A Universal Mechanism for Implementing Functional Mock-up Units

Christian Møldrup Legaard[1], Daniella Tola[1], Thomas Schranz[2],
Hugo Daniel Macedo[1] and Peter Gorm Larsen[1]
[1]*DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Aarhus, Denmark*
[2]*Graz University of Technology, Graz, Austria*

Keywords: Co-simulation, Functional Mock-up Interface, Functional Mock-up Unit, Tool.

Abstract: Producing independent simulation units that can be used in a Functional Mock-Up Interface (FMI) setting is challenging. In some cases, a modelling tool may be available that provides the exact capabilities needed by exporting such units. However, there may be cases where existing tools are not suitable, or the cost is prohibitive, thus it may be necessary to implement a Functional Mock-up Unit (FMU) from scratch. Correctly implementing an FMU from scratch requires a deep technical understanding of the FMI specification and the technologies it is built upon. A consequence of FMI being a C-based standard is that an FMU must, generally, be implemented in C or a compiled language that offers a binary-compatible with C such as C++, Rust, or Fortran. In this paper we present UniFMU, a tool that makes it possible to implement FMUs in any language, by writing an adapter that can be plugged in to our modular approach. UniFMU also provides both a graphical user interface and command-line interface feature for generating new FMUs from a selection of programming languages. We expect our tool and approach to be useful for the simulation community both when porting simulators written in languages without FMI support, and when implementing or re-implementing such support.

1 INTRODUCTION

When modelling Cyber-Physical Systems (CPSs), it is advantageous to model different parts using different formalisms and tools and then combine the different models as simulation units using co-simulation (Gomes et al., 2018). One of the most popular standards for co-simulation is called the Functional Mock-up Interface (FMI) (Modelica Association, 2019), which defines how different simulators are coupled and a format for packaging simulation units. A co-simulation combines a number of such packaged simulation units, termed Functional Mock-up Units (FMUs), using a master algorithm that combines the simulation of each of the independent simulation units (Thule et al., 2019a; Thule et al., 2019b) into a joint simulation of the system.

A common way to obtain FMUs is to use FMI-enabled modelling tools such as OMEdit (Asghar and Tariq, 2010), Simulink (Simulink09, 2009) or 20-sim (Controllab Products B.V., 2013) to create models interactively using a GUI, which can subsequently be exported as FMUs. While existing tools may cover the needs of most modelling applications, the need for specialized FMUs that can only be implemented by hand will frequently arise. Unfortunately, the process of creating an FMU from scratch is cumbersome and difficult as it requires:

- A deep understanding of the FMI specification.
- The code to be implemented in a C-compatible language.
- Cross-compilation to support multiple platforms.
- Manual creation and synchronization of the modelDescription.xml file.
- Correct packaging of assets as an FMU archive.

Due to the many pitfalls of this process, it is impractical for anyone but experts to produce FMUs by hand.

This paper presents an extendable tool called *Universal Functional Mock-up Unit* (UniFMU) that facilitates the implementation of FMUs in any programming language. Specifically, our contribution is a tool that provides:

- Support for Python, C# and Java FMUs out of the box.

Legaard, C., Tola, D., Schranz, T., Macedo, H. and Larsen, P.
A Universal Mechanism for Implementing Functional Mock-up Units.
DOI: 10.5220/0010577601210129
In *Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2021)*, pages 121-129
ISBN: 978-989-758-528-9

- An easy to use extension mechanism to provide support for any language.
- CLI to generate template FMUs using a single command.
- Pre-built binaries for Windows, Linux and macOS, eliminating the need for cross-compilation and complex tool-chain setup.
- Flexible configuration of the execution environment, such as running inside a Docker container or activating a virtual environment.
- A work in progress GUI for modifying FMU's modelDescription.xml files, eliminating the need to modify by hand.

The generated FMUs are fully FMI compliant meaning they can be used in any FMI enabled co-simulation tool, without making any modifications. The tool is freely available and can be accessed in the GitHub repository: *https://github.com/INTO-CPS-Association/unifmu/*.

