Extending Energy Models for Wireless Network Simulation with FMI-based Hybrid Co-Simulation

Lars Moons
Department of Computer Science
University of Antwerp
lars.moons@student.uantwerpen.be

***Abstract*—This paper proposes a novel workflow for extending energy models in wireless network simulation using hybrid co-simulation based on the Functional Mock-up Interface (FMI). Currently, adding new energy models to the ns-3 energy framework ns-3 requires a tedious workflow involving creating a circuit diagram, solving the circuit and writing the code, which can be time-consuming and error-prone. To address these challenges, the proposed workflow integrates visual design, drag-and-drop functionality, and predefined model building blocks from tools like OpenModelica and Simscape, which will interact with discrete-event simulators like ns-3 through FMI.**

***Index Terms*—energy modeling, ns-3, hybrid co-simulation, FMI, wireless network simulation**

I. Introduction

Energy modeling is a critical aspect of evaluating the performance of wireless network protocols. Therefore, in the network simulator ns-3, a generic energy framework is provided, with basic structures for modeling energy sources [1]. While there are some pre-built models available, more advanced energy models require programmers to write their own code. In a previous study [2], the authors encountered this situation, when they modeled a battery-less IoT device with energy harvesting using the ns-3 energy model interface. Their workflow involved creating a circuit diagram, solving the circuit using analysis techniques, and implementing the resulting formula in code. This approach proves effective for smaller and less complex circuits. However, when more realism is necessary, this workflow becomes challenging due to potential errors, time consumption, and unfamiliarity with circuit analysis techniques. Furthermore, integrating energy models developed by different teams using different toolsets can pose difficulties.

To overcome these challenges, we propose a new workflow that incorporates visual design, drag-and-drop functionality, and predefined model building blocks [3]. Existing tools like OpenModelica and Simscape offer such functionality, but they do not directly integrate with discrete-event simulators like ns-3. Hence, the challenge lies in combining these two environments through hybrid co-simulation. The integration can be made possible with the Functional Mock-up Interface (FMI) standard, which enables communication between the two environments. Previous work in [4], [5], and [6] has explored FMI-based hybrid co-simulation, but its application to simulating energy models in wireless network simulations appears to be a novel contribution.

In the following sections, we will briefly review discrete-event and physical simulation, discuss the FMI standard, and present two proof-of-concept implementations: one based on Python and another based on C++. We will talk about future work and finally conclude the paper.

II. Discrete-Event Simulation

In a discrete-event, simulation events occur at specific points in time. Each event marks a change of state in the system, and states can only change when events take place. The simulator maintains a queue of events that are scheduled to execute at a specified simulation time. These events are executed in sequential time order. After an event is completed, the simulator proceeds to the next event in the queue or terminates if there are no more events remaining. Discrete-event simulators are often used in network simulators to model events such as packet transmission or changes in network topology [7]. Two popular open-source frameworks in this domain are ns-3 and OMNeT++.

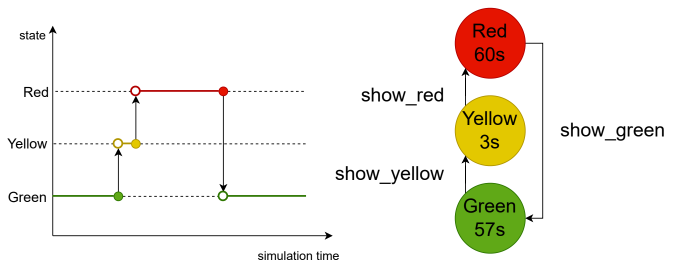


Fig. 1: Discrete-event system of a traffic light.

A simple example of a discrete-event system is a traffic light, see Fig. 1. The traffic light has three states: `RED`, `YELLOW` and `GREEN`, and is controlled by three events: `show_red`, `show_yellow` and `show_green`, which set the traffic light to the colour indicated by their names. The system starts in the `GREEN` state and schedules the `show_yellow` event to occur after 57 seconds. The simulator advances the simulation time to 57 seconds and executes the `show_yellow` event. During this event, the traffic light transitions to the `YELLOW` state, and the `show_red` event is scheduled to execute after 3 seconds, relative to the current simulation time. Continuing the simulation, the time is advanced to 60 seconds and the `show_red` event is executed.

