time operations in safety-critical systems, medical devices, and automobiles (Armugham et al. 2021). Efficiency is improved by enabling auto-calibrating scenarios, reducing the need for extensive testing and debugging (Liu and Negrut 2021; Gundermann et al. 2017). Performance is enhanced by optimizing the architecture and operation of the system, leading to a more efficient and responsive system (Vanommeslaeghe et al. 2018; Tavella et al. 2016). Additionally, performing the process on the embedded device itself boosts responsiveness and scalability.

However, several challenges remain. First, PBS can be computationally demanding, restricting their deployment on resource-constrained devices. Second, integration with existing embedded software can be complex and may require significant development effort and expertise. Finally, their sensitivity to input data quality makes them difficult to apply in real-world scenarios (Duraiswami and Zotkin 2016). Despite these challenges, the benefits of integrating physics-based simulations into embedded systems are evident. As technology advances, their adoption is likely to grow across a wide range of applications.

2.2 FMI and FMUs

The Functional Mockup Interface (FMI), developed by the Modelica Association, is an open standard for sharing and co-simulating Modelica models, which are typically exported as Functional Mock-up Units (FMUs). FMUs encapsulate code, data, and metadata in a uniform package, enhancing the credibility and usability of simulations in complex engineering processes (Gall et al. 2021).

FMI 2.0 is the most widely adopted and extensively supported version of FMI, with over 210 tools offering compatibility and ensuring seamless integration across industries such as automotive, aerospace, and energy.[1] While FMI 3.0 introduces advanced co-simulation features, richer data-type support, and scheduled-execution interfaces—such as 8-, 16-, 32-, and 64-bit integer types, single-precision floats, and opaque binary variables—its adoption remains nascent compared to 2.0. Given its proven stability, broad toolchain compatibility, and full backward-compatibility with existing FMUs, FMI 2.0 is the preferred choice for most engineering workflows today, and the one we will focus on in this paper.

The following outlines the sequence of FMI function calls that ensure accurate state propagation, event handling, and system updates within the simulation. All FMU function calls omit the `fmi2_` prefix for better readability. During initialization, the FMU is created with `instantiate`, and memory is allocated for state variables and event indicators. The experiment is configured using `setupExperiment`, followed by a transition into initialization mode with `enterInitializationMode`. Initial variable values are retrieved using `getReal` and `getInteger`. After exiting initialization mode with `exitInitializationMode`, an initial event iteration is performed using `newDiscreteStates`. The simulation loop starts by retrieving continuous states and derivatives via `getContinuousStates` and `getDerivatives`. Time advancement is handled with `setTime`, and the next simulation step is performed using `setContinuousStates`. The event handling mechanism is activated when necessary, transitioning into event mode with `enterEventMode` and resolving

[1] `https://fmi-standard.org/tools/`

DOI 10.3384/ecp218535

discrete state updates via `newDiscreteStates`. If required, the simulation re-enters continuous-time mode using `enterContinuousTimeMode`. Throughout the process, output variables are gathered by invoking `getReal` and `getInteger`.

2.3 MicroPython

MicroPython (George and MicroPython community 2014) is a lightweight implementation of Python designed for microcontrollers. Unlike CPython, which requires a full OS, MicroPython runs directly on bare-metal hardware, providing direct access to peripherals like GPIO, I2C, SPI, and UART, making it ideal for resource-constrained environments. Arduino has embraced MicroPython as a supported platform for programming microcontrollers, providing tools like the Arduino Lab for MicroPython to facilitate development. This integration allows developers to leverage the simplicity of Python while utilizing Arduino's extensive hardware ecosystem.

MicroPython leverages one of Python's key advantages: its ease of use, especially in comparison to C/C++, the dominant languages for embedded systems programming. According to (Fangohr 2004), Python-based development can significantly reduce the time needed for implementation and debugging.

Additionally, as discussed in (Tollervey 2017), MicroPython includes an extensive standard library and supports external modules, allowing developers to leverage a broad range of functionalities without needing to write low-level firmware from scratch. It also has built-in libraries to harness as much as its support can offer, like the for the ESP32, as shown in Figure 1

Despite its compact size, MicroPython remains highly extensible. Developers can integrate it with C/C++ code for performance-critical tasks, ensuring that time-sensitive operations are not compromised. Furthermore, the active community and robust documentation provide support, making MicroPython an attractive option for both hobbyists and professionals.

```
from machine import Pin
from onewire import OneWire
from ds18x20 import DS18X20
# Initialize DS18B20 sensor on GPIO pin 4
ds = DS18X20(OneWire(Pin(4)))
# Scan for connected devices
roms = ds.scan()
# Start temperature conversion
ds.convert_temp()
print("Temperature:", ds.read_temp(roms[0]),
↪  "°C")
```

Figure 1. Snippet of the MicroPython code to read temperature on an embedded device from an external sensor

3 The ufmu framework

3.1 Overview

Our approach integrates an FMU following the FMI 2.0 standard into a MicroPython-compatible C module, enabling users to interact with FMUs directly from MicroPython scripts. Figure 2 illustrates the process, which involves: (A) Exporting the simulation as FMU, (B) Decompressing the FMU, (C) Parsing and modifying the .c, .h, .xml files, (D) Building MicroPython with the ufmu library, (E) Flashing the device with the firmware.

