Project title: Real-time damage assessment of R/C bridges based on an innovative low-cost acceleration sensing system

FMU Development

Deliverable number: D2.2

Version 0.5/1.0

Funded by the Hellenic Foundation for Research and Innovation
No. 73263

Project Acronym: RISE
Project Full Title: Real-time damage assessment of R/C bridges based on an innovative low-cost acceleration sensing system
Call: 2nd Call for H.F.R.I. Research Projects to Support Faculty Members and Researchers
Grant Number: 73263
Project URL: www.rise-project.gr

Deliverable nature:	Report (R)
Contractual Delivery Date:	May 31, 2024
Actual Delivery Date	May 30, 2024
Number of pages:	5
Keywords:	FMU, FMI Standard, Python Library
Authors:	**REDI Engineering Solutions PC** **Konstantinos Mixios** - *PhD Candidate, Aristotle University of Thessaloniki* **Vassilis Papanikolaou** - *Assistant Professor, Aristotle University of Thessaloniki* **Olga Markogiannaki** - *PostDoc Researcher, Aristotle Univeristy of Thessaloniki*

Contents

List of Figures

List of Tables

1. Introduction

Functional Mock-up Units (FMUs) are a key component in the field of modeling and simulation, particularly within the context of the Functional Mock-up Interface (FMI) standard. FMUs are self-contained, reusable software components that represent a part or the entirety of a system model. They are designed to be used in various simulation environments and tools, providing a standardized way to exchange dynamic system models.

FMUs are based on the FMI standard, which defines how models can be packaged and exchanged between different software tools. The Functional Mock-up Interface (FMI) standard is a versatile and widely adopted framework for model exchange and co-simulation in systems modeling and simulation. Developed by the Modelica Association, FMI facilitates the seamless integration and interoperability of models across different tools and platforms, promoting reusability and efficiency. Key features of the FMI standard include standardized interfaces for communication, cross-platform compatibility, encapsulation of model descriptions and code, and support for hierarchical modeling. The standard is supported by a wide range of simulation tools, fostering a rich ecosystem for model exchange and co-simulation. FMI's advantages include enhanced interoperability, reusability, scalability, and collaboration. It is extensively used in industries such as automotive, aerospace, energy systems, and industrial automation, enabling comprehensive system-level analysis and optimization.

An FMU encapsulates several elements: a model description (an XML file that describes the model's variables, parameters, inputs, and outputs), a model implementation (compiled code, often in the form of a shared library, or source code that implements the model's behavior), and any additional data necessary for the model, such as tables or images.

There are two main types of FMUs: Co-Simulation FMUs and Model Exchange FMUs. Co-Simulation FMUs contain their own solver for time integration and are responsible for advancing their simulation state independently. They are used when the model needs to be simulated alongside other models, potentially with different solvers. Model Exchange FMUs, on the other hand, do not contain their own solver and rely on the importing tool to perform time integration. These are used when the model is integrated into a larger system simulation where a single solver manages all components.

FMUs are important in modeling and simulation for several reasons. First, they promote standardization and interoperability. FMUs follow the FMI standard, which ensures compatibility across different tools and platforms, allowing models developed in one tool to be used in another without modification. This standardization facilitates collaboration and model exchange between different teams, departments, and organizations using different simulation tools. Designed to be reusable components, once developed, an FMU can be used in multiple projects and contexts, saving time and effort in model development. This reusability also enhances the modularity of simulation models, enabling the reuse of validated and tested components. FMUs provide flexibility and integration capabilities. They can represent various types of systems, including mechanical, electrical, thermal, hydraulic, and control systems, making them suitable for multi-domain simulations. Additionally, FMUs can be integrated into larger simulation environments, facilitating the simulation of complex systems with components from different domains. By encapsulating the model and its solver (in the case of co-simulation FMUs), FMUs can optimize simulation performance for specific tasks. This encapsulation allows parallel development and testing of different system components, improving overall development efficiency. They enable scalable simulation setups, ranging from simple component models to complex system simulations involving numerous interconnected FMUs. This scalability allows for hierarchical modeling where high-level system models are composed of interconnected FMUs representing subsystems. Finally, the broad adoption of the FMI standard and FMUs in industries such as automotive, aerospace, energy, and manufacturing drives further development and support from various simulation tool vendors. This widespread industry adoption underscores the significance of FMUs in advancing modern simulation practices.

2. FMU Development

2.1. Description of FMU implementation

The purpose of this deliverable was to develop an FMU module for a finite element model (FEM) that is executable with OpenSees. The FMU model is designed to accept input variables that will be optimized in the subsequent phases of the project. These input variables will be adjusted through a model update process, based on acceleration recordings, to ensure that the results of the modal analysis correspond to the performance of the physical model. To achieve this, a Python library was developed to encapsulate an OpenSeesPy model (an OpenSees model implemented in Python) into an FMU format file compliant with FMI standards. This library is responsible for converting the OpenSeesPy model into a format compatible with all FMI standard parsers. Additionally, an orchestrator was developed to manage the input variables of the FMU file, and retrieve the results of the modal analysis. We have to note that the developed library supports only the Co-Simulation type of FMUs and not the Model Exchange FMUs at this point. The difference between the two FMU types is explained in the previous chapter.

2.2. Deliverable execution instructions

To use the developed library and build an FMU model, the user should follow the steps below:

1. Create a custom Python environment and activate it via Anaconda Python distribution.
   The code block below creates a new Python environment and activates it. This procedure should be executed within a designated folder. The user should install Anaconda python distribution in order to execute the following code.

```
conda create --name RISE-env python=3.11
conda activate RISE-env
```

2. Install the developed library. The code block below, will install the developed library to the new virtual environment.

```
python -m pip install fmup-1.5.10.0-py3-none-any.whl
```

3. Install all dependencies specified in the provided requirements file to ensure the ability to build and run the FMU.

```
python -m pip install -r requirements.txt
```

4. Develop Python model according to the guidelines provided by the library and build the FMU. In order to use the python library and build the FMU, the user should execute in the terminal the following code snippet. The script-file.py is the name of the python file, which is formed with respect to the library.

```
fmup build -f script-file.py requirements.txt
```

5. The generated FMU file is now accessible via the fmpy library and can be executed from the orchestrator. The associated script in the repository is the *main.py* file.

The Python model structure, designed for assembly into an FMU using the developed library, should adhere to the following architectural layout.

```
from fmup import Fmi2Causality, Fmi2Slave, Boolean, Integer, Real, String

class Model(Fmi2Slave):
        author = "REDI-ENG."
        description = "How-to-define-input-varibles"

        def __init__(self, **kwargs):
                super().__init__(**kwargs)
                self.intOut = 1
```

```
        self.realOut = 3.0
        self.register_variable(Integer("intOut", causality=Fmi2Causality.output))
        self.register_variable(Real("realOut", causality=Fmi2Causality.output))


    def do_step(self, current_time, step_size):
        # Within this method, the solution of the model will be undertaken.
        return True
```

We should note that the developed library does not bundle Python, which makes it a tool coupling solution. This means that you can not expect the generated FMU to work on a different system (the system would need a compatible Python version and libraries). However, to simplify the procedure the wrapper uses Python API, making the pre-built binaries for Linux and Windows 64-bits compatible with any Python 3 environment. If you need to compile the wrapper for a specific configuration, you will need CMake and a C++ compiler. fmup does not automatically resolve 3rd party dependencies. If your code includes e.g. numpy, the target system also needs to have numpy installed. This is the reason why the user should load the requirements.txt file.

A. Appendix

Repository: www.github.com/rclab-auth/RISE-D2.2

The above repository contains the following files.

1. fmup library wheel
2. An example file of how the numerical model should be structured in order to be wrapped to a FMU file utilizing the developed library
3. The FMU of a bridge model developed with OpenSeesPy.
4. An orchestrator python file in order to call and execute the FMU.
5. An extented README.md file elucidating additional features of the developed library.
6. a .bash file for example execution.