Date of publication xxxx 00, 0000, date of current version August 02, 2019.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Language and Platform Independent Co-simulation Framework based on the Functional Mock-up Interface

LARS I. HATLEDAL[1], ARNE STYVE[2], GEIR HOVLAND[3], AND HOUXIANG ZHANG.[1], (Senior Member, IEEE)
[1]Department of Ocean Operations and Civil Engineering, Norwegian University of Science and Technology, Ålesund, Norway
[2]Department of ICT and Natural Sciences, Norwegian University of Science and Technology, Ålesund, Norway
[3]Department of Engineering Sciences, University of Agder, Grimstad, Norway

Corresponding author: Lars I. Hatledal (e-mail: laht@ntnu.no).

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

ABSTRACT

The main goal of the Functional Mock-up Interface (FMI) standard is to allow the sharing of simulation models across tools. To accomplish this, FMI relies on a combination of XML-files and compiled C-code packaged in a zip archive. This archive is called a Functional Mock-up Unit (FMU). In theory, an FMU can support multiple platforms, but not necessarily in practice. Furthermore, software libraries for interacting with FMUs may not be available in a particular language or platform. Another issue is related to the protection of intellectual property (IP). While an FMU is free to only provide the C-code in its binary form, other resources within the FMU may be unprotected. Distributing models in binary form also opens up the possibility that they may contain malicious code.
In order to meet these challenges, this paper presents an open-source co-simulation framework based on FMI, which is language and platform independent thanks to the use of well-established remote procedure call (RPC) technologies. One or more FMUs are wrapped inside a server program supporting one or more language independent RPC systems over various network protocols. Together, they allow cross-platform invocation of FMUs from multiple, including previously unsupported, languages. The client-server architecture allows the effective protection of IP while also providing a means of protecting users from malicious code.

INDEX TERMS Co-simulation, Distributed simulation, FMI, FMU, Model Exchange, RPC

I. INTRODUCTION

No one simulation tool is suitable for all purposes, and complex heterogeneous models may require components from several different domains, perhaps developed in separate, domain-specific tools. Co-simulation refers to an enabling technique, where different sub-systems making up a global simulation are being modeled and run in a distributed fashion. Each sub-system is a simulator and is broadly defined as a black box capable of exhibiting behavior, consuming inputs, and producing outputs [1]. Co-simulation is a hot topic in research fields such as automotive [2], [3], maritime [4]–[6] and power systems [7]. Compared to more traditional monolithic simulations, co-simulation encourages re-usability, model sharing and fusion of simulation domains. A crucial point is that it allows users to simulate models exported from different tools together, enabling simulation of the type of complex cyber-physical systems found in areas such as the automotive and maritime industry. Fig. 1 illustrates a possible co-simulation scenario for a vessel, which requires models from several different domains. Co-simulation is absolutely imperative for this scenario to succeed, not only because models from different domains need to be coupled, but also because the models may originate from different, perhaps competing companies that would not be willing to share their models in any other form than as a black-box model.

Distributed co-simulation refers to the idea that a co-simulation can be distributed not only logically, but physically across a network. There are several reasons to perform a co-simulation with one or more remote simulators. For

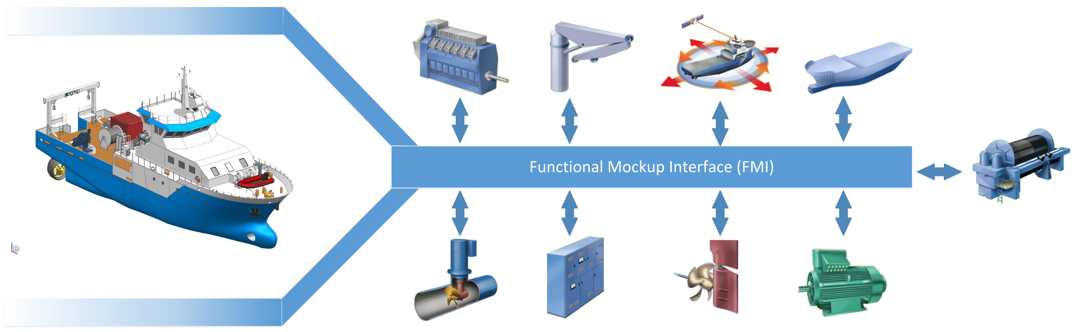

FIGURE 1: Simulation of a complex cyber-physical system in the maritime domain. The complete vessel model is constituted by the individual sub-modules connected through FMI for Co-simulation. *Sub-model figures courtesy of the Virtual Prototyping of Maritime Systems and Operations project (Research Council of Norway, grant nr. 225322).*

instance, a simulator may impose one or more requirements onto the simulation environment, such as a platform, software, or license requirement, that is for some reason impossible to meet. In such a case, the simulator can run in a compatible environment and accessed remotely. Also, if the overall simulation is suited for parallelization, it may be more efficient to balance the workload over several computation nodes. Another use-case is to prevent the execution of malicious code on a sensitive system by accessing it from a sand-boxed environment. Physically distributed co-simulation is also an excellent way of protecting intellectual property (IP), as clients would not have direct access to the simulation model. It's also worth noting that distributed co-simulations are vital for enabling digital twin technology, which requires the integration of industrial internet of things devices.

Multi-domain co-simulation is not without its challenges [8]. However, the Functional Mock-up Interface (FMI) standard [9] tries to make this task easier and more accessible by defining a standard way of interfacