RAPID PROTOTYPING OF SELF-ADAPTIVE-SYSTEMS USING PYTHON FUNCTIONAL MOCKUP UNITS

Christian Møldrup Legaard
Cláudio Gomes
Peter Gorm Larsen

DIGIT, Department of Engineering, Aarhus University
Finlandsgade 22, 8200 Aarhus N, Denmark
{cml,claudio.gomes,pgl}@eng.au.dk

Frederik Forchhammer Foldager

Agrointelli
Agro Food Park 13, 8200 Aarhus N
Denmark
ffo@agrointelli.com

ABSTRACT

During the development of Cyber-Physical Systems (CPSs), it is crucial to enable efficient collaboration between different disciplines. Co-simulation plays a key role in this by allowing the system as a whole to be simulated by composing simulations of its parts. The ability to do this coupling relies on the models adhering to a well-defined interface. The Functional Mockup Interface (FMI) defines this interface and the models that implemented it are called Functional Mockup Units (FMUs). While a wealth of specialized simulation tools can generate FMUs, they are often commercial and do not support of complex software prototypes. Rather than implement these as FMUs from scratch (FMI requires expertise in C), losing valuable time, the contribution presented in this paper is a tool that allows FMUs to be implemented rapidly in Python. The advantages of this approach are demonstrated in an industrial use case, where a tracking simulator is implemented as an FMU.

Keywords: Python, prototyping, self-adaptive system, co-simulation, FMI

1 INTRODUCTION

The global competition for intelligent physical products is increasing and as a consequence, the time to market becomes of prime importance. Different disciplines have different traditions for how they can model and analyse the properties that are most important for them. Cyber-Physical Systems (CPSs) are heterogeneous and involve many different disciplines and thus facilitating optimal collaboration between such disciplines is essential (Lee 2008). Here co-simulations can be an effective technique to couple the different models together at the early stages of development, such that it is possible to analyse the overall CPS performance based on the complex interactions between its components (Gomes et al. 2018).

Since the different models typically make use of different branches of mathematics, the interoperability between them is typically ensured using different kinds of standards. Thus each individual model needs to be represented as a stand-alone simulation unit with a standardised interface. In this work we make use of the Functional Mockup Interface (FMI) version 2.0 for co-simulation (FMI v. 2.0 2014) where such units implementing the FMI standard are called Functional Mockup Units (FMUs). Numerous legacy modeling and simulation tools can export their models as FMUs. The contribution presented here complements such tools enabling faster production of FMUs using the Python programming language to support faster collaboration between different disciplines.

SummerSim, July 20-22, 2020, Madrid, Spain; ©2020 Society for Modeling & Simulation International (SCS)

We argue that this prototyping technology is particularly valuable in the early stages of co-simulation (as in agile methodologies) but also for prototyping monitoring and self-adaptive systems (Weyns 2019). The main advantage of our tool is that it does not require compilation, and is platform-independent. Furthermore, Python has a vast ecosystem of numerical libraries, making it the ideal language for prototyping for the aforementioned systems.

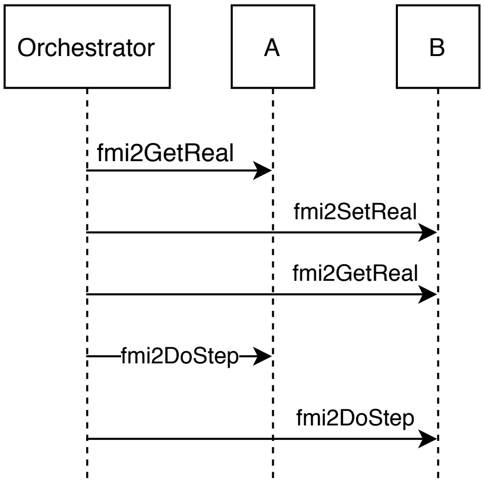
After this introduction, Section 2 provides the necessary background to be able to understand the main contribution of this paper which is presented in Section 3. Afterward, Section 4 presents a case study using the new contribution to produce an online tracking simulator of an unmanned agricultural robot. Finally, Sections 5 and 6 provide an overview of related work and a few concluding remarks about this work respectively.

2 BACKGROUND

2.1 Co-simulation

Co-simulation is a technique that enables the simulation of an entire system by simulating its individual parts (Kübler and Schiehlen 2000). Given the wide range of Modelling and Simulation (M&S) tools, standardized interfaces have been developed to enable these different tools to communicate their simulation results. One such interface is the Functional Mockup Interface (FMI) (Blochwitz et al. 2012).