We expect our tool and approach to be useful for the simulation community. As we describe in Section 5, by implementing one of two available protocols or using one of our language backends a user porting simulators written in languages without FMI support can focus on implementing the FMI standard functionality in his favourite language. Our work also lowers the complexity required to modeling tool providers interested in implementing or improve tools with FMI export capabilities. With our work the tool can rely on a modular approach where UniFMU deals with the C API and the export implementation can focus on providing the model semantics.

In the following, we provide a brief introduction to related work in Section 2. Then, Section 3 provides an introduction to FMI with an emphasis on the implementation of an FMU from scratch. Next, Section 4 demonstrates the alternative implementation facilitated by UniFMU. Next, Section 5 provides the details of our approach that may be useful to adopters interested to generate FMUs using our tool. Following this Section 6 describes how UniFMU executes the authors code inside the FMU as well as how the tool can be extended to support a new language. Section 7 then describes how the generated FMUs can be ran inside a docker container. Finally, Section 8 provide a few concluding remarks including the planned future work.

2 RELATED WORK

The difficulty of authoring FMUs by hand has led to the development of several tools that support the authoring of FMUs. Typically, each tool focuses on supporting a specific language and implements its own workflow. A selection of this type of tool is seen in Table 1.

Table 1: Overview of tools that can: (i) import and simulate FMUs, (ii) export models as FMUs.

Name	Language	Import	Export
FMUSDK	C	x	x
FMI++	C/C++	x	x
FMI4j	Java	x	x
JavaFMI	Java	x	x
JFMI	Java	x	
PythonFMU	Python		x
PyFMU	Python		x
OvertureFMU	VDM		x

A drawback of using language specific tools is that it requires the user to install and learn how to use several different tools. Another issue is that these tools tend to cover only the most popular languages and/or languages where interoperability with C is easy to implement.

In addition to tools that allow authoring of FMUs, some simulation tools allow user written code to be mixed with the code implemented by FMUs. For example (fmp, 2021; Widl et al., 2013) allow FMUs to be imported and simulated in Python. This makes it possible to insert additional Python code in the simulation loop, essentially implementing a *virtual FMU* without the hassle of packaging. Clearly, this approach for mixing in code is rather limited in terms of reuse and the inability to mix different languages in the simulation. Our tool provides a generic way to generate FMUs that can be used in any FMI enabled simulation tool.

More relevant are works that provide a separation between the FMI C-API and the implementation of the FMU such *Proxy-FMU* developed in (Hatledal et al., 2019). Here, the authors propose using *remote procedure calling* (RPC) to allow FMUs to run in a distributed setting, while at the same time removing restrictions on the FMUs implementation language. A further development by the same authors is the *fmu-proxify* tool that allows existing FMUs to be wrapped in a way such that it enables distributed co-simulation in any FMI enabled simulation tool.

Similar to Proxy-FMU our tool also uses RPC as a mechanism to enable the execution of arbitrary code inside an FMU. The main contribution of UniFMU is that the tool ships with support for several languages that for which template FMUs can quickly be generated using the CLI. Another distinction is that our tool makes it easy to run additional commands like selecting a specific virtual environment to run the FMU inside or deploying it in a docker container as described in section 7.

3 CREATING AN FMU IN C

One of the most difficult and time consuming aspects of FMI based co-simulation is implementing models and packaging them as FMUs. In many cases, free or commercial tools are available that automate the conversion of the model into an FMU. However, in practice there are situations where no tool covering your particular needs exists. For instance consider use cases where an FMU would need to:

- Capture input from a operator through a GUI.
- Interact with remote hardware.
- Integrate data-driven components such as neural-networks.

In such cases it may be necessary to implement the FMU from scratch in C. As a running example we introduce a simple *Adder*-FMU that takes the sum of its two inputs to form its output, as shown in fig. 1. We chose this example to illustrate the effort required to implement even the simplest FMU from scratch. Additionally, we omit details of how to implement many of the specialized FMI methods, that require careful considerations to memory management. Hopefully, this will be enough to convince the reader that creating a more complex FMU from scratch would be even more challenging and time consuming.

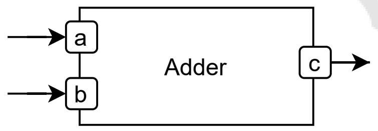


Figure 1: Adder FMU computes c = a + b.

