FMIExchange.jl: Foreign Models Through a Familiar Interface

Lucas Bex[1, 2] and Geert Deconinck[1, 2]

[1]dept. of Electrical Engineering - KU Leuven
[2]EnergyVille

ABSTRACT

The Functional Mock-up Interface (FMI) standard [1] facilitates interoperability of differential algebraic equation (DAE) simulation software. Compliant models (so-called Functional Mock-up Units (FMUs)) can be simulated through a standardized interface, which allows to easily exchange these models between different software tools. The Julia ecosystem contains an extensive FMI package suite that supports (among others) simulation, import, export and differentiation of FMUs. This paper presents FMIExchange.jl [4], which provides utilities to simplify interfacing FMUs with the popular DifferentialEquations.jl package [5]. Additionally, FMIExchange.jl provides tools to compose simulations combining multiple FMUs and native Julia models.

Keywords

Julia, Differential equation, Functional Mock-up Interface, Simulation, Model Integration

1. Statement of Need

The Functional Mock-up Interface (FMI) standard [1] presents a standardised software interface for simulating an ordinary differential equation (ODE) with events to enhance interoperability of simulation and modelling tools. Many differential algebraic equation (DAE) modelling tools have the possibility to export models (after transformation to an ODE) to Functional Mock-Up Units (FMU, Fig. 1), which comply with this interface. This standardisation allows to co-simulate multiple dynamic models in a different software environment from the one they were developed in.
FMUs define a set of states states u and inputs p_i required for simulating the model (1), as well as a set of outputs p_o (2) which are exposed to the user. FMUs may contain events, that cause discontinuouties in the solution of the ODE. Events may trigger at fixed times or whenever the state/inputs fulfil some condition. Three types of FMU are available: Co-Simulation FMUs only expose inputs and outputs and come bundled with an integrator and event handler, Model Exchange (ME-)FMUs require an external ODE integrator/event handler and expose all quantities, Scheduled Execution FMUs expose model partitions, algorithms to compute quantities of interest, and require a scheduler to trigger execution of these model partitions; this text focuses on ME-FMUs.

$$f(u, p_i, t) = \dot{u} \tag{1}$$
$$g(u, p_i, t) = p_o \tag{2}$$

An FMU bundles an ODE model into a (binary) shared library and accompanying XML description of the model; this allows to call the model from other programs e.g. using Julia's `ccall` functionality. Within the Julia ecosystem, FMI.jl and related packages [6] provide powerful functionality for interacting with and simulating individual FMUs. While it is possible to combine FMUs with other (native Julia or FMU-based) models using aforementioned packages, this requires familiarity with the FMI standard and significant effort from the user to ensure correctness. FMIExchange.jl aims to simplify this process and make FMUs easier to use for an audience that is unfamiliar with the FMI standard. To this end, FMIExchange.jl predefines callbacks to correctly handle FMU events and integration. Additionally, some functionality is provided to simplify composing multiple (FMU-based or native) models in a single DifferentialEquations.jl-based simulation.

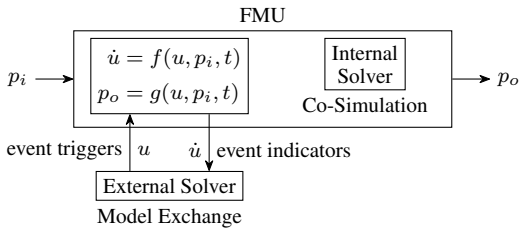

Fig. 1: FMUs present a standardized software interface for simulating ODEs. A solver can be bundled with the FMU (Co-Simulation) or the FMU may use an external solver (Model Exchange).

2. Composition Functionality

To compose multiple models in a simulation, their inputs/outputs must be appropriately connected to each other and all dynamics must be bundled into a single function of the form `f(u, p, t)` or `f!(du, u, p, t)` and a set of callbacks. To this end, FMIExchange.jl works with the simulation model depicted in Fig. 2: all model states u_j are bundled in a flat `u` vector; model inputs/outputs are connected through an intermediate buffer which is the parameter `p` argument in `f`. The `p` argument is accessible both to callbacks and the dynamics functions and thus this approach allows for the most flexibility in composing simulations. Here, FMIExchange.jl differs from other packages, such as most packages in the SciML ecosystem: the `p` argument is used to store time-varying quantities rather than parameters which are fixed during the simulation.
To aid the user in composing simulations of the format in Fig. 2, FMIExchange.jl provides i) functionality to combine dynamics functions f_j into a DifferentialEquations.jl-compatible function `f` (with in-place and out-of-place methods), ii) predefined callbacks to automatically update model output between integration timesteps, and iii) human-readable address map generation func-

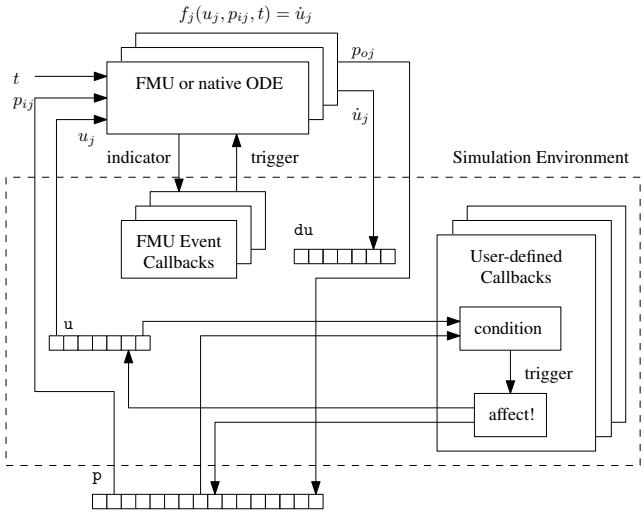

Fig. 2: Composing multiple models in a simulation

tionality for both the `u` and `p` argument. FMIExchange.jl does not provide any simulation capabilities itself; it generates callbacks and dynamics functions to use in the DifferentialEquations.jl framework. As such, the user retains the full flexibility of the DifferentialEquations.jl interface: it is possible to use FMIExchange.jl to compose some models, and bypass it for other models in that same simulation. This flexibility includes the ability to reintroduce fixed simulation parameters back into the `p` vector if so desired.

For the addressing and state/input/output bundling functionality, FMIExchange.jl relies on custom types *AbstractSimModel* and *AbstractModelSpecification*. Addresses in the form of indices of `p`/`u` and model functionality (dynamics function and callbacks) are contained in the *AbstractSimModel*. The *AbstractSimModel* (or a vector of them) is used when bundling the dynamics functions and generating callbacks for automatically calling model output functions between integration steps. To avoid having to manually assign addresses and provide a human-friendly interface, FMIExchange.jl can automatically generate address maps from human-readable *AbstractModelSpecification*s. *AbstractModelSpecification*s specify states/inputs/outputs by name rather than address and can be converted directly into *AbstractSimModel*s using an address map.

The demo [3] provides a practical example of an FMIExchange.jl worklflow. The demo implements a model-predictive (MPC) and a rule-based controller for a household energy system. In this demo, there is some model mismatch between the optimisation model of the MPC and the physical system. The effect of this model mismatch is studied through dynamic simulation of the system.

3. FMU Utilities

Function Mock-Up Units are merely a special kind of model to include in a simulation. As such, all functionality from Section 2 is the same for native ODE models and ME-FMU models. However, if not using FMIExchange.jl, correctly handling integration of ME-FMUs requires the user to interact directly with the FMU code object as implemented by the FMI.jl package which adheres strictly to the FMI standard. FMIExchange.jl handles this for the user and does not require any familiarity with FMI.jl or the FMI standard. Note that it is possible to simulate multiple FMUs and native ODEs using solely FMI.jl and DifferentialEquations.jl, but FMIExchange.jl greatly simplifies this process through its *AbstractSimModel* interface. The FMU-handling functionality can be used on its own or together with the simulation composition functionality in Section 2. FMIExchange.jl is currently compatible with FMI version 2 [2].

FMIExchange.jl exposes all FMU states/inputs/outputs in the `u` and `p` variables for easy access. Additionally, FMIExchange.jl can generate dynamics functions and callbacks that correctly embed the FMU in a DifferentialEquations.jl simulation. The functionality includes: handling time-based, state-based and input-based FMU events, switching between FMU modes for calculating derivatives/calculating events/finishing integration steps, and automatically recalculating FMU outputs after an integration step. Finally, FMIExchange.jl preallocates and reuses caches to avoid excessive memory allocations at each `ccall` call to the FMU.

4. Conclusion

FMIExchange.jl simplifies composing complex simulations with multiple interacting ODE models that are implemented either in Julia or as a Functional Mock-up Unit. The package achieves this by providing functionality for bundling model dynamics functions of different models and for easily generating various commonly needed callbacks to embed in a DifferentialEquations.jl simulation. Automatic address map generation and support for human-readable addresses is included in the package to simplify composition of many models in a simulation. FMIExchange.jl itself does not provide any simulation functionality, but allows the user to retain full control through the powerful DifferentialEquations.jl interface.

Within this framework, Functional Mock-up Units are merely another type of model, allowing seamless composition of native Julia and FMU-based models. FMIExchange.jl provides functionality for automatically generating the required callbacks and dynamics function for correct FMU simulation. As a result, users can easily compose native Julia models and FMUs in a single simulation, without requiring extensive knowledge of the FMI standard.

5. Acknowledgements

The authors acknowledge the financial support from KU Leuven through the TECHPED - C2 project (C24M/21/021) and the Research Foundation - Flanders (FWO) (grant no. 1S00325N).

6. References

[1] Modelica Association. Functional Mock-up Interface. url: https://fmi-standard.org/.

[2] Modelica Association. Functional Mock-Up Interface for Model Exchange and Co-Simulation. url: https://github.com/modelica/fmi-standard/releases/download/v2.0.4/FMI-Specification-2.0.4.pdf, November 2022.

[3] Lucas Bex. FMIExchange demo. url: https://gitlab.kuleuven.be/u0138844/fmiexchangedemo, Jan 2024.

[4] Lucas Bex and Geert Deconinck. FMIExchange.jl. doi:10.5281/zenodo.18710589.

[5] Christopher Rackauckas and Qing Nie. Differentialequations.Jl–a Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 5(1):15, May 2017. doi:10.5334/jors.151.

[6] Tobias Thummerer, Lars Mikelsons, and Josef Kircher. NeuralFMU: Towards Structural Integration of FMUs into Neural Networks. In *Proceedings of the 14th Modelica Conference*, pages 297–306, Sweden, September 2021. doi:10.3384/ecp21181297.