The FMI defines a C API that binaries exported from each M&S have to implement. Each binary exported from a M&S tool corresponds to a subsystem that has inputs, internal behavior, and outputs. Other software can communicate with such a binary by setting inputs, asking it to compute the internal behavior, and querying its outputs, as illustrated in Figure 1a. To convey this information, FMI standardizes an XML representation of the inputs, outputs and internal structure of the binary. The binary and metadata are packaged in a zip file with the `fmu` extension. The FMI denotes these packages as Functional Mockup Units (FMUs). Figure 1b illustrates the typical structure of an FMU.



(a) Example master algorithm.

```
SineGenerator.fmu/
├── binaries
│   ├── linux64
│   │   └── pyfmu.so
│   └── win64
│       └── pyfmu.dll
├── modelDescription.xml
└── resources
    ├── sinegenerator.py
    └── slave_configuration.json
```

(b) File structure a FMU.

Figure 1: Use of FMI functions and FMU structure.

2.2 Self-Adaptive Systems

Self-adaptation is increasingly recognized as important not just for software systems (Weyns 2019) but also for cyber-physical systems (Zhou et al. 2019, Kritzinger et al. 2018). In its essence, a self-adaptive system can perform well in an environment that is difficult to model at the design time of the system. For example, the case study we introduce in Section 4 is a robot that functions in a wide range of soil types, often different than the ones available for testing at design time.

The implementation of a self-adaptive system comprises techniques such as calibration of models (to enable remote sensing of the system's state), and optimization (to determine the best course of action). We refer to the cited surveys for more details. We argue that prototyping such a system requires access not just to detailed models of the physical dynamics of the system, but also to a high-level language, that provides plenty of numerical libraries. FMI provides easy access to FMUs and therefore to the dynamics of the system, but, to the best of our knowledge, there are not many M&S tools that have a vast library for numerical computing.

Despite the simple interface that the FMI provides (a factor that has contributed to its wide adoption), if one needs to have a custom FMU, one needs to program it in the C language. We argue that C is not the right level of abstraction to prototype self-adaptation processes.

In the next section, we describe the implementation of a generation tool that produces FMUs that are specified in Python, while at the same time implementing the FMI standard. Such FMUs can be changed without requiring recompilation of the binaries, and allow the use of any Python libraries for the computation (provided these are either packaged with the FMU, or available in the execution platform).

3 CONTRIBUTIONS

The contribution of this paper is *PyFMU* (https://github.com/INTO-CPS-Association/pyfmu), a tool which enables rapid development of FMUs using Python. The process of exporting the FMU is fully automated by the tool and requires little knowledge of the FMI Standard and no knowledge of C. The goal of the tool is to enable rapid prototyping of FMUs for a wide range of applications. First, we demonstrate the process of creating an FMU from scratch, in Section 3.1. Following this, we describe the mechanism the tool uses to export and execute the Python code, in Section 3.2.

The FMUs and code used to produce the results can be found as an attachment to the repository's releases: https://github.com/INTO-CPS-Association/pyfmu/releases/tag/0.0.4

3.1 Creating an FMU

A simple sine wave generator is used as an example. The generator has a single output y, and three parameters: amplitude a, frequency ω and phase θ. The output is determined as follows:

$$y = a\sin(\omega t + \theta).$$

The process of creating an FMU using the tool is split into 3 steps, as seen in Figure 2. The first and last steps are carried out by a command-line interface (CLI) supplied with the tool. To create a new project, the sub-command *generate* is used:

```
pyfmu generate -p SineGenerator
```

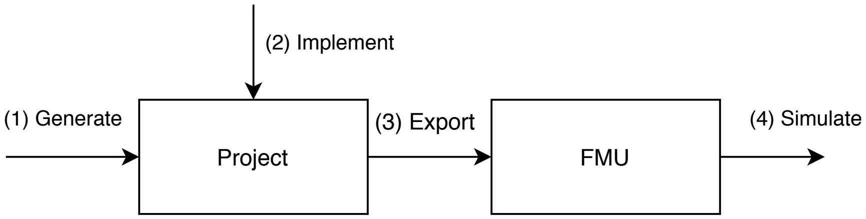


Figure 2: This figure illustrates the typical workflow of when using the tool. The first step is to generate a project. Next, the FMU is implemented in Python. Finally, the project is exported as an FMU, which can then be used as part of a co-simulation.

The command creates a new directory named *SineGenerator* referred to as a *project*. By default, the tool generates the templates of the files needed to start implementation. In this case, the project structure is:

```
SineGenerator/
├── project.json
└── resources
    └── sinegenerator.py
```

The Python script is referred to as the *slave script*. Inside the slave script, the class *SineGenerator* is defined, which is referred to as the *slave class*. When the FMU is instantiated it is this class that acts as its implementation. The purpose of the project.json file is to specify which class to instantiate if multiple scripts are present.