To understand the challenges of generating an FMU we first examine the its structure. In plain terms an FMU is a zip-archive containing a collection of files that together define the interface and behavior of a model. A minimal example of how the contents of the adder FMU zip folder may look like is depicted in fig. 2.

The folder must contain the functional behavior of an FMU, which is realized by a shared library (unifmu.so) located in the `binaries` directory (one for each operating system), and a configuration file, the `modelDescription.xml`, stored in the root of the FMU that provides metadata about the unit. At runtime, the master algorithm dynamically links the binary for the specific platform allowing the master to invoke the appropriate methods to get and set

```
adder_c.fmu
└─binaries
│ ├─linux64
│ │ └─adder.so
│ ├─windows64
│ │ └─adder.dll
│ └─darwin64
│   └─adder.dylib
└─modelDescription.xml
```

Figure 2: Directory structure of adder's C implementation.

variables of the model through the *C-API*. To obtain this library several methods declared in the FMI's C-headers must be implemented in a C compatible language and compiled separately for every platform that the FMU is expected to be run on. A small selection of these is seen in listing 1.

```c
typedef struct
{
 fmi2Real a;
 fmi2Real b;
 fmi2Real c;
} Adder;

void *fmi2Instantiate(const char *
    name,
    ...)
{
 Adder *instance = malloc(...);
 return instance;
}

int fmi2DoStep(void *c, ...)
{
 Adder *fmu = c;
 fmu->c = fmu->a + fmu->b;
 return fmi2OK;
}
```

Listing 1: Implementation of Adder in C.

The code shown in the snippet represents a simplified implementation of the many details and functions that must be implemented, and in addition to it we compile a binary for each architecture including the FMI standard headers as shown in fig. 3. In addition to the sheer number of functions that have to be implemented, low-level programming considerations of memory management and ownership of strings makes it difficult to implement the functions correctly. This has been described as one of the challenges of creating FMUs, as the documentation of the FMI standard is insufficient (Schweiger et al., 2019; Bertsch et al., 2014).

The binaries provides the functional behavior of the model, the `modelDescription.xml` declares the

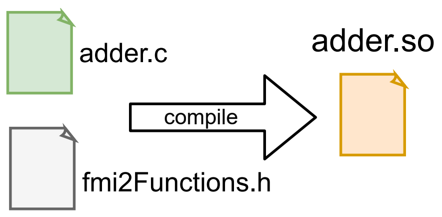


Figure 3: FMI headers and their implementation compiled into a shared library.

interface and capabilities of the model. Among other things, this file declares inputs and outputs of the model, their type and their default value. This information is used by the master algorithm to connect pairs of inputs and outputs of models during the configuration of a co-simulation. It should be stressed that in the general case the binary itself is oblivious to the contents of the `modelDescription.xml` file. As such special care should be taken to ensure that the variables declared in the description are consistent with the implementation. For example if one of the inputs to the adder is declared as an integer in the description rather than a floating point number, the binary would still treat it as a floating point number leading to an incorrect output value. We discuss the issue of ensuring consistency of the `modelDescription.xml` further in section 8.

4 CREATING AN FMU IN UniFMU

UniFMU provides a *Command Line Interface* (CLI) that can be used to author FMUs in several popular languages such as *Python*, *C#* and *Java* as shown in Table 2, and we plan to expand the list of supported languages in the future. We distinguish between a backend and a language, since one language can implement multiple backends. We define a backend as the method of communication between the UniFMU wrapper and the FMU. This is described in more detail in Section 5.

Table 2: List of supported languages and backends, * denotes default backend.

Language	Backends
Python	gRPC*, ZeroMQ
C#	gRPC
Java	gRPC

The CLI is implemented in Python, but it should be stressed that the generated FMUs do not depend on Python during simulation (except for FMUs implemented in Python). The tool can be installed using, *pip*, the de-facto package manager for Python, using a single command:

```
pip install unifmu
```

The package manager installs the CLI as well as any resources needed during the generation and runtime of the FMUs. Alternatively, the tool can be installed from source using the instructions found in the associated GitHub repository[1].

To generate an FMU you invoke the program with the sub-command *generate* with arguments specifying the language and name of the exported FMU. For example to generate an FMU named `python_adder.fmu` in Python the following command can be used:

```
unifmu generate python python_adder.fmu
```

Executing this command creates an FMU with the file structure shown in fig. 4. The generated FMU is fully functional and serves as a template that can be modified to implement the desired behavior for the model.

```
adder.fmu
├── binaries
│   ├── darwin64
│   ├── linux64
│   │   └── unifmu.so
│   └── windows64
├── resources
│   ├── model.py[i]
│   ├── backend_grpc.py[b]
│   ├── fmi2.py[b]
│   └── launch.toml
└── modelDescription.xml
```

Figure 4: Python FMU directory tree. [b] denotes backend, [i] denotes implementation.

An important difference between an FMU implemented in C and one implemented using UniFMU is that the behavior of the model is not defined by the binaries, but rather by the file(s) stored in the `resources` folder. This makes it possible to reuse the same binaries for all FMUs, independently of the language that they are implemented in. These binaries are pre-compiled and shipped with the tool for Linux, Windows and macOS. Two benefits of this is that the FMU can run on all platforms without any additional effort from the author and that it is not necessary to install a compiler tool-chain.

The `model.py` file shown in fig. 4 implements the functionality of the FMU. Inspecting the code inside of the `model.py` FMU, the most relevant function is `do_step` shown in listing 2.

[1]https://github.com/INTO-CPS-Association/unifmu

```python
class Model(Fmi2FMU):
    def __init__(self):
        self.a = 0.0
        self.b = 0.0
        self._update_outputs()

    def _update_outputs(self):
        self.c = self.a + self.b

    def exit_initialization_mode(
   self):
        self._update_outputs()
        return Fmi2Status.ok

    def do_step(self, ...):
        self._update_outputs()
        return Fmi2Status.ok
```

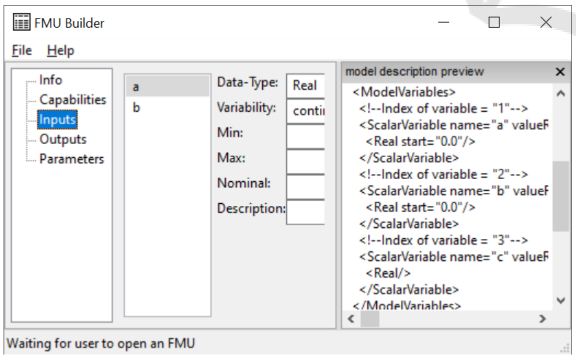
Listing 2: model.py implementation.

It contains the addition operation functionality provided by the FMU, and the variables `a` and `b` are the inputs to the FMU, and the variable `c` is an output variable containing the result of the addition of the two inputs. These variables and their types are all defined in the `modelDescription.xml` file.

In addition to the CLI, UniFMU can be launched as a GUI, seen in fig. 5, using the subcommand:

```
unifmu gui
```

The GUI is currently work in progress, and can so far be used to access the same functionality as the CLI. The vision is to extend this GUI to be able to declare and modify the variables corresponding to the contents of the modelDescription.xml.



Data-Type: Real
Variability: conti
Min:
Max:
Nominal:
Description:

```
<ModelVariables>
  <!--Index of variable = "1"-->
  <ScalarVariable name="a" valueR
    <Real start="0.0"/>
  </ScalarVariable>
  <!--Index of variable = "2"-->
  <ScalarVariable name="b" valueR
    <Real start="0.0"/>
  </ScalarVariable>
  <!--Index of variable = "3"-->
  <ScalarVariable name="c" valueR
    <Real/>
  </ScalarVariable>
</ModelVariables>
```

Figure 5: GUI for UniFMU.

5 HOW DOES IT WORK?

The main motivation behind UniFMU is to allow arbitrary code written in any language to be executed within an FMU. The mechanism used by the tool is to provide a generic binary that spawns a separate process for each instantiated slave during runtime. Unlike the binary, the spawned processes aren't restricted to the narrow set of compiled languages that are conventionally used to implement FMUs. This opens the possibility for using interpreted languages or languages that rely on a garbage collector to manage memory.

In order to forward the FMI calls from the master algorithm to the concrete implementation provided *slave processes* a remote procedure call (RPC) based on *gRPC*[2] is used to pass commands from the binary to the slave process. Seen from the perspective of the master algorithm this additional layer of indirection is totally opaque, meaning that FMUs produced by the tool can be used in any FMI compliant simulation tool.

Comparing this with the conventional approach, we add an extra layer between the master algorithm and the FMU itself. The comparison between the conventional and UniFMU approach is illustrated in fig. 6. In both approaches the master algorithm communicates with the binary files through the C API that implements the FMI standard. The difference is that the UniFMU wrapper "translates" these C API calls to messages that can be exchanged through a RPC, such as gRPC or ZeroMQ. Only one of these two backends is necessary to implement when supporting a new language. The specific backend used is defined in the configuration file of the FMU, as shown in the launch.toml file in listing 3. In addition to the backend, the launch.toml file specifies the command used to start the backend process. It is possible to specify different commands based on different OSs, since they may require a different setup.

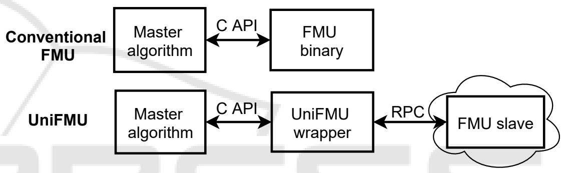


Figure 6: Conventional versus UniFMU approach.

```toml
[grpc]
linux = ["python3", "backend_grpc.py"]
macos = ["python3", "backend_grpc.py"]
windows = ["python", "backend_grpc.py"]
```

Listing 3: launch.toml.

gRPC. Is a RPC that is based on HTTP/2 for transporting messages, and Protocol Buffers (protobuf)[3] for describing the information in the messages as a

[2]https://grpc.io/
[3]https://developers.google.com/protocol-buffers

schema. An example of the `DoStep` schema, defined in the protobuf file, is illustrated in listing 4.

```
service SendCommand{
    rpc Fmi2DoStep(DoStep)
        returns (StatusReturn) {}
}
message DoStep{
    double current_time = 1;
    double step_size = 2;
    bool no_step_prior = 3;
}
```

Listing 4: Structure of DoStep message and Fmi2DoStep call in protobuf schema.

ZeroMQ. ZeroMQ[4] is a networking library that allows messages to be transmitted efficiently across transport layers such as TCP or as Inter Process Communication. Unlike the gRPC backend there is no explicit schema-file that dictates the structure of the messages. Instead, the messages are structured according to a simple protocol described in the developer documentation found in the UniFMU repository. The serialization of the messages is performed by the Serde[5] library. This makes it possible to automatically generate high-performance serialization for several formats such as:

- JSON
- Pickle
- Flatbuffers

For dynamically typed languages such as Python or JavaScript the schemaless approach may be simpler to implement.

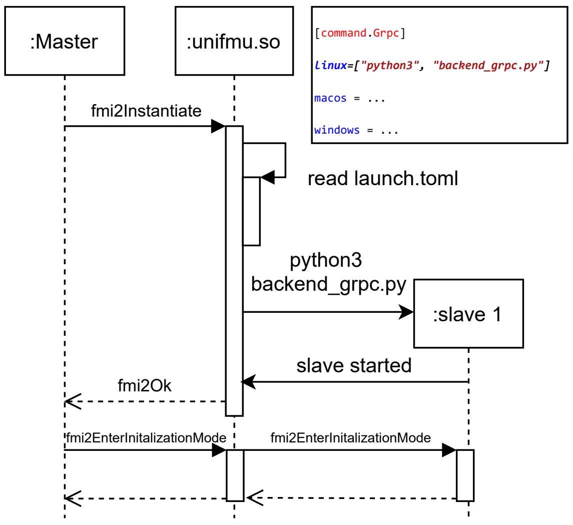
The main difference between these two backends is that gRPC uses a protobuf schema, while ZeroMQ is schemaless. Using a schema to define the messages and calls between the UniFMU wrapper and the FMU allows to declare the types of each message. This reduces the risk of using incorrect types when implementing the functions, easing the process of creating FMUs in statically-typed languages.

