ENABLING PYTHON DRIVEN CO-SIMULATION MODELS WITH PYTHONFMU

Hatledal, Lars Ivar*
Zhang, Houxiang
Department of Ocean Operations and Civil Engineering
Norwegian University of Science and Technology
Postbox 1517, 6025 Aalesund, Norway

Collonval, Frédéric
Modeling & Simulation
Safran Tech
CS80112 Chateaufort
78772 Magny Les Hameaux, France

KEYWORDS

Co-simulation; Modelling; FMI; FMU; Python

ABSTRACT

This paper introduces PythonFMU, an easy to use framework for exporting Python 3.x code as co-simulation compatible models compliant with version 2.0 of the Functional Mock-up Interface (FMI). The framework consists of a set of helper classes and a command line utility for transforming compliant python source into ready to use cross-platform FMUs. PythonFMU seamlessly takes care of a number of low-level FMI functions such as getting and setting variable values, and state handling, including serialization and deserialization. Furthermore it provides pre-built binaries for Windows and Linux 64-bits, generates the required *modelDescription.xml* containing meta-data about the model and packages all related files into a Functional Mock-up Unit (FMU) - ready to be imported into any FMI compatible simulation tool. The framework can be effortlessly installed using de-facto standard Python package managers pip and conda. While PythonFMU is more geared towards ease of use and enabling Python driven co-simulation models, it is shown to have adequate performance compared to much more low-level alternatives targeting other programming languages.

INTRODUCTION

The Functional Mock-up Interface (FMI) [Blochwitz et al., 2012] is a tool independent standard managed by the Modelica Association that supports both Model Exchange (ME) and Co-Simulation (CS) of dynamic models. A key goal of FMI is to improve the exchange of simulation models between suppliers and original equipment manufacturers (OEMs). The current major version of the standard is 2.0, which was released in 2014. A minor revision, 2.0.1, was released in 2019.

An FMU is a model that implements the FMI standard and is distributed as a zip-file with the extension *.fmu*. This archive contains:

- An XML-file that contains meta-data about the model, named modelDescription.xml.
- C-code implementing a set of functions defined by the FMI standard.
- Other optional resources required by the model implementation.

The FMI standard consists of two main parts, both of which a single FMU may support:

- FMI for ME: Models are exported without solvers and are described by differential, algebraic, and discrete equations with time-, state-, and step-events.
- FMI for CS: Models are exported with a solver, and data is exchanged between subsystems at discrete communication points. In the time between two communication points, the subsystems are solved independently from each other.

The work presented in this paper, however, is only concerned about the co-simulation part of the standard.

Many tools support importing co-simulation FMUs, however, fewer tools supports exporting such FMUs. Many of whom are commercial and or domain specific. Furthermore, FMUs generated with these tools may not support the optional parts of the standard such as state handling, which are required by some more advanced co-simulation algorithms in order to achieve better numerical accuracy and stability during simulations[Broman et al., 2013, Cremona et al., 2016, Tavella et al., 2016].

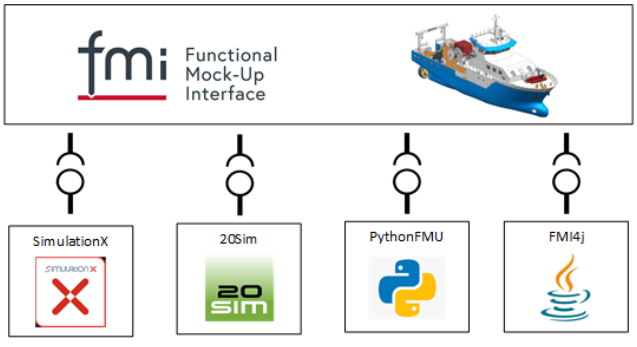

Fig. 1: Possible use of PythonFMU in realizing complex cyber-physical systems using FMI based co-simulation.

Python [Van Rossum et al., 2007] is one of the most popular programming language today [O'Grady, 2020]. The major reasons for that are the ease of learning the language, the huge spectra of libraries covering fields

*Corresponding author. E-mail: laht@ntnu.no

Communications of the ECMS, Volume 34, Issue 1,
Proceedings, ©ECMS Mike Steglich, Christian Mueller,
Gaby Neumann, Mathias Walther (Editors)
ISBN: 978-3-937436-68-5/978-3-937436-69-2(CD) ISSN 2522-2414

such as video game, machine learning, web server or scientific computing and the recent explosion of data science in which Python plays a central role. Of particular importance for scientific computing is the creation of the *Numpy*[Oliphant, 2007] library that bridge the gap between efficient code in C or Fortran languages and the ease of a scripting language. That library is now at the heart of all major scientific Python libraries from Pandas for data analysis to Scipy for classical algebra operators or Scikit-learn for machine learning analysis.

This paper introduces PythonFMU, an easy to use Python based framework that allows plain Python code to be exported as FMI compatible co-simulation FMUs. Figure. 1 shows how PythonFMU could potentially be used to implement complex cyber-physical systems that are aggregates of models from different simulation domains.

The paper is organized as follows. Firstly some related works are given. After which PythonFMU is introduced. Then some benchmark results are presented. Finally some concluding remarks and notes on future works are provided.

RELATED WORK

A number of open-source software frameworks for exporting FMUs from source-code have been developed in the recent years. While many more tools are capable of exporting FMUs like 20Sim, OpenModelica, MATLAB and SimulationX, this paper is more focused on frameworks that allows the generation of FMUs from plain source-code. Each of these are described in more detail below.

CPPFMU [SINTEF Ocean, 2017] is a set of interfaces and helper functions for writing FMI-compliant model/slave code in C++ using high-level features such as exceptions and automatic memory management, rather than having to implement the low-level C functions specified by FMI. However, while CPPFMU makes implementing and compiling the shared library required by an FMU, it does not handle generating the *modelDescription.xml* nor packaging of the FMU. CPPFMU was developed as part of the R&D project Virtual Prototyping of Maritime Systems and Operations (ViProMa) Hassani et al. [2016], and is currently maintained by SINTEF Ocean.

FMUSDK [QTronic, 2017] is a free, BSD licensed, software development kit (SDK) provided by QTronic to demonstrate basic use of FMUs for ME and for CS as defined by FMI version 1.0 and 2.0. The software is written in C++, but models are to be implemented in C. The first version of FMU SDK was released already in 2010, with the latest version, *2.0.6*, being released in 2018.

Like CPPFMU, FMUSDK does not auto-generate the *modelDescription.xml*. The main difference between these tools is that CPPFMU provides a more high level and structured API in C++, whereas FMUSDK requires source-code to be written in quite low-level C. On the other hand, FMUSDK supports ME and provides utilities for packaging the model as an FMU, whereas CPPFMU only provides helper functions to aid in development.

JavaFMI [Galtier et al., 2017] is a set of components for working with the FMI standard using Java, developed by SIANI institute (Las Palmas University) and funded by the European Institute for Energy Research (EIFER). It support both import and export of FMUs. For export, it support FMI 2.0 for Co-simulation. Generated FMUs runs both on Linux and Windows. JavaFMI has been actively maintained since its inception in 2013 and is licensed under the LGPLv3.

FMI4j [Hatledal et al., 2018] is a MIT licensed software package for dealing with Functional Mock-up Units (FMUs) on the JVM. It support both import and export of FMUs. For export, it support FMI 2.0 for Co-simulation. FMI4j is written in Kotlin, which is 100% interoperable with Java. On the native side, FMI4j makes use of CPPFMU to implement the FMI functions. FMUs generated using FMI4j can run on both Linux and Windows.

While both JavaFMI and FMI4j allows FMUs to be created using the Java language, they differ quite a bit in their implementation and usage. JavaFMI uses message-passing to bridge Java and the underlying C functions defined by FMI, while FMI4j relies on the Java Native Interface (JNI) for this. Consequentially, FMI4j generates much faster executables. Another key difference is how users define their model. JavaFMI is imperative, e.g meta-data is defined using API functions. FMI4j on the other hand is declarative, with meta-data defined using annotations.

Evidently, some open-source software for generating FMUs from source code already exists. See Table. I for a summary. However, only the ones targeting the JVM can be said to be easy to use as these manages everything related to the creation of an FMU. Still, the JVM may not be a natural choice for many for implementing models and the barrier for using these tools are high for non Java developers. CPPFMU and FMUSDK both eases the process within the realm of C/C++, but still requires significant know-how in order to produce a ready to use FMU. Furthermore, these tools only covers a small subset of available programming languages.

TABLE I: Open-source framework for exporting source-code as FMUs.

Tool	**Target language**	**Target platform**	**FMI version**
JavaFMI	JVM	Win, Linux	2.0
FMI4j	JVM	Win, Linux	2.0
CPPFMU	C++	Win[a], Linux[a]	1.0 & 2.0
FMUSDK	C	Win[a], Linux[a], OSX[a]	1.0 & 2.0

[a] Binaries are only built for the current platform.

PYTHONFMU

PythonFMU is a MIT licensed framework that enables the packaging of Python 3.x code as co-simulation FMUs, currently maintained in collaboration between NTNU and Safran Tech. The library required by users to implement their own FMI co-simulation slaves as well as the Command Line Interface (CLI) required to build the actual FMU is easily retrieved using either the *pip* or *conda* package managers. Unlike some FMU exporters, FMUs built with PythonFMU runs out of the box on both Windows and Linux 64-bit systems. PythonFMU has been implemented using the limited Python API, which makes it compatible with any Python 3.x version. However, PythonFMU does not bundle a Python distribution, which means that a compatible Python distribution must be present on the target system for the FMU to work. The same is true for any imported 3rd party libraries. Consequentially, if the slave makes use of e.g. the *numpy* package for scientific computing, this library must already be present on the target system. To remedy this, PythonFMU allows users to specify any dependency it should have on 3rd party libraries. This information is bundled with the FMU as a standard *requirements.txt* for use with one of Python's package managers. Thus allowing end-users to easily figure out what kind of libraries that must be installed for it to run on a particular machine.

Listing 1: Writing FMI 2.0 compatible slaves in Python using PythonFMU.

```
from pythonfmu import *

class PythonSlave(Fmi2Slave):

    author = "John Doe"
    description = "A simple description"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.realOut = 0.1
        self.register_variable(Real("realOut",
            causality=Fmi2Causality.output))

    def do_step(current_time, step_size):
        return True
```

Listing. 1 shows the minimal required code to write FMI 2.0 compatible co-simulation models in Python using PythonFMU. Additional FMI functions like e.g. *setupExperiment*, *enterInitializationMode*, *exitInitializationMode* and *terminate* have default no-op implementations and may be overridden on demand. PythonFMU automatically handles getting and setting variables, logging, resetting, state handling, serialization and deserialization as well as generating the required *modelDescription.xml*. The fact that PythonFMU handles state handling makes it possible to use with advanced co-simulation master algorithms that depends on rollback capabilities, like variable step algorithms. This is important in order to achieve numerically stable and accurate simulation results. Listing. 2 shows how to build an FMU from Python source that implements the PythonFMU API using the accompanying CLI. Additional options may be specified, such as documentation and associated project files. The FMU built by PythonFMU contains pre-built binaries for Windows and Linux 64-bit. This lowers the threshold for using it tremendously compared to many exporting tools as a C++ compiler does not have to be installed and the user does not have to figure out how to cross-compile.

Like FMI4j, PythonFMU makes use of CPPFMU for implementing the C functions required by the FMI standard. This shows a clear utility for CPPFMU as an enabler for higher-level applications to support the export of FMI compatible co-simulation models.

Listing 2: Building an FMU from Python source using the PythonFMU CLI.

```
pythonfmu-builder -f PythonSlave.py
```

RESULTS

In the following some performance metrics for PythonFMU is given.

Table. II show the performance of PythonFMU compared to other similar tools. The FMUs used all implements the same model. The model does no computation during stepping, but defines a single real, integer, boolean and string variable. These variables are read by the importing tool after each iteration. 100.000 iterations were run. That makes for a total of 400.000 calls through the FMI API. The benchmark was performed on a computer running Windows 10 fitted with an Intel i7-8700k processor.

TABLE II: Time required to step a simple FMU with one integer, real, string and boolean variable 100.000 times. All variables are read after each step.

Tool	**Version**	**Time[s]**
FMUSDK	2.0.6	4.6
CPPFMU	-	4.6
FMI4j	0.30.0	6.1
JavaFMI	2.6.0	40
PythonFMU	0.6.0	7.9/7.3[a]

[a] Using lambdas for getters, as demonstrated in Listing. 3.

From the results we can see that FMUSDK and CPPFMU are equally fast, and as expected, faster than both FMI4j and PythonFMU. This is natural as both of these uses CPPFMU internally and have an additional overhead from having to cross the native bridge using JNI and the Python C API respectively. JavaFMI is by far the slowest contender, due to it's choice of using message passing over direct API calls through JNI. Note that PythonFMU provides two results. By supplying a lambda function to the optional *getter* and *setter* parameters of PythonFMUs *ScalarVariable* as

demonstrated in Listing. 3, users may increase performance of variable read/write. When not specifying lambda functions for the getter and setter, PythonFMU defaults to using the built in Python functions *getattr* and *setattr* respectively. Furthermore, the use of lambdas allows non Python fields to be used as variables.

Listing 3: Supplying a lambda for increased flexibility and performance at the cost of a slight increase in verbosity.

```
self.register_variable(Real("realOut",
            causality=Fmi2Causality.output
            getter=lambda: self.real))
```

Note that the results presented here does not necessarily translate to more complex models with more code evaluation, as the presented benchmark it is mainly interested in measuring the performance of raw FMI calls. As Python is an interpreted language it is naturally slower to run than e.g. C/C++.

CONCLUSIONS

This paper introduces PythonFMU, an easy to use framework for exporting Python code as FMI 2.0 for co-simulation compatible models. The framework is easy to install and requires very little boilerplate code, allowing users to focus on the problem at hand. This coupled with the fact that Python is an easy to use scripting language with a strong standard library and a rich set of 3rd party libraries makes it ideal for fast prototyping. Furthermore, the position Python has as a language for scientific computing should make PythonFMU a natural choice for data scientists that want to take advantage of or contribute to co-simulation technology. In fact, PythonFMU was specifically developed to allow data scientist with little or no background from co-simulation or software engineering at NTNU to contribute with models related to the development of digital twins, as Python and it's strong ecosystem of libraries allows easy integration with e.g. models that is connected to web services or utilizes neural networks. While the focus of PythonFMU is ease of use and being an enabler for Python driven co-simulation models, the performance is shown to be quite adequate compared to more low-level implementations.

Future works includes adding more features, bug-fixes and improving documentation. The source is available at `https://github.com/NTNU-IHB/PythonFMU`, and users are encourage to contribute.

ACKNOWLEDGMENT

The research presented in this paper is supported by the Norwegian Research Council, SFI Offshore Mechatronics, project number 237896.

References

T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 076, pages 173–184. Linköping University Electronic Press, 2012.

D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of fmus for co-simulation. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–12. IEEE, 2013.

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)*, pages 173–183. IEEE, 2016.

V. Galtier, M. Ianotto, M. Caujolle, R. Corniglion, J.-P. Tavella, J. É. Gómez, J. J. H. Cabrera, V. Reinbold, and E. Kremers. Building parallel fmus (or matryoshka co-simulations). In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 663–671. Linköping University Electronic Press, 2017.

V. Hassani, M. Rindarøy, L. T. Kyllingstad, J. B. Nielsen, S. S. Sadjina, S. Skjong, D. Fathi, T. Johnsen, V. Æsøy, and E. Pedersen. Virtual prototyping of maritime systems and operations. In *ASME 2016 35th International Conference on Ocean, Offshore and Arctic Engineering*. American Society of Mechanical Engineers Digital Collection, 2016.

L. I. Hatledal, H. Zhang, A. Styve, and G. Hovland. Fmi4j: A software package for working with functional mock-up units on the java virtual machine. In *The 59th Conference on Simulation and Modelling (SIMS 59)*. Linköping University Electronic Press, Linköpings universitet, 2018.

S. O'Grady. The redmonk programming language rankings: January 2020, 2020. URL `https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/`. (Date accessed 08-March-2020).

T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

QTronic. Fmusdk, 2017. URL `http://www.qtronic.de/de/fmusdk.html`. (Date accessed 06-March-2020).

SINTEF Ocean. Cppfmu, 2017. URL `https://github.com/viproma/cppfmu`. (Date accessed 06-March-2020).

J.-P. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. 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)*, pages 1–5. IEEE, 2016.

G. Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007.

AUTHOR BIOGRAPHIES

LARS IVAR HATLEDAL received the B.Sc. degree in automation and the M.Sc. degree in simulation and visualization from the Norwegian University of Science

and Technology (NTNU), Ålesund, Norway, in 2013 and 2017, respectively, where he is currently pursuing the Ph.D. degree with the Department of Ocean Operations and Civil Engineering. Email: laht@ntnu.no

DR. FRÉDÉRIC COLLONVAL is the lead developer of a collaborative multi-systems and multi-physics simulation tool, of the Collaborative System Design team at Safran R&D center. He obtained his PhD in numerical simulation and modeling in gas turbine combustion chamber at TU Munich. He then joined Safran Group to work on airplane engine performance modeling and associated simulation tools. In 2018, he co-founded a new kind of collaborative simulation tool to target multi-physics simulation in pre-design phase for rapid design evaluation at Safran. He is interested in enhancing physical simulation tools by leveraging the features of innovative open-source projects.

PROF. HOUXIANG ZHANG received the Ph.D. degree in mechanical and electronic engineering, in 2003, and the Habilitation degree in informatics from the University of Hamburg, in February 2011. Since 2004, he has been with the Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, Institute of Technical Aspects of Multimodal Systems (TAMS), University of Hamburg, Germany. He joined the NTNU, Norway, in April 2011, where he is currently a Professor of robotics and cybernetics. His research interests lies on two areas: one is on biological robots and modular robotics and the other is on virtual prototyping and maritime mechatronics.