SineGenerator implementation

```python
class SineGenerator(Fmi2Slave):
        def __init__(self):

                author = "Christian Møldrup Legaard"
                modelName = "SineGenerator"
                description = "A single output sine-wave generator"

                super().__init__(
                modelName=modelName,
                author=author,
                description=description)

                self.register_variable('y','real','output')
                self.register_variable('w','real','parameter','fixed',start=1)
                self.register_variable('a','real','parameter','fixed',start=1)
                self.register_variable('p','real','parameter','fixed',start=0)

        def setup_experiment(self, start_time, end_time, tolerance):
                self._start_time = start_time

        def exit_initialization(self):
                self.y = self.a * sin(self.w * self._start_time + self.p)

        def do_step(self, current_time, step_size, no_step_prior):
                self.y = self.a * sin(self.w * current_time + self.p)
```

Figure 3: Typical FMU structure. The highlights show the lines that were added to the generated code.

The tool also generates the initial content of these files, which minimizes the amount of code the user has to write. For reference see Figure 3 which shows a full implementation of the SineGenerator. To implement the FMU, one needs to define its inputs, outputs, and parameters. In FMI standard these entities are generalized as *variables*. To declare a new variable the *register_variable* function is used. For example the output *y*, is defined as: `self.register_variable('y','real','output')`.

The SineGenerator is peculiar, in the sense that it does not take any inputs from other FMUs. Had this not been the case an input *x*, could easily be defined as `self.register('x','real','input',start=0)`. Note that start values must be declared for inputs according to the FMI specification. PyFMU performs validation on the variables to ensure that the model adheres to the FMI specification, even when executed as pure Python code.

Next, the behavior of the FMU must be implemented. This is done by defining special methods in the slave class, corresponding to the methods of the FMI interface. These are invoked whenever the matching function on the FMI interface is called. For example, to define the equivalent of the *fmi2DoStep* function of the FMI standard (recall Figure 1a), the `do_step` method is defined as in Figure 3. The correspondence between the FMI function and the Python counterparts is shown in Figure 4c.

The final stage is to export the project, producing in an FMU with a structure as shown in Figure 1b. Exporting the project as an FMU is done using the *export* command:

```
pyfmu export -p /SineGenerator -o somedir/SineGenerator
```

3.2 Implementation

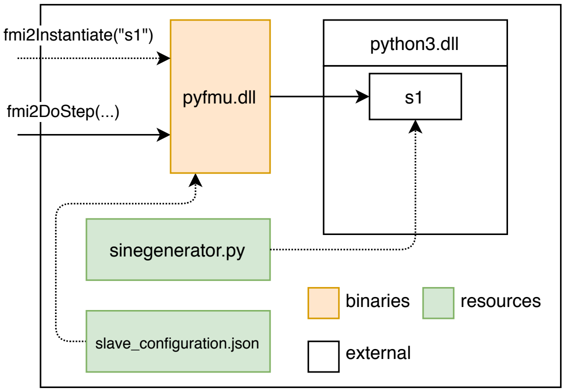
Recall the SineGenerator example, its structure is seen in Figure 1b. PyFMU uses an approach where generic binaries `pyfmu.so/dll` are pre-compiled and used by all exported FMU. It acts as a "wrapper" around the Python code implementing the FMU. Whenever a call is made to FMI interface the call is passed to a Python interpreter which is running inside the FMU, as illustrated in Figure 4a. The tool ships with pre-compiled binaries for several platforms, ensuring that an FMU produced on one platform, will also run on the others. The main advantage of this approach is that it eliminates the need for the user to ever deal with FMU compilation issues.

The Python interpreter is included in the binary using an approach referred to as *embedding*. That is, the Python interpreter is embedded into the C/C++ program, by including its headers and linking the library. This makes it possible to programmatically interact with Python, for example by importing modules, instantiating objects and calling functions. Figure 4b shows how the *_doStep* method is called on a Python object from C.

The same mechanism is used for creating instances of the slave class within the Python interpreter. To instantiate the objects within the interpreter the binary reads the name of the slave script and class from the *slave_configuration.json* file. The process of generating the *modelDescription.xml* file is fully automated by the tool. Rather than going through the trouble of reading and parsing the script as a text file, the tool simply creates an instance of the slave class. The tool then inspects the declared variables in an object-oriented fashion and creates the appropriate XML elements in the model description. This approach has the added benefit that the full flexibility of Python can be used to generate complex FMU interfaces with ease. For example, Figure 5 shows how the SineGenerator may be modified to add multiple outputs.

4 CASE STUDY: PROTOTYPING A TRACKING SIMULATOR

In the process of moving towards unmanned agricultural operations, novel monitoring systems are needed to account for unforeseen events occurring during field operations. For example, the soil surface characteristics may vary locally based on topography, soil composition, and moisture content. The variations can reduce the



(a) Illustration of the process used to instantiate Python slave inside the FMU and how subsequent calls are propagated to the slave. When fmi2Instantiate is called for the first time the binary starts a Python interpreter. Following this, the binary reads the name of the script and which class to instantiate from the slave_configuration.json file. This information is used to create an instance of the class inside the interpreter. After this calls to the FMI component are propagated to the newly instantiated Python object.

```c
#include <Python.h>
...
PyObject* f = PyObject_CallMethod(
    pInstance_,
    "_do_step",
    "(ddO)",
    currentTime,
    stepSize,
    noSetPrior);
```

(b) Invoking the slave's doStep from the wrapper.

fmi2Functions.h	SineGenerator
fmi2DoStep	do_step
fmi2SetupExperiments	setup_experiments
fmi2ExitInitializationMode	exit_initialization

(c) Examples of FMI function mapping to the corresponding slave functions.

Figure 4: Mechanism used to wrap execute Python code within the FMUs.

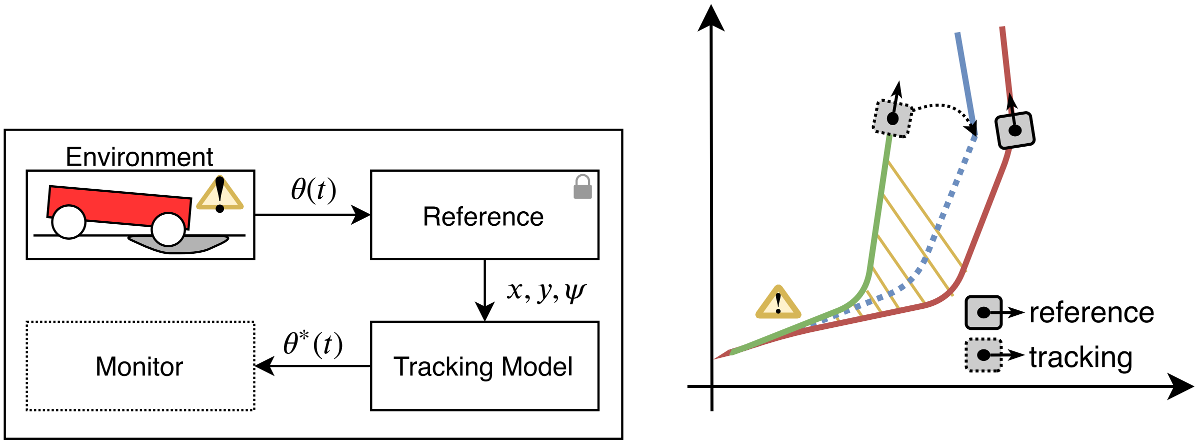
```python
def __init__(self):
    ...
    n_outputs = 3
    for i in range(n_outputs):
        self.register_variable(f'y{i}','real','output')
```

Figure 5: Programmatic variable declaration.

traction of the wheels and consequently affect motion and maneuverability. With better monitoring systems, the vehicle control system can adapt the steering despite these variations. In this section, we describe how PyFMU allowed us to leverage the full flexibility of Python to quickly prototype a tracking simulator of an agricultural autonomous vehicle, called Robotti. Initial modeling efforts of the system were described in (Foldager et al. 2018).

The goal of the tracking simulator is to detect changes in the reference system's dynamics caused by changes in the environment, during a single simulation, as shown in Figure 6a. Asides from detecting that a change has happened, the tracking simulator attempts to find a new set of parameters θ^* which would explain the observed change in behavior. This is made possible by the fact that the tracking simulator itself contains a simplified dynamical model of the Robotti, referred to as the *tracking model*. During co-simulation, both models are run in parallel producing two separate trajectories as shown in Figure 6b.

Undisturbed, the two trajectories will evolve in an almost similar fashion. However, the moment that the reference system parameters change the two trajectories will start to diverge and the tracking simulator will trigger a re-calibration. We first give a brief description of Robotti and how it is modeled in this case study. Then we describe the implementation of the tracking simulator using PyFMU.



(a) Co-simulation scenario of tracking simulator. Each box corresponds to an FMU and the arrows identify connections between them. The reference represents a black-box model of the physical system, which is influenced by a set of parameters controlled by the environment θ, which vary over time. The goal of the tracking model is to match the behavior of the reference as close as possible. It does so by finding the set of parameters θ^* which results in an trajectory as close as possible to the reference system.

(b) Illustration of the online calibration process. The reference and tracking models share the same trajectory until an event occurs which causes a change in the reference model. After some time the discrepancy between the two models has grown sufficiently large to trigger a recalibration. The recalibration process finds the parameters θ^* which results in the trajectory closest to the reference (dotted blue line), within a bounded interval of time (e.g., since the discrepancy occurred). Following this the tracking model resumes form the end of the optimal trajectory (solid blue line).