To understand the flexibility of this approach, it is useful to examine the process for creating an instance of an FMU, and the forwarding of the FMI commands from the binary to the slave instance, as depicted in fig. 7.

First the master algorithm invokes the `fmi2Instantiate` function defined by the binary. Following this, the wrapper reads the *launch.toml* file, which is present in any FMU generated by UniFMU. This is the configuration file which defines

[4]https://zeromq.org/

[5]https://serde.rs/



```
[command.Grpc]
Linux=["python3", "backend_grpc.py"]
macos = ...
windows = ...
```

Figure 7: Instantiation of slave and forwarding of FMI method calls.

the details about which communication backend is used to communicate with the slave process and even more importantly the specific command used to spawn the process. The *launch.toml* file in fig. 7 shows how the commands defined in the file are used to instantiate a Python FMU. For example an FMU implemented in Python using the gRPC backend would use the following command to launch the slave process:

```
python3 backend_grpc.py
```

Here the backend_grpc.py script serves as an implementation by implementing the `Fmi2DoStep` command, defined in the protobuf schema shown in listing 4. The `DoStep` message declares the parameters used in the `Fmi2DoStep` command. The complete protobuf schema can be found in our GitHub repository.

6 HOW TO EXTEND SUPPORT TO A NEW LANGUAGE?

There may be cases where a user would like to implement an FMU in a language not yet supported by the tool. One of the advantages of UniFMU is that new languages can be added without making any modification to the binaries of the FMU. This section shows the steps followed to extend the UniFMU support to include C# by implementing a gRPC backend. We chose the gRPC option, because a schema based serialization format like the one used by gRPC is especially suitable for compiled languages as it allows messages to be constructed using strongly typed objects. Following our running example, we will use the adder introduced in section 3 as a concrete example.

The implementation of the gRPC backend in C# amounts to implement the remote procedure calls defined in *unifmu_fmi2.proto* file under `schemas` in the repository[6]. The protobuf compiler, *protoc*, is then used to compile the UniFMU protobuf schema into C# code, as illustrated in fig. 8.

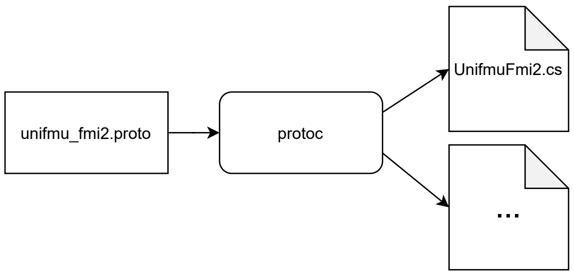


Figure 8: Generate C# files containing the protobuf schema.

The next steps are to create a base class of the FMI2 functions and implement the functionality of the adder. This can easily be done by translating these implementations from Python to C# code. The adder derives functions from the FMI2 base class, thus the specific implementations of the FMI2 functions are implemented in each FMU. After this, the gRPC server is coded, implementing the functions that were generated from the protobuf schema and calling them on the adder. A snippet of the `Fmi2DoStep` function implementation in the gRPC server is shown in **??**.

```
public override Task<StatusReturn>
    Fmi2DoStep(DoStep request,...)
{
    Fmi2Status status =
    this.fmu.DoStep(
        request.CurrentTime,
        request.StepSize,
        request.NoStepPrior);
    ...
}
```

Listing 5: gRPC server implementation of Fmi2DoStep.

The `Fmi2doStep` implementation is the only necessary function to implement in the adder, since this is where the core functionality is described. Listing 6 shows the `Fmi2doStep` implementation of the adder.

[6]https://github.com/INTO-CPS-Association/unifmu/

```
public override Fmi2Status DoStep(
    double currentTime, ...)
{
    this.c = this.a + this.b;
    return Fmi2Status.Ok;
}
```

Listing 6: C# example of fmi2doStep implementation.

One of the last steps is to implement the handshake connection that will be performed between the UniFMU wrapper and the FMU itself. This can be done in the `backend_grpc.cs` file, where the specific FMU to initialize can also be defined. The last step is to define the commands to be called for initialization of the FMU in the `launch.toml` file.

When generating a new C# FMU, the file structure will be as illustrated in fig. 9, where the `adder.cs` file can be exchanged for any other C# file.

```
adder_csharp.fmu
├─binaries
├─resources
│ ├─schemas
│ ├─model.cs^i
│ ├─backend_grpc.cs^b
│ ├─fmi2.cs^b
│ ├─csharp.csproj
│ └─launch.toml
└─modelDescription.xml
```

Figure 9: C# FMU directory tree. [b] denotes backend, [i] denotes implementation.

As we have demonstrated in this section, there is no need to write any C/C++ code when creating a new backend for UniFMU. The work amounts to implement a gRPC server using the protobuf schema and an abstract class containing the FMI2 function declarations.

7 EXECUTING FMUs INSIDE DOCKER CONTAINER

Invoking code from within an FMU needs the host machine to provide the necessary runtime environment to do so. For example, running python scripts requires that the host machine has a compatible python interpreter and all libraries used in the scripts installed. This limits the portability of the FMUs, especially when shared between different host machines, potentially running different operating systems.

To mitigate this issue runtime dependencies can be packaged into a virtualization environment such as docker[7]. A detailed description of the extension and more advanced topics, such as remote deployment, building and deployment settings can be found in (Schranz et al., 2021). In essence, to dockerize there is no need to change the wrapper code, all of the necessary actions are performed using a short script (run.sh for unix, run.ps1 for Windows), as seen in Listing **??**. The script builds a docker image using the FMUs global unique identifier and runs it. Once the wrapper disconnects, the process terminates, the container is stopped and disposed. The actions within the Dockerfile depend on the choice of backend. An adaptation to the framework to support all backends is under development.

To dockerize the FMU, the commands given in the launch.toml can be set to run a shellscript:

```
[grpc]
linux = ["/bin/bash", "run.sh" ]
macos = ["/bin/bash", "run.sh" ]
windows = ["powershell", ".\\run.ps1"]
```

```
#!/bin/sh
...
# build image
docker build -t "$uid" .
# run container
docker run --net=host --rm ...
```

Listing 7: Shell script, run.sh, used to deploy docker container on unix.

8 CONCLUDING REMARKS AND FUTURE WORK

In this paper we have introduced the tool UniFMU that makes it possible to implement FMUs in several languages with built-in support by the tool. We have demonstrated the process of creating an FMU in a supported language, Python, and have demonstrated the process for extending the tool to support C#. This makes it possible to produce FMUs with limited knowledge of the internal workings and with limited knowledge of C. In the future we hope to be able to use a similar Java extension to enable the Overture FMU (Thule et al., 2018) export to be moved over to the Visual Studio Code VDM substantiation (Rask et al., 2020).

One issue not discussed in the paper is performance. Invoking functions of an FMU using RPC instead of calling them directly through the C-ABI incurs a performance cost. As part of (Hatledal et al., 2019) the authors provide the results of experiments where the total simulation time of multiple FMUs is measured for the two approaches. The results seem to indicate that there is an almost constant overhead per RPC call resulting in the largest impact on models that must be simulated with small step sizes. A possible way to reduce the performance overhead is to reduce the total number of RPC calls. For example several FMI calls made in between two `fmi2DoStep` calls could be grouped and sent as a single message, since the outputs would not change in between.

UniFMU provides a way to package and execute arbitrary code inside of an FMU. However, it does not directly provide a way to ensure consistency between the model description and the code. Other works like (Legaard et al., 2020; Hatledal et al., 2020) solve this issue by declaring the interface in the implementation of the FMU and using code generation targeted for specific languages to export the model description. This approach cannot readily be applied for a large number languages without implementing without implementing code generation for each language individually. There is a work in progress GUI, where it is possible to import the FMU, and using the editor manage the input and output variables of the model description, as shown in fig. 5.

In addition to the standalone GUI for the tool, it is possible to bundle the offering in the INTO-CPS Association services, for instance in the front-end used to setup and launch co-simulations using MAESTRO, the INTO-CPS Application (Macedo et al., 2020; Talasila et al., 2020). It is also a possibility to enable the Model-Based Design of Cyber-Physical Systems community to use and make the tool available in the HUBCAP project cloud platform (Larsen et al., 2020; Kulik et al., 2020).

ACKNOWLEDGEMENTS

We acknowledge the funding from the Poul Due Jensen Foundation for funding the project Digital Twins for Cyber-Physical Systems (DiT4CPS).

REFERENCES

(2021). CATIA-Systems/FMPy. CATIA Systems.

Asghar, S. A. and Tariq, S. (2010). Design and implementation of a user friendly OpenModelica graphical connection editor. page 81.

Bertsch, C., Ahle, E., and Schulmeister, U. (2014). The Functional Mockup Interface – seen from an industrial perspective. pages 27–33.

[7]https://www.docker.com/

Controllab Products B.V. (2013). http://www.20sim.com/. 20-sim official website.

Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. (2018). Co-simulation: a Survey. *ACM Comput. Surv.*, 51(3):49:1–49:33.

Hatledal, L., Zhang, H., and Collonval, F. (2020). Enabling python driven co-simulation models with pythonfmu. pages 235–239.

Hatledal, L. I., Styve, A., Hovland, G., and Zhang, H. (2019). A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface. *IEEE Access*, 7:109328–109339.

Hatledal, L. I., Styve, A., Hovland, G., and Zhang, H. (2019). A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface. *IEEE Access*, 7:109328–109339. Conference Name: IEEE Access.

Kulik, T., Macedo, H. D., Talasila, P., and Larsen, P. G. (2020). Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In Fitzgerald, J. S. and Oda, T., editors, *Proceedings of the 18th International Overture Workshop*, pages 20–34. Overture.

Larsen, P. G., Macedo, H. D., Fitzgerald, J., Pfeifer, H., Benedikt, M., Tonetta, S., Marguglio, A., Gusmeroli, S., and Jr., G. S. (2020). An Online MBSE Collaboration Platform. pages 263–270. INSTICC, Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH.

Legaard, C. M., Gomes, C., Larsen, P. G., and Foldager, F. F. (2020). Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mockup Units. SummerSim '20. ACM New York, NY, USA.

Macedo, H. D., Rasmussen, M. B., Thule, C., and Larsen, P. G. (2020). Migrating the INTO-CPS Application to the Cloud. In Sekerinski, E., Moreira, N., Oliveira, J. N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., and Delmas, D., editors, *Formal Methods. FM 2019 International Workshops*, pages 254–271, LNCS 12233. Springer-Verlag.

Modelica Association (2019). Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads.

Rask, J. K., Madsen, F. P., Battle, N., Macedo, H. D., and Larsen, P. G. (2020). Visual Studio Code VDM Support. In Fitzgerald, J. S. and Oda, T., editors, *Proceedings of the 18th International Overture Workshop*, pages 35–49. Overture.

Schranz, T., Alfalouji, Q., Falay, B., Legaard, C., Wilfling, S., and Schweiger, G. (2021). Coupling physical and machine learning models: Case study of a residential building. In *14th International Modelica Conference (Submitted Manuscript)*.

Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schöggl, J.-P., Posch, A., and Nouidui, T. (2019). Functional Mock-up Interface: An empirical survey identifies research challenges and current barriers.

Simulink09 (2009). Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/.

Talasila, P., Sanjari, A., Villadsen, K., Thule, C., Larsen, P. G., and Macedo, H. D. (2020). Introducing Test Driven Development and Upgrades to the INTO-CPS Application. In Cleophas, L. and Massink, M., editors, *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops*, pages 311–317, Cham. Springer International Publishing.

Thule, C., Lausdahl, K., Gomes, C., Meisl, G., and Larsen, P. G. (2019a). Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61.

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

Thule, C., Palmieri, M., Gomes, C., Lausdahl, K., Macedo, H. D., Battle, N., and Larsen, P. G. (2019b). Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks. In *Co-Sim-19 workshop*.

Widl, E., Müller, W., Elsheikh, A., Hörtenhuber, M., and Palensky, P. (2013). The FMI++ library: A high-level utility package for FMI for model exchange. In *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, pages 1–6.