The following subsections detail the process. Section 3.2 covers the extraction and preparation of the FMU archive for integration into the embedded environment. Section 3.3 discusses the process of adapting the FMU to work seamlessly with MicroPython, including wrapping simulation functions for compatibility. Several challenges had to be addressed, such as adapting FMU execution for a resource-constrained embedded environment, handling compilation differences between Unix and ESP32, and ensuring modularity and reusability of the functions. Section 3.4 discusses how these issues were tackled and the optimizations applied. Finally, Section 3.5 reports on the evaluation of the ufmu framework.

ufmu supports both the Unix and the ESP32 ports of MicroPython. All the code and necessary instructions to reproduce the experiments are openly available on Github under the open-source license of the MIT,[2] alongside the end-user documentation.[3]

3.2 FMU Preprocessing

The first step of the framework involves exporting the simulation as an FMU (Step A). FMUs are zip archives typically containing C source files, an XML model description file, and precompiled binaries for various platforms, as can be seen in States 2 and 3 of Figure 2. The framework then consists in preprocessing the FMU.

First, the FMU file is unzipped (Step B) using a Makefile, revealing its internal structure, including the `modelDescription.xml` file and source code (State 3). Then, the `modelDescription.xml` file is parsed to extract metadata about the FMU, such as variable definitions, simulation parameters, and function mappings (Step C). This parsing is performed using the `xmllint` software to dynamically configure the library.

Once the relevant information is extracted, the source files are modified to integrate with the MicroPython-compatible library. This includes wrapping core FMU functions to expose them as callable functions from MicroPython, adjusting variable storage and memory handling to fit the microcontroller's limitations, and adding required definitions to handle different prefixes for function names. Additionally, configuration headers are generated based on the extracted metadata, storing variable

[2] https://github.com/Imaginus02/MicroFMU-library
[3] https://imaginus02.github.io/MicroFMU-library

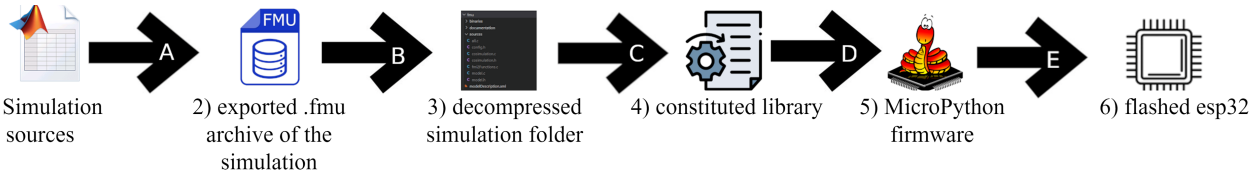

Figure 2. Flow diagram of the MicroPython firmware creation process, showing the integration of the FMU execution engine, preprocessing, and compilation steps.

definitions and simulation parameters efficiently (State 4).

This preprocessing ensures that each FMU is correctly configured before compilation, allowing it to run on the target environment with no manual intervention (either Unix or ESP32 in our case).

3.3 Library Integration into MicroPython

Once the FMU source code has been preprocessed, the next step involves integrating it into the MicroPython firmware (Step D), and flashing it on the device (Step E).

Integration is achieved using a C module that bridges the FMU execution engine and the MicroPython interpreter. This module exposes the core FMU functions via the minimal MicroPython-compatible ufmu API. As detailed in Annex A, the API includes a main class and utility functions to initialize a simulation instance with specified time parameters, access variable names and descriptions, read and modify variable values, and iterate through the simulation instance with output at each step.

The build process then differs by target: building for Unix uses a traditional Makefile workflow, while building for the ESP32 relies on CMake and the ESP-IDF framework, with different flags for function name overrides and linker configuration. Once the FMU execution engine is shipped into the MicroPython firmware, the end-developer only needs to define how the simulation is controlled and executed using this API.

Figure 3 illustrates the use of the ufmu API to control and execute the **BouncingBall.fmu** simulation. A simulation instance is created by calling the `setup_simulation` function with a start time, a stop time, and a step size. Once initialized, simulation variables can be accessed using `get_variable_names`, which returns an array containing the names of all FMU variables, and `get_variables_description`, which provides additional details on each variable's purpose and properties. The simulation instance is implemented as a Python *generator*, allowing users to step through it with a simple `for` loop. At each iteration, it yields an array of current variable values in a consistent order. Users can dynamically modify some of the variables during execution using `change_variable_value`, in line with the FMI 2.0 specification. This design ensures both simplicity and flexibility when interacting with the simulation.

```
>>> import ufmu as fmu
>>>
>>> # Print available variables and their
 ↪  descriptions
>>> for name in fmu.get_variables_names():
>>>     description =
 ↪  fmu.get_variables_description(name)
>>>     base_value =
 ↪  fmu.get_variables_base_values(name)
>>>     print(f"'{name}' (= {base_value} at
 ↪  t0): {description}")
'step' (= 0 at t0): Simulation step count
'time' (= 0.0 at t0): Simulation time
'h' (= 1.0 at t0): Position of the ball
'der(h)' (= 0.0 at t0): Derivative of h
'v' (= 0.0 at t0): Velocity of the ball
'der(v)' (= 0.0 at t0): Derivative of v
'g' (= -9.81 at t0): Gravity acting on the ball
'e' (= 0.7 at t0): Coefficient of restitution
>>>
>>> # Set up the simulation parameters
>>> startTime, stopTime, stepSize = 0, 3, 0.1
>>>
>>> # Initialize simulation instance
>>> sim = fmu.setup_simulation(startTime,
 ↪  stopTime, stepSize)
>>>
>>> # Change a variable (e.g., 'h' for height)
>>> fmu.change_variable_value(sim, 'h', 20)
>>>
>>> # Run the simulation and print time and
 ↪  height for each step
>>> for s, t, h, der_h, v, der_v, g, e in sim:
>>>     print(f"t: {t:.2f}s, h: {h:.2f}m")
t: 0.10s, h: 19.95m
t: 0.20s, h: 19.80m
...
```

Figure 3. REPL (Read-Evaluate-Print-Loop) snippet of MicroPython code to retrieve simulation information and run the **BouncingBall.fmu** simulation.

3.4 Challenges and Optimizations

DOI 10.3384/ecp218535

Developing the ufmu framework to run FMUs from MicroPython on an ESP32 presented several challenges that required careful consideration and optimization to ensure the library's functionality and efficiency. Since microcontrollers in embedded systems are typically dedicated to a single task, the approach adopted was to embed a single FMU within the MicroPython environment and expose it through the fmu module.

One of the main challenges was the discrepancy between the compilation processes for Unix and ESP32 targets. The Unix port relies on Makefiles, while the ESP32 port uses CMake and the ESP-IDF framework. This difference necessitated adjustments in build configurations to ensure compatibility across both platforms. Specifically, the ESP32 compilation process required additional flags to override certain function name prefixes in the FMU source code, which were incompatible with the linker's expectations. By dynamically adjusting the build flags, we ensured proper linking of functions during compilation, resolving this issue effectively.

Memory management was another critical challenge, given the ESP32's limited RAM (approximately 4MB). To optimize memory usage, the library reuses the same variable to store the results of each simulation step. Instead of retaining historical data, each new step overwrites the previous values, ensuring minimal memory consumption. This approach allows the library to handle simulations with a large number of variables and steps without exceeding the microcontroller's memory limits. The end-developer decides whether to retain the values or not.

Function name prefix conflicts also posed a significant challenge. The FMU source code often included prefixes that conflicted with the linker's expectations during the ESP32 compilation process. To address this, the library dynamically modifies the function names during preprocessing, ensuring they align with the linker's requirements. This adjustment was crucial for enabling successful compilation and execution on the ESP32.

3.5 Evaluation

While the Unix port served as a reliable testing ground, the ESP32 port required additional debugging due to its unique constraints. The bouncing ball simulation, provided as a FMU test case, was instrumental in identifying and resolving issues related to memory management, function naming, and execution speed.

We assessed the ufmu framework by measuring the impact of shipping the **BouncingBall.fmu** on the firmware size, and measuring the execution time and peak memory usage for running this model from 0 to 10 with two distinct configurations: one with a step size of 0.01 and another with a step size of 0.0001.

Two ports were compared to a plain C simulator[4] developed alongside the ufmu library: MicroPython for the ESP32 target, and MicroPython for Unix. To ensure the reliability and accuracy of our performance measurements, each simulation was executed 10 times under identical conditions, with the same code as shown in Figure 4, and the mean value of the results was calculated. This approach minimizes the impact of any outliers or transient system behaviors that could skew the data. The Plain C simulator and the MicroPython Unix port were both executed within a WSL2 environment on a Windows 10 system, leveraging the computational power of an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. The MicroPython ESP32 implementation, on the other hand, was run directly on the embedded hardware to evaluate its performance in a resource-constrained environment.

The results are summarized in Table 1. Integrating ufmu with the **BouncingBall.fmu** model increased the ESP32 MicroPython firmware by 220 KB (15 %), bumping it from 1.5 MB to roughly 1.72 MB. The Plain C simulator outperformed both Unix and MicroPython implementations in terms of execution speed (respectively 2 and 100 times faster). However, its memory usage, because of the fact that it keep all step result, is hardly comparable to the MicroPython one. The executable size is minimal, as it avoids the overhead associated with interpreted languages like MicroPython.

The MicroPython Unix port exhibited a bug when handling simulations with more than 5,000 steps, preventing it from completing the larger simulation (step size = 0.0001).[5] This issue is reflected in the results with an "X" in Table 1. Despite this limitation, the Unix port demonstrated competitive performance for smaller simulations.

The MicroPython ESP32 port, while significantly slower than the other two due to the hardware limitations of the ESP32, successfully completed both simulations. These results highlight the trade-offs between performance and resource efficiency when deploying simulations on embedded systems.

```
import ufmu as fmu
import time
import micropython
start_time = 0
end_time = 10
step_size = XX
start = time.ticks_ms()
simInstance = fmu.setup_simulation(start_time,
↪  end_time, step_size)
fmu.change_variable_value(simInstance, 'h', 20)
for i in simInstance:
        pass
print("Time taken: ",
↪  time.ticks_diff(time.ticks_ms(), start),
↪  "ms")
micropython.mem_info()
```

Figure 4. The code run by MicroPython to get the result presented in Table 1. The value replaced here by XX is respectively 0.01 and 0.00001

[4] https://github.com/Imaginus02/FMUSimulator

[5] This corresponds to a known issue: https://github.com/Imaginus02/MicroFMU-library/issues/1

4 Related Work

4.1 Portable runtime environments for Python-based FMUs

The concept of portability extends beyond just MicroPython. In the field of FMUs, ensuring that Python-based FMUs can execute across various systems without dependency conflicts is crucial. One approach to achieving this portability is through containerization.

A notable effort in this direction is the addition of Docker support to UniFMU (Schranz et al. 2021), a tool designed for building and executing FMUs. By encapsulating all necessary runtime dependencies in a Docker container, this approach allows FMUs to be distributed and deployed seamlessly across different environments. This capability is particularly beneficial in scenarios requiring co-simulation, as demonstrated in the case of a robotic arm and its controller, where two Python-based FMUs were successfully executed within a Docker container. This method eliminates compatibility issues and simplifies the deployment process, aligning with the broader goal of making Python-based embedded applications more accessible and portable.

By combining the lightweight nature of MicroPython with advances in FMU portability, developers can leverage Python for a wider range of embedded and simulation applications. The ability to run MicroPython on constrained hardware while simultaneously supporting Python-based FMUs in containerized environments highlights the versatility and growing influence of Python in embedded systems and simulation workflows.

This approach becomes particularly valuable when deploying large-scale embedded systems for co-simulation scenarios, such as in (Jung, Shah, and Weyrich 2018), where a dynamic, multi-agent-based simulation framework was used to model a smart warehouse. In this scenario, various IoT components—such as storage racks, temperature sensors, forklifts, and incoming goods—communicate and make autonomous decisions in real time. Each component is represented as an independent simulation agent capable of dynamically joining an ongoing co-simulation, enabling a flexible "Plug-and-Simulate" behavior. By leveraging modular co-simulation techniques, the study demonstrates how distributed embedded systems can interact seamlessly, allowing for efficient and adaptive digital twin implementations. The ability to deploy Python-based FMUs within portable runtime environments, such as Docker, further enhances this flexibility, enabling scalable and reproducible simulations across heterogeneous IoT ecosystems.

4.2 FMI and eFMI in Embedded, Automotive, and Industrial Applications

Recent advancements have demonstrated the feasibility of using FMUs and their embedded counterpart, eFMU, for real-time simulation on microcontrollers. The increasing computational capabilities of embedded hardware, along with the standardization of the FMI, have unlocked new possibilities for Cyber-Physical Systems (CPS), industrial automation, and automotive systems.

In the context of CPS, (Hong et al. 2020) proposed a simulation framework leveraging FMI, incorporating a redundancy reduction algorithm to optimize performance for resource-constrained microcontrollers. Similarly, (Gundermann et al. 2017) validated embedded FMI for lifetime testing of electromechanical systems through a 60-day real-time study on a linear stepper motor, confirming its reliability in embedded environments.

The eFMI standard, developed under the EMPHYSIS project and detailed in (Lenord et al. 2021), is tailored to meet the constraints of embedded systems, including hard real-time requirements and limited memory. It enables seamless integration of high-fidelity, physics-based models into embedded control systems, making it particularly impactful in domains like automotive, aerospace, and industrial automation, where such models can enhance control strategies, diagnostics, and predictive maintenance.

In the automotive sector, FMI plays a pivotal role in software validation, virtual Electronic Control Unit (ECU) testing, and real-time simulation of vehicle dynamics. (Mikelsons and Samlaus 2017) demonstrated the integration of FMI-based models into ECU validation workflows, identifying software issues early in the development process and improving reliability.

To overcome the incompatibility of traditional Co-Simulation FMUs with embedded automotive applications—due to stringent real-time constraints, MISRA compliance, and AUTOSAR architecture—eFMI has been extended to AUTOSAR environments (Armugham et al. 2021). In a case study, a physics-based model was integrated into a Bosch ECU using an AUTOSAR toolchain, illustrating a compliant, efficient, and cost-effective approach that significantly reduced the manual coding effort typically required.

Beyond automotive use cases, FMI and eFMI are also advancing industrial automation by enabling embedded simulations for real-time failure detection and predictive diagnostics. As embedded FMI technology evolves alongside edge computing and optimized firmware, its role in real-time embedded systems is expected to grow substantially, driving innovation across various high-demand sectors.

Building upon these advancements, our MicroPython FMU implementation bridges a gap in the embedded systems landscape by making simulation capabilities accessible to Python programmers in resource-constrained environments. While existing work has demonstrated FMU viability on dedicated microcontrollers, our approach extends these benefits to the growing ecosystem of MicroPython-enabled devices, enabling rapid prototyping, interactive debugging, and simplified deployment without sacrificing real-time performance. This contributes to the democratization of complex simulation

DOI
10.3384/ecp218535

Table 1. Executing the BouncingBall.fmu simulation in different Python environments with setting start_time = 0, end_time = 10, and step_size = 10^{-2} or 10^{-5}. h is changed to 20 at the start of the simulation. X means that the test resulted in an error.

Exec. environment	**Plain C simulator**		**MicroPython Unix**		**MicroPython ESP32**	
Executable size (KB)	112		843		1,725 (+220 / +15 %)	
Step size	10^{-2}	10^{-5}	10^{-2}	10^{-5}	10^{-2}	10^{-5}
Time (s)	0.001	1.005	0.002	X	0.139	141.309
Memory usage (KB)	1,632	64,032	355	X	21	46

capabilities, allowing domain experts without extensive embedded C programming knowledge to leverage FMI's standardized approach for applications ranging from educational settings to industrial IoT deployments where Python's flexibility and MicroPython's efficiency can significantly accelerate development cycles.

4.3 Cloud vs. Edge Deployment for FMI-Based Simulations

Several studies have investigated the trade-offs between cloud-based and edge-based IoT simulations. (Savaglio et al. 2019) assessed the performance, latency, scalability, and resource utilization of both deployment options. Their findings suggest that edge computing provides lower latency and better scalability for IoT applications, whereas cloud computing offers superior data handling and processing reliability. Recent advancements in edge deployment, such as those proposed by (Bu, Filipau, and Baklanov 2024), have further streamlined the process of deploying cyber-physical models at the edge or in the cloud. Their framework utilizes FMUs as executable binaries and JavaFMI as the simulation engine, encapsulating each model deployment within a microservice. This approach ensures flexibility and scalability, as demonstrated by their examples of a winch controller at the edge and a wireline logging unit simulator in the Azure DevOps pipeline.

For FMI-based simulations, the choice between cloud and edge computing depends on factors such as real-time constraints, network bandwidth, and security considerations. Many real-time embedded applications benefit from edge-based execution, as it allows FMUs to run with minimal latency directly on the device. The classification of hardware types, as outlined by (Bormann et al. 2025), provides further clarity on this decision. Table 2 summarizes the classification of common devices used in IoT simulations and deployments.

For instance, devices like the Raspberry Pi, classified as Class 15, are well-suited for fog computing, while microcontrollers such as the ESP32, classified as Class 10, are ideal for edge computing. This classification helps in aligning the hardware capabilities with the specific requirements of IoT simulations and deployments.

The work of (Jung, Shah, and Weyrich 2018) complements this discussion by proposing a dynamic co-simulation approach for IoT systems, where each component is simulated independently and can dynamically join the co-simulation during runtime. This "Plug-and-Simulate" behavior is particularly advantageous for edge deployments, where heterogeneous devices with varying computational capabilities must interact seamlessly. Their modular framework aligns with the microservice architecture proposed by (Bu, Filipau, and Baklanov 2024), highlighting the importance of flexibility in edge-based FMI deployments.

Table 2. Classification of devices for IoT simulations and deployments by (Bormann et al. 2025)

Device	**Classification**
Arduino Uno	Class 5 (Edge Computing)
ESP32	Class 10 (Edge Computing)
Raspberry Pi	Class 15 (Fog Computing)
Modern Smartphone	Class 16 (Edge/Cloud Computing)
Laptop	Class 17 (Cloud Computing)
High end Home Computer	Class 18 (Cloud Computing)

On the cloud side, (Stelzig and Rodenberg 2023) introduces the concept of Simulation Model as a Service (SMaaS), which leverages cloud infrastructure to deploy, execute, and track simulation models efficiently. While cloud-based solutions excel in handling large-scale simulations and complex data processing, they often face challenges related to latency and real-time responsiveness. This trade-off underscores the need for hybrid approaches, where computationally intensive tasks are offloaded to the cloud, while time-critical operations are handled at the edge.

The integration of these approaches is further supported by (Vanommeslaeghe et al. 2018), which explores the co-simulation of embedded platforms and physics-based models. Their work emphasizes the importance of optimizing hardware architectures to support FMI-based simulations, particularly in edge environments where resource constraints are a significant concern. By combining insights from these studies, it becomes clear that the choice between cloud and edge deployment is not binary but rather a spectrum, where the optimal solution depends on the specific requirements of the application, such as latency tolerance, computational complexity, and scalability

needs.

Building on these deployment insights, our MicroPython FMU implementation presents a good middle ground particularly well-suited for Class 10 devices like the ESP32. By embedding FMU capabilities directly into MicroPython, we enable sophisticated edge-based simulations without requiring cloud connectivity or fog computing resources. This approach addresses the latency concerns highlighted in edge computing research while maintaining the flexibility needed for dynamic co-simulation scenarios. The memory-optimized design of our module responds directly to the resource constraints emphasized in the literature, while its Python interface removes barriers to implementation that typically accompany embedded FMU deployments. This positions our work as a practical enhancement to the "Plug-and-Simulate" paradigm, extending FMI's benefits to resource-constrained IoT devices where traditional deployment methods would be prohibitively complex or performance-intensive.

4.4 Discussion

By combining insights from these studies, it becomes clear that hardware optimization for FMI execution involves a multi-faceted approach, encompassing efficient memory management, real-time scheduling, and hardware-specific optimizations. These advancements are paving the way for more sophisticated FMI-based simulations on resource-constrained devices, enabling their use in a wider range of applications, from industrial automation to autonomous systems. The state of the art in FMI-based simulations for embedded systems demonstrates significant progress across multiple fronts, from portable runtime environments to hardware optimization. The integration of Python-based FMUs with containerization technologies, such as Docker, has enhanced portability and reproducibility, enabling seamless deployment across diverse environments. This is particularly valuable for co-simulation scenarios, where modular and dynamic frameworks, as proposed by (Jung, Shah, and Weyrich 2018), allow heterogeneous systems to interact efficiently.

Despite these advances, there remain significant limitations to the adoption of MicroPython-based FMU implementations in industrial settings. Professional environments typically demand rigorous certification, long-term support, and deterministic performance guarantees that interpreted languages like Python may struggle to provide. While our implementation offers valuable prototyping capabilities, industry practitioners will likely continue favoring native C/C++ implementations for production systems where reliability and performance predictability are paramount.

The development of eFMI has further expanded the applicability of FMI in resource-constrained environments, enabling high-fidelity physics-based models to run on microcontrollers. This advancement is particularly transformative for automotive and industrial applications, where real-time execution and compliance with standards like AUTOSAR are critical. The successful integration of eFMI into automotive ECUs, as demonstrated by (Armugham et al. 2021), highlights its potential to bridge the gap between simulation and real-world execution. eFMI being compatible with FMI, we expect that our framework could be used to run FMUs optimized for resource-constrained environments thanks to GALEC.

Our current implementation focuses exclusively on FMI 2.0 standard compatibility, which presents both advantages and limitations. While FMI 2.0 offers widespread tool support and established functionality, it lacks newer features from FMI 3.0 such as binary co-simulation interfaces and enhanced array variable handling that could potentially improve performance on constrained devices. Future work could explore adapting our approach to leverage these newer standard capabilities as tool support matures.

The debate between cloud and edge deployment for FMI-based simulations underscores the importance of balancing computational power, latency, and scalability. While cloud-based solutions excel in handling large-scale simulations, edge computing offers the low latency and real-time responsiveness required for many embedded applications. Hybrid approaches, combining the strengths of both paradigms, are emerging as a promising solution for complex, distributed systems. Our choice of the ESP32 platform was strategically motivated by its unique combination of computational capability, memory resources, and wide adoption in the IoT ecosystem. With dual-core processing up to 240MHz and typically 4MB of flash memory, it represents an optimal middle ground between severely constrained Class 1-5 devices and more powerful Class 15+ systems. Furthermore, the ESP32's native support for both WiFi and Bluetooth positions it ideally for co-simulation scenarios where edge devices must communicate with other simulation components.

As the field continues to evolve, the convergence of these technologies—portable runtime environments, eFMI, cloud-edge hybrid architectures—will play a pivotal role in addressing the challenges of real-time execution, resource constraints, and system integration.

5 Conclusions

This paper introduced ufmu, a lightweight library that enables FMU simulations to run efficiently on resource-constrained microcontrollers. By integrating and executing FMU models within the MicroPython environment, ufmu provides an accessible and flexible FMU tool that may prove useful for education and rapid prototyping in embedded systems.

The library achieves this by translating FMU model descriptions into a C-based structure and embedding them within the MicroPython build process. This approach ensures seamless execution while exposing a minimal high-level Python API for user interaction. As a result, ufmu allows developers to run model-based simulations directly

DOI
10.3384/ecp218535

on microcontrollers without relying on a full desktop or cloud computing setup.

While the current implementation successfully enables FMU simulations to run on an ESP32 microcontroller, there are several areas for future improvement to enhance the library's functionality, usability, and performance.

As a natural progression of this work, adding support for the FMI 3.0 standard is a key objective. The current implementation supports basic simulation execution but supporting advanced states such as setup, pause, and resume modes, would be desirable. For example, users could pause a simulation to modify parameters and then resume execution without restarting the simulation.

Future development will also focus on expanding support to other microcontroller platforms, such as STM32 or Raspberry Pi Pico.

References

Aivaliotis, Panagiotis. et al. (2019). "Methodology for enabling Digital Twin using advanced physics-based modelling in predictive maintenance". In: *Procedia CIRP* 81, pp. 417–422. ISSN: 22128271. DOI: 10.1016/j.procir.2019.03.072. (Visited on 2023-10-08).

Armugham, Siva Sankar et al. (2021). *eFMI (FMI for Embedded Systems) in AUTOSAR for Next Generation Automotive Software Development*. URL: https://www.sae.org/publications/technical-papers/content/2021-26-0048/ (visited on 2025-02-19).

Blockwitz, Torsten et al. (2012). "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models". en. In: *Functional Mockup Interface 2.0*. DOI: 10.3384/ecp12076173. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=76&Article_No=17.

Bormann, Carsten et al. (2025-01). *Terminology for Constrained-Node Networks*. Internet Draft draft-ietf-iotops-7228bis-01. Num Pages: 28. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/draft-ietf-iotops-7228bis (visited on 2025-02-20).

Bu, Fanping, Mikalai Filipau, and Nikolay Baklanov (2024). "Advanced Edge Deployment: Abstracting Cyber-Physical Models via FMU Mastery". en. In: *Modelica Conferences*, pp. 170–177. ISSN: 1650-3740. DOI: 10.3384/ecp207170. URL: https://www.ecp.ep.liu.se/index.php/modelica/article/view/1142 (visited on 2025-02-19).

Duraiswami, Ramani and Dmitry N. Zotkin (2016-10-01). "Efficient physics based simulation of spatial audio for virtual and augmented reality". In: *Journal of the Acoustical Society of America* 140.4. Number: 4_Supplement, pp. 2999–3000. ISSN: 0001-4966, 1520-8524. DOI: 10.1121/1.4969294. (Visited on 2023-10-08).

Fangohr, Hans (2004). "A comparison of C, MATLAB, and Python as teaching languages in engineering". In: *Computational Science-ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part IV 4*. Springer, pp. 1210–1217.

Gall, Leo et al. (2021-09). "Continuous Development and Management of Credible Modelica Models". en. In: *Modelica Conferences*, pp. 359–372. ISSN: 1650-3740. DOI: 10.3384/ecp21181359. URL: https://www.ecp.ep.liu.se/index.php/modelica/article/view/214 (visited on 2025-02-19).

George, Damien P. and the MicroPython community (2014). *MicroPython*. Ed. by George Robotics Limited. https://micropython.org/. Accessed: 2025-07-30.

Gundermann, Julia et al. (2017-07-04). "The Embedded Simulation via FMI and its Application to the Simulation of Lifetime Tests Including Wear". In: The 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, pp. 541–545. DOI: 10.3384/ecp17132541. (Visited on 2023-10-09).

Hong, Seokjoon et al. (2020-01-01). "F-DCS: FMI-Based Distributed CPS Simulation Framework with a Redundancy Reduction Algorithm". In: *Sensors* 20.1. Number: 1, p. 252. ISSN: 1424-8220. DOI: 10.3390/s20010252. (Visited on 2023-10-08).

Jung, Tobias, Payal Shah, and Michael Weyrich (2018). "Dynamic Co-Simulation of Internet-of-Things-Components using a Multi-Agent-System". In: *Procedia CIRP* 72, pp. 874–879. ISSN: 22128271. DOI: 10.1016/j.procir.2018.03.084. (Visited on 2023-10-08).

Lenord, Oliver et al. (2021-09-27). "eFMI: An open standard for physical models in embedded software". In: 14th Modelica Conference 2021, pp. 57–71. DOI: 10.3384/ecp2118157. (Visited on 2023-10-02).

Liu, C. Karen and Dan Negrut (2021-05-03). "The Role of Physics-Based Simulators in Robotics". In: *Annual Review of Control, Robotics, and Autonomous Systems* 4.1. Number: 1, pp. 35–58. ISSN: 2573-5144, 2573-5144. DOI: 10.1146/annurev-control-072220-093055. (Visited on 2023-10-02).

Magargle, Ryan et al. (2017-07-04). "A Simulation-Based Digital Twin for Model-Driven Health Monitoring and Predictive Maintenance of an Automotive Braking System". In: The 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, pp. 35–46. DOI: 10.3384/ecp1713235. (Visited on 2023-12-04).

Mikelsons, Lars and Roland Samlaus (2017-07-04). "Towards Virtual Validation of ECU Software using FMI". In: The 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, pp. 307–311. DOI: 10.3384/ecp17132307. (Visited on 2023-10-15).

Saad, Omar M. et al. (2023-12-01). "Earthquake Forecasting Using Big Data and Artificial Intelligence: A 30-Week Real-Time Case Study in China". In: *Bulletin of the Seismological Society of America* 113.6, pp. 2461–2478. ISSN: 0037-1106, 1943-3573. DOI: 10.1785/0120230031. (Visited on 2023-12-04).

Savaglio, Claudio et al. (2019-04). "IoT Services Deployment over Edge vs Cloud Systems: a Simulation-based Analysis". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Paris, France: IEEE, pp. 554–559. ISBN: 978-1-72811-878-9. DOI: 10.1109/INFCOMW.2019.8845305. (Visited on 2023-10-02).

Schranz, Thomas et al. (2021-09). "Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU". en. In: *Modelica Conferences*, pp. 419–424. ISSN: 1650-3740. DOI: 10.3384/ecp21181419. URL: https://www.ecp.ep.liu.se/index.php/modelica/article/view/220 (visited on 2025-02-19).

Stelzig, Philipp Emanuel and Benjamin Rodenberg (2023-12). "Simulation Model as a Service (SMaaS): A concept for integrated deployment, execution and tracking of system simulation models". en. In: *Modelica Conferences*, pp. 33–42. ISSN:

1650-3740. DOI: 10.3384/ecp20433. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/911 (visited on 2025-02-19).

Tavella, Jean-Philippe et al. (2016-09). "Toward an accurate and fast hybrid multi-simulation with the FMI-CS standard". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). Berlin, Germany: IEEE, pp. 1–5. ISBN: 978-1-5090-1314-2. DOI: 10.1109/ETFA.2016.7733616. (Visited on 2023-10-08).

Tollervey, Nicholas H. (2017). *Programming with MicroPython: Embedded Programming with Microcontroller and Python*. First edition. Beijing Boston Farnham: O'Reilly. 195 pp. ISBN: 978-1-4919-7273-1.

Vanommeslaeghe, Yon et al. (2018). "Towards Co-Simulation of Embedded Platforms and Physics-Based Models". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. ISSN: 978-1-5386-7383-6.

Zoting, Shivani (2023). *Simulation Software Market Size, Trends, Growth, Report 2030*. URL: precedenceresearch.com/simulation-software-market (visited on 2023-12-01).

A ufmu class and methods description

`simulation_instance`
Class representing a configured simulation instance:

- Implements iterator protocol for step-by-step simulation
- Maintains internal simulation state

`simulate(tStart, tEnd, h)`
Executes a complete simulation in one call.

- *Parameters:* `tStart`, `tEnd`, `h` (float): start time, end time, step size
- *Returns:* list of output value tuples

`setup_simulation(tStart, tEnd, h)`
Creates a simulation instance for step-by-step execution.

- *Parameters:* same as above
- *Returns:* a `simulation_instance` object

`get_variable_count()`
Returns the number of variables in the model.

- *Returns:* integer count

`get_variables_names(...)`
Retrieves variable names.

- *Parameters:* optional indices/names
- *Returns:* tuple of variable name strings

`get_variables_base_values(...)`
Retrieves initial values of variables.

- *Parameters:* optional indices/names
- *Returns:* tuple of values (float/int)

`get_variables_description(...)`
Retrieves variable descriptions.

- *Parameters:* optional indices/names
- *Returns:* tuple of variable description strings

`change_variable_value(...)`
Modifies a variable's value during simulation.

- *Parameters:*
  - `sim_instance` (object)
  - `var_ref` (int or str)
  - `value` (float)
- *Returns:* boolean success indicator