As a result, the traffic light transitions to the `RED` state, and the `show_green` event is scheduled to execute after 60 seconds. After that, the simulation time is advanced to 120 seconds, and the `show_green` event is executed. This returns us to the initial state, and the process repeats.

III. Physical Simulation

Physical simulation employs mathematical equations to model the behaviour of physical objects. Unlike the discrete events, it operates in a continuous state space. Two popular tools that support physical simulation are OpenModelica and Simscape. These tools are available either as open-source software or free for academic research. They eliminate the need to numerically solve systems of equations that govern the laws of physics. Instead, they provide pre-written blocks that can be connected graphically and let the solver do all the tedious work. In addition to their graphical interfaces, these tools also support their own modeling language. These languages resemble regular programming languages but operate at a higher level of abstraction. They are equation-based rather than assignment-based, enabling acausal data flow that does not fix input or output variables. This flexibility allows inputs and outputs to be derived depending on the situation.

Let's consider a simple physical system, such as a discharging capacitor, see Fig. 7. The system consists of two main components: a capacitor and a resistor. Using the aforementioned tools, these components can be connected graphically (Fig. 2a). The graphical model is then translated into a textual modeling language (Fig. 2b), in which the behaviour of a discharging capacitor is described with a differential equation:

$$\frac{dV}{dt} = -\frac{V}{CR}$$

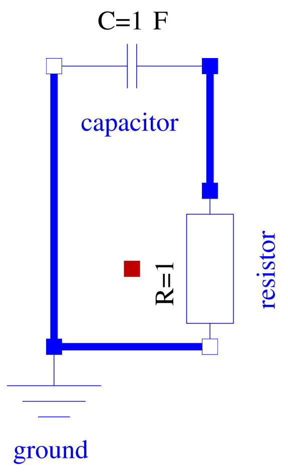
Finally (Fig. 2c), the equation is solved numerically using Euler's method which approximates the derivative by finite differences:

$$y'(t_0) \approx \frac{y(t_0 + h) - y(t_0)}{h}$$

Note that this example serves to provide intuition rather than demonstrate the exact working of OpenModelica or Simscape.

IV. FMI standard

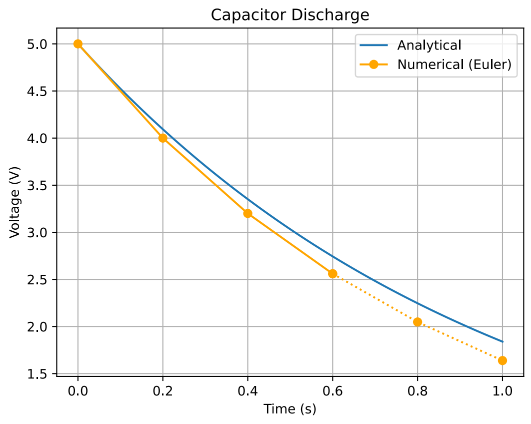
The Functional Mock-up Interface (FMI) is the de facto standard for co-simulation and model-exchange. It will enable the integration between discrete-event and physical simulation from the previous sections. OpenModelica and Simscape have both incorporated FMI into their software platforms, offering support for importing and exporting models. The FMI specification, currently at version 3.0, provides a generic and powerful interface However, as tooling support is still evolving, we will focus on Version 2.0, which includes all the necessary features for our specific use-case. FMI defines two types of interfaces, FMI for Co-Simulation (CS) and FMI for Model Exchange (ME). While CS operates as a stand-alone black-box, ME requires an external numerical solver. For our implementation, we use the CS type as it offers sufficient functionality.