Figure 6: Tracking simulator overview.

4.1 Robotti

The agricultural robotic vehicle is a four-wheeled Ackermann steered autonomous system. The vehicle is designed as a generic platform applicable for various agricultural operations such as weeding, spraying, or cultivation. The navigation and steering control are performed on-board on a ROS-based system and is equipped with various sensors such as RTK-GPS, LiDAR, IMU and encoders. A photo of Robotti is shown in Figure 8.

For the tracking simulator, we use a bicycle dynamic model adapted from (Kong et al. 2015, Rajamani 2011), and given as follows. We assume the longitudinal velocity $\dot{x}$ is constant and model the lateral dynamics. The lateral and rotational acceleration were obtained by summing up the forces and moments at the center of the vehicle. The orientation and position were obtained by numerically integrating Equations (8) and (9) twice. F_{c_f} is the sum of the local tire forces on the front and rear wheels calculated by the product of the slip angle α and the tire stiffness coefficient. The slip angles are calculated as a function of the orientation of the vehicle ψ, the velocity components $\dot{x}$ and $\dot{y}$ and the steering angle on each of the front wheels $\delta_{l/r}$ where subscripts indicate left and right. l_f and l_r are the distances between the center of gravity of the vehicle and the front and rear wheels in the longitudinal direction as shown in Figure 7. The equations of the model are summarized in Equations (1) to (9), and parameters and their order of magnitude are shown in Table 1

Front tire slip angle $$\alpha_f = \delta_f - (\dot{y} + l_f \dot{\psi})/\dot{x} \tag{1}$$

Rear tire slip angle $$\alpha_r = (\dot{y} - l_r \dot{\psi})/\dot{x} \tag{2}$$

Lateral tire force at a front wheel $$F_{c_f} = C_{\alpha_f} \alpha_f \tag{3}$$

Lateral tire force at a rear wheel $$F_{c_r} = C_{\alpha_r}(-\alpha_r) \tag{4}$$

Longitudinal acceleration $$\ddot{x} = \dot{\psi}\dot{y} + a \tag{5}$$

Lateral acceleration $$\ddot{y} = -\dot{\psi}\dot{x} + (1/m)(F_{c_f}\cos(\delta_f) + F_{c_r}) \tag{6}$$

Yaw acceleration $$\ddot{\psi} = (1/I_{zz})(l_f F_{c_f} - l_r F_{c_r}) \tag{7}$$

Velocity in the global frame $$\dot{X} = \dot{x}\cos(\psi) - \dot{y}\sin(\psi) \tag{8}$$

Velocity in the global frame $$\dot{Y} = \dot{x}\sin(\psi) + \dot{y}\cos(\psi) \tag{9}$$

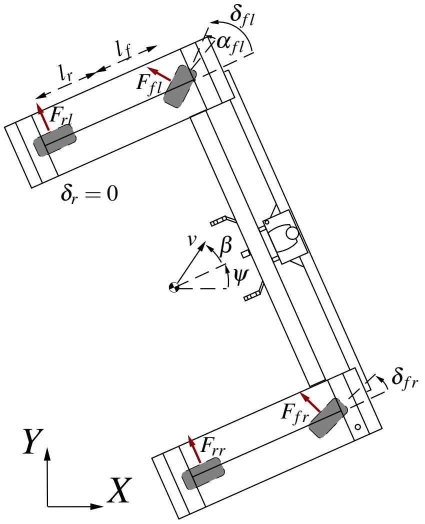


Figure 7: Sketch of the Robotti

Figure 8: Robotti in the field. Photo: Agrointelli.

Parameter	Magnitude	Description
l_f	10^0	Distance COG to front wheel (m).
m	10^3	Mass of the vehicle.
I_{zz}	10^3	Rotational inertia.
$C_{\alpha_{f/r}}$	10^2	Tire cornering stiffness.

Table 1: Parameter and their magnitudes (specific values are omitted to protect company property)

For simplicity, the surface-tire interaction was modeled by a linear relationship between the tire cornering stiffness and the tire slip angle. In the actual model of the Robotti, this interaction is modeled by non-linear relationships that include the surface friction coefficient, the tire normal load, and the steering angle. Moreover, the dynamics of Robotti were derived for a four-wheel vehicle as schematically shown in Figure 7. The actual Robotti model was implemented using the 20-sim tool (Broenink 1997, Kleijn 2009) and exported as an FMU.

4.2 Results And Discussion

The purpose of the prototyped tracking simulator is to show when the model proposed in Equations (1) to (9) fails to accurately represent reality. This is achieved by constantly monitoring the behavior of the system.

When discrepancies exceed a tolerance value, a new calibration is started, that tries to find new parameters for the model that explain the measured data (recall Figure 6b).

Our tracking simulator was prototyped in Python and exposed as an FMU through PyFMU. To produce the simulation results, the tracking simulator uses the model in Equations (1) to (9) and the Python numerical library SciPy (Eric Jones and Travis Oliphant and Pearu Peterson and others 2001). The actual data comes as inputs to the PyFMU, and the recalibration process is implemented using the SciPy library.

To validate the tracking simulator, we used a co-simulation with the FMU produced from the actual model of the Robotti, created in 20-sim. To more easily trigger the recalibration process, we change the C_{α_f} parameter of the actual Robotti FMU during the co-simulation. These results are explained in Figure 10.

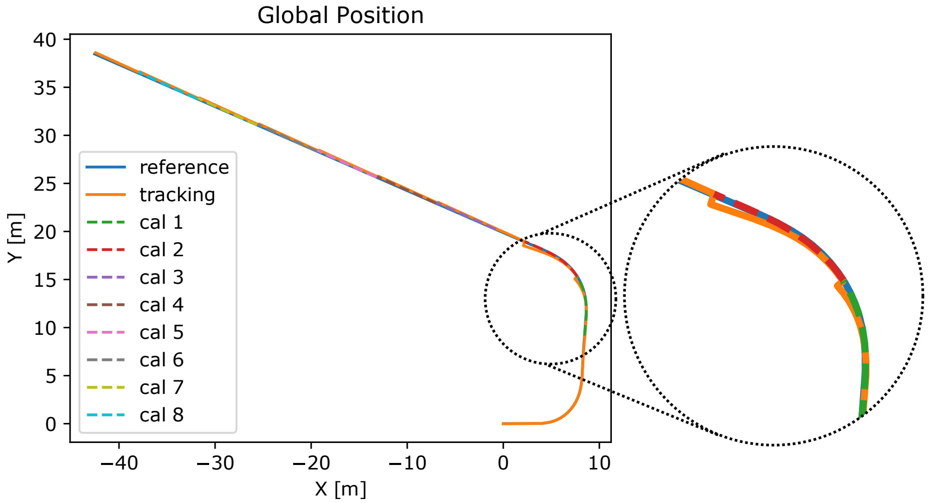


Figure 9: Global position of the robot. The orange line represents the simulation while the blue line represents the values of the actual Robotti FMU. Until the first bend, the simulation matches the actual Robotti FMU. On the second bend, however, there is divergence. This triggers the first re-calibration. The dashed line represents the interval of actual data used to find a new C_{α_f} parameter. Interleaved by a brief cool-down period, there are two subsequent re-calibrations, as the error is still above the tolerance.

Detecting the local surface characteristics during operation is a problem involving the surface (e.g., soil type and water content), the tires, and the vehicle dynamical system prescribing the traction and maneuverability. The soil-tire interaction is typically modeled using classical terramechanics models (Wong et al. 1984) which is a semi-empirical model that is often used as a basis for modeling off-road applications. In this work, we use a bike model to track the motion of a more complex vehicle model in an FMU.

The design of a tracking simulator requires advanced domain knowledge and a careful choice of the parameters used in the recalibration process. The recalibration process problem is no different than a model identification problem. What makes the design of a tracking simulator unique is that each recalibration process is a slightly different model identification problem. For instance, the first re-calibration may succeed, while the second one may not. Developing a tracking simulator that works every time is outside the scope of this paper and the subject of ongoing work.

These challenges are better understood by prototyping and experimentation activities, which are enabled by tools such as PyFMU. This has allowed us to grasp some of the trade-offs between the different parameters that go into the design of a tracking simulator. For instance, a larger re-calibration interval is not always better, as it may lead to diverging optimization.

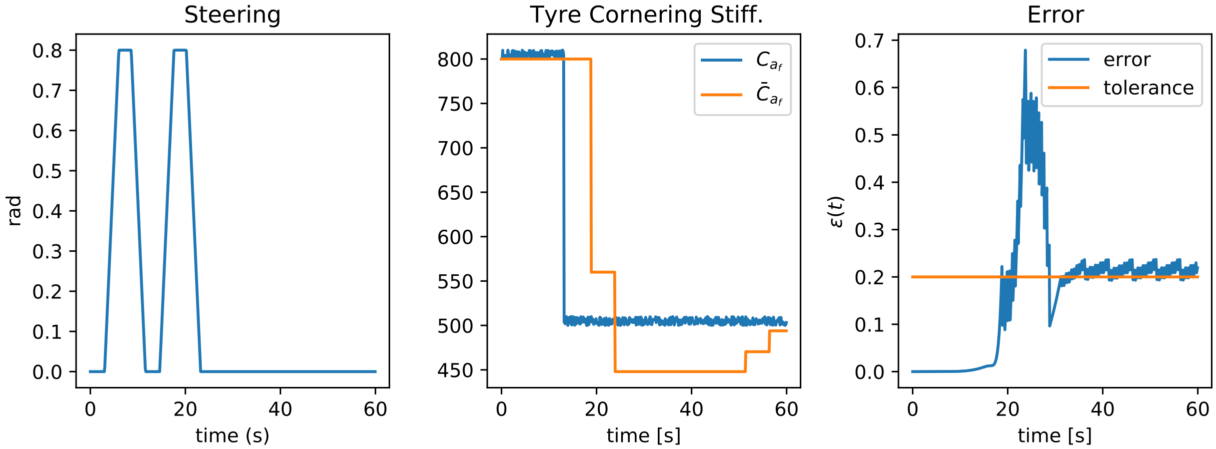


Figure 10: Steering angle, tyre stiffness and error of reference and adaptive model.

5 RELATED WORK

We restrict our scope to approaches to the implementation of FMUs for co-simulation. We therefore exclude other co-simulation interfaces (e.g., HLA (IEEE 2014)). Additionally, we consider only general approaches that are not restricted to modelling a specific domain such as fluid dynamics or power systems. To the authors knowledge no survey exists that provides a comprehensive comparison of tools capable of producing FMUs. Rather than attempt this, several works are presented highlighting the two main approaches which are used for exporting FMUs.

The first approach of generating FMUs is using a compiled language such as C/C++. Aslan et al. (2015) describes an object-oriented framework for the implementation of FMUs for co-simulation. New FMUs are developed by providing C++ code that extends a given base class. There is the possibility that a user specifies the model equations in C++, and the code provides some standard solvers to solve it. FMI (2015) proposes a similar object-oriented framework but supports only FMI1. The strength of this approach is the efficiency of the model and portability once compiled for a given platform.

The alternative to compiling FMUs is the approach used by PyFMU referred to as “tool-coupling” by Widl and Müller (2017). Thule et al. (2018) uses this approach for interfacing with models written in the language VDM-RT (Verhoef et al. 2006). Gomes et al. (2018) describes a method and a tool for the modification of existing FMUs, by wrapping them into new FMUs, according to the needs of specialized master algorithms. The new FMUs need to be recompiled, and may modify the inputs provided (e.g., implementing different input approximation functions), the outputs requested, or the way time stepping is performed.

More related to the tool is PythonFMU (https://github.com/NTNU-IHB/PythonFMU) which we originally planned to use. However, at the time we encountered issues with the generated model description files. This, among other limitations, spawned the work on PyFMU as a separate project. The authors of PythonFMU has since fixed several of these issues.

6 CONCLUDING REMARKS

Being able to rapidly prototype and simulate different designs in the early design stages of CPSs is extremely valuable, especially in the context of self-adaptive system development. The PyFMU tool makes it possible to implement FMUs with ease, using the popular Python programming language and its extensive set of numerical libraries.

The tracking simulator implemented in the case study demonstrates the advantages of this approach; as it would have been very time consuming to implement and refine the optimization algorithm using an existing approach. This is because the convergence of the recalibration process of a tracking simulator depends on when it started, which portion of the real trajectory is taken into account, and on several other parameters of the optimization engine. Having a prototype enables developers to more easily understand the different parameters involved and ultimately fine-tune the system. For instance, this prototype allowed us to conclude that a larger re-calibration interval is not always better, as it may fail to converge.

Another important perspective is that the content of the PyFMU is *pure* Python code. This means it can be developed, tested, and debugged, independently of the FMU where it resides. The ability to verify changes to the models quickly using a debugger was a big advantage to us.

Our tool opens the possibility of co-simulation to the many people who know Python but do not have sufficient knowledge of FMI or C to implement one from scratch. The tool currently supports the core functionality of FMI, however, some optional features such as providing the derivatives of the FMU and declaring physical units are subject of ongoing work. An continuously updated list of supported features and capabilities can be found on the front page of the code repository.

REFERENCES

2015. “FMI++”. https://sourceforge.net/projects/fmipp/.

Aslan, M., U. Durak, and K. Taylan. 2015, July. “MOKA: An Object-Oriented Framework for FMI Co-Simulation”. In *Conference on Summer Computer Simulation*, pp. 1–8. Chicago, Illinois, Society for Computer Simulation International San Diego, CA, USA.

Blochwitz, T., M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012, November. “Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models”. In *9th International Modelica Conference*, pp. 173–184. Munich, Germany, Linköping University Electronic Press.

Broenink, J. F. 1997. “Modelling, Simulation and Analysis with 20-Sim”. *Journal A Special Issue CACSD* vol. 38 (3), pp. 22–25.

FMI v. 2.0 2014. “Functional Mock-up Interface for Model Exchange and Co-Simulation”.

Foldager, F., O. Balling, C. Gamble, P. G. Larsen, M. Boel, and O. Green. 2018, July. “Design Space Exploration in the Development of Agricultural Robots”. In *AgEng conference*. Wageningen, The Netherlands.

Gomes, C., B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere. 2018. “Semantic Adaptation for FMI Co-Simulation with Hierarchical Simulators”. *SIMULATION* vol. 95 (3), pp. 1–29.

Gomes, C., C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. 2018. “Co-Simulation: A Survey”. *ACM Computing Surveys* vol. 51 (3), pp. Article 49.

IEEE 2014. “IEEE 1516 High Level Architecture.”.

Eric Jones and Travis Oliphant and Pearu Peterson and others 2001. “SciPy: Open source scientific tools for Python”. [Online; accessed 31 March 2020].

Kleijn, C. 2009. *20-Sim 4.1 Reference Manual*. Getting Started with 20-sim.

Kong, J., M. Pfeiffer, G. Schildbach, and F. Borrelli. 2015, June. “Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design”. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1094–1099. Seoul, South Korea, IEEE.

Kritzinger, W., M. Karner, G. Traar, J. Henjes, and W. Sihn. 2018. “Digital Twin in Manufacturing: A Categorical Literature Review and Classification”. *IFAC-PapersOnLine* vol. 51 (11), pp. 1016–1022.

Kübler, R., and W. Schiehlen. 2000. “Two Methods of Simulator Coupling”. *Mathematical and Computer Modelling of Dynamical Systems* vol. 6 (2), pp. 93–113.

Lee, E. A. 2008. “Cyber Physical Systems: Design Challenges”. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369.

Rajamani, R. 2011. *Vehicle Dynamics and Control*. Springer Science & Business Media.

Thule, C., K. Lausdahl, and P. G. Larsen. 2018, July. “Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs”. In *The 16th Overture Workshop*, edited by K. Pierce and M. Verhoef, pp. 23–38. Oxford, Newcastle University, School of Computing. TR-1524.

Verhoef, M., P. G. Larsen, and J. Hooman. 2006. “Modeling and Validating Distributed Embedded Real-Time Systems with VDM++”. In *FM 2006: Formal Methods*, edited by J. Misra, T. Nipkow, and E. Sekerinski, Lecture Notes in Computer Science 4085, pp. 147–162, Springer-Verlag.

Weyns, D. 2019. “Software Engineering of Self-Adaptive Systems”. In *Handbook of Software Engineering*, edited by S. Cha, R. N. Taylor, and K. Kang, pp. 399–443. Cham, Springer International Publishing.

Widl, E., and W. Müller. 2017, July. “Generic FMI-Compliant Simulation Tool Coupling”. In *The 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, pp. 321–327.

Wong, J. Y., M. Garber, and J. Preston-Thomas. 1984. “Theoretical Prediction and Experimental Substantiation of the Ground Pressure Distribution and Tractive Performance of Tracked Vehicles”. *Proceedings of the Institution of Mechanical Engineers, Part D: Transport Engineering* vol. 198 (4), pp. 265–285.

Zhou, P., D. Zuo, K. Hou, Z. Zhang, J. Dong, J. Li, and H. Zhou. 2019, February. “A Comprehensive Technological Survey on the Dependable Self-Management CPS: From Self-Adaptive Architecture to Self-Management Strategies”. *Sensors* vol. 19 (5), pp. 1033.

ACKNOWLEDGEMENTS

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. We also acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS co-simulation and the initial modeling of Robotti. Finally, we thank Daniella Tola for proofreading drafts of this manuscript.

AUTHOR BIOGRAPHIES

CHRISTIAN MØLDRUP LEGAARD is a PhD student at the Department of Engineering at Aarhus University. His research is centered on a combination of co-simulation and machine learning. His email address is cml@eng.au.dk.

CLÁUDIO GOMES is a Post-Doc at the Department of Engineering at Aarhus University. His research is centered on co-simulation and digital twins. His email address is claudio.gomes@eng.au.dk.

FREDERIK FORCHHAMMER FOLDAGER is an industrial PhD student at the Department of Engineering at Aarhus University and he is employed at Agrointelli. His research is centered on modeling of soil–machine interaction. His email address is ffo@agrointelli.com.

PETER GORM LARSEN is a Full Professor at the Department of Engineering at Aarhus University. His research is centered on digital twins, co-simulation and tool building. His email address is pgl@eng.au.dk.