(a) Graphical model

```
model CapacitorDischarge
  parameter Real C = 1;
  parameter Real Resistance = 1;
  parameter Real initialVoltage = 5;
  Real voltage(start=initialVoltage);
equation
  der(voltage) = -voltage / (C * Resistance);
end CapacitorDischarge;
```

(b) Modeling language



(c) Numerical solution

Fig. 2: Physical system of a discharging capacitor.

A Functional Mock-up Unit (FMU) encapsulates a model that exposes the FMI interface. It is packaged as a ZIP file containing an XML-based model description and a shared library that implements the FMI interface as a C API, as in Fig. 3. Additionally, an FMU may include source code and documentation. The shared library could be compiled from a single C source file. However, in practice, the functionality is often decomposed into three components: the FMI source, which precisely defines the functions according to the FMI standard; the co-simulation source, which includes a numerical solver used by all CS models; and the model source, which contains code specific to the particular physical system being modeled.

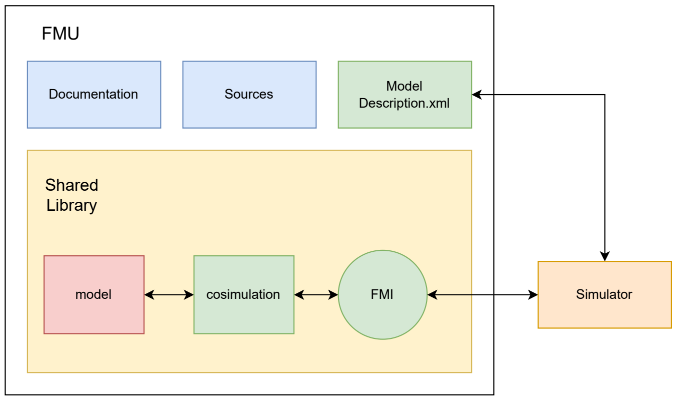


Fig. 3: Components of the FMU architecture.

V. Implementation

In this section, we will discuss the implementations, which are available on Github [1]. We will start by implementing it in Python, which is an ideal language for a proof-of-concept due to its simplicity, which allows us to focus on the core concepts rather than low-level details like memory management and build processes.

A. Discrete-Event Simulator

The discrete-event simulator is implemented as a Python class with the following methods:

- `schedule`: Places an event into the queue based on its scheduled time.
- `run`: Executes the simulation by executing and scheduling new events. It continues until there are no events left or a stop condition is met.
- `advance`: Advances the simulation time by `t` seconds by running the simulation for `t` seconds.
- `reset`: Clears the event queue and resets the simulation time to 0 seconds.

B. Physical Simulator

The physical simulator component is simple, as no implementation is required. We only need to download an existing FMU from either the Reference-FMUs project or one exported from OpenModelica.

C. Discrete Event Model

The discrete-event model is conceptually part of the discrete-event system, but connects with the physical system through the FMI interface. In Python, directly interfacing with the C-based FMI is not possible, so we need a Python binding. To simplify working with FMU models, such as extracting the ZIP file and parsing the XML description, we use FMPy, as it is recommended on the official FMI standard website and is actively maintained. FMPy provides access to the following functions from the FMI interface:

- `fmi2Instantiate`: initialises the FMU model state.
- `fmi2SetupExperiment`: Sets the simulation start and stop time.
- `fmi2EnterInitializationMode`, `fmi2ExitInitializationMode`: Do nothing, but are considered good practice to call.
- `fmi2GetReal` Retrieves values from the FMU model state.
- `fmi2SetReal` Sets values of the FMU model state.
- `fmi2DoStep` Simulates the FMU model up to the specified time using a numerical solver like Euler's method.
- `fmi2Terminate`, `fmi2FreeInstance`: Cleans up the model by freeing memory

For a sequence diagram, refer to Fig. 4. For an overview of the architecture, refer to Fig. **??**.

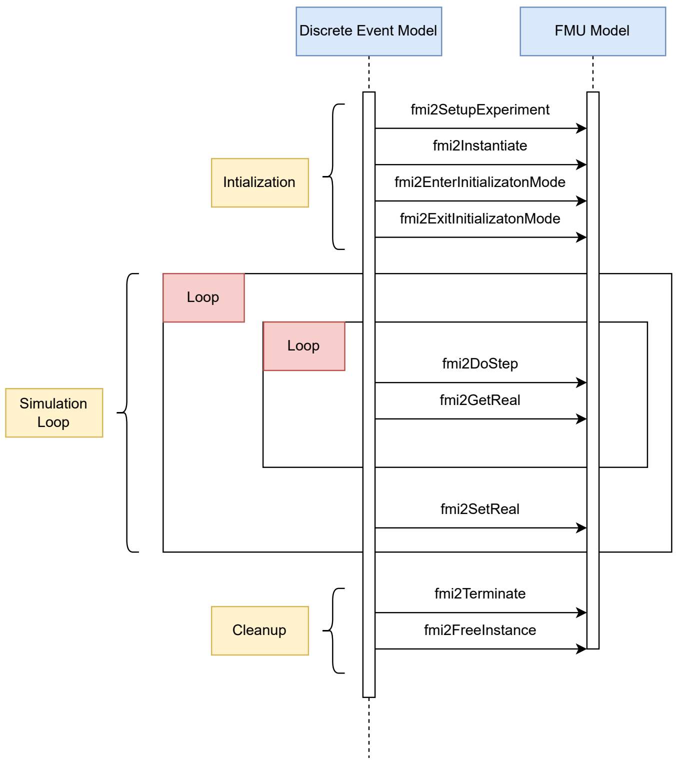


Fig. 4: Sequence diagram of FMI interaction.

D. Bouncing Ball FMU

To provide a concrete example, let's consider a model of a bouncing ball. The model was taken from the Reference-FMUs project, which contains hand-coded FMUs for testing and debugging. The behaviour of the bouncing ball is described by two differential equations:

$$\begin{cases} \frac{dh}{dt} = v \\ \frac{dv}{dt} = g \end{cases}$$

[1]https://github.com/lbm98/devs-fmu

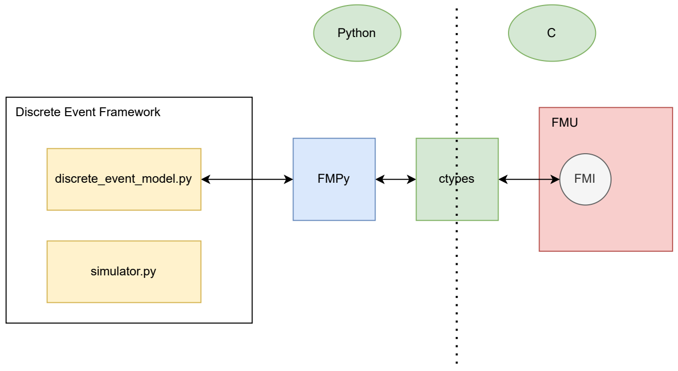


Fig. 5: Python implementation architecture.

relating height, velocity, and acceleration, along with two when-conditions

$$\text{when } h \leq 0 \text{ then} \quad \begin{cases} h := 0 \\ v := -e \cdot v \end{cases}$$

$$\text{when } v < v_{\text{min}} \text{ then} \quad \begin{cases} h := 0 \\ v := 0 \end{cases}$$

where e is the coefficient of restitution. The Github repository provides notebooks that visualise the behaviour of the bouncing ball by plotting its height over time, also shown in Fig. 6a. During the simulation, no discrete-events occurred, but the next experiment adds an event that resets the height and velocity to their initial values, see Fig. 6b.

We repeat the experiment using an FMU generated by OpenModelica for the same bouncing ball model, and the resulting behaviour is similar, as shown in Fig. 7a and Fig. 7b.

E. Implementation in C++

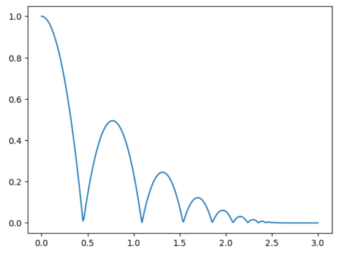
The target discrete-event frameworks ns-3 and OMNeT++ are written in C++. Integrating a Python runtime into the simulation would add significant overhead. Therefore, work has begun on the C++ implementation, but it is still in the early stages. This brings us to future work.

VI. Future Work

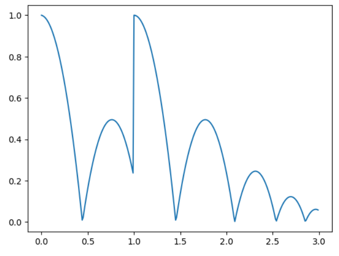
In future work, we plan to test additional models using both the Python and C++ implementations. This will help us verify their compatibility, performance and possible complications. Our overarching goal is to create a comprehensive library of energy models saved as OpenModelica or Simscape graphical models, that will seamlessly integrate with ns-3, enriching its energy model framework in a user-friendly, maintainable, and flexible way.

VII. Conclusion

In conclusion, this paper proposes a workflow that combines discrete-event simulation with physical simulation using the Functional Mock-up Interface (FMI). By integrating visual design and drag-and-drop functionality from tools like OpenModelica and Simscape with discrete-event simulators like ns-3, advanced energy models can be created more effectively. The implementation in Python demonstrates the feasibility of the approach, and work is underway to develop a C++ implementation for integration with ns-3. This workflow has the potential to enhance energy modeling in wireless network simulations and could ultimately lead to a comprehensive library of user-friendly, maintainable, and flexible energy models.

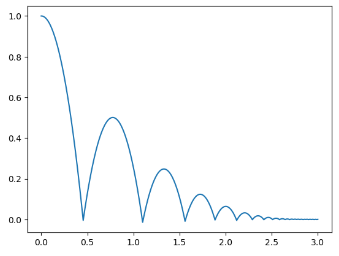


(a) Default simulation

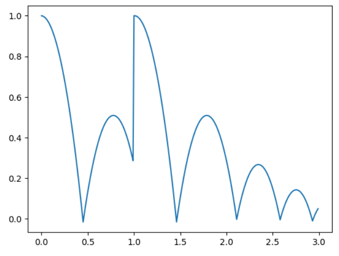


(b) Simulation with reset event

Fig. 6: Bouncing ball simulated with Reference FMU model.



(a) Default simulation



(b) Simulation with reset event

Fig. 7: Bouncing ball simulated with OpenModelica model.

References

[1] H. Wu, S. Nabar, and R. Poovendran, “An energy framework for the network simulator 3 (ns-3),” in *4th International ICST Conference on Simulation Tools and Techniques*, 2012.

[2] M. Capuzzo, C. Delgado, J. Famaey, and A. Zanella, “An ns-3 implementation of a battery-less node for energy-harvesting internet of things,” in *Proceedings of the 2021 Workshop on ns-3*, 2021, pp. 57–64.

[3] P. Carreira, V. Amaral, and H. Vangheluwe, *Foundations of multi-paradigm modelling for cyber-physical systems*. Springer Nature, 2020.

[4] F. Cremona, M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, “Hybrid co-simulation: It’s about time,” *Software & Systems Modeling*, vol. 18, pp. 1655–1679, 2019.

[5] F. Cremona, M. Lohstroh, D. Broman, M. Di Natale, E. A. Lee, and S. Tripakis, “Step revision in hybrid co-simulation with fmi,” in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, IEEE, 2016, pp. 173–183.

[6] J.-P. Tavella, M. Caujolle, C. Tan, *et al.*, “Toward an hybrid co-simulation with the fmi-cs standard,” 2016.

[7] G. F. Riley and T. R. Henderson, “The ns-3 network simulator,” *Modeling and tools for network simulation*, pp. 15–34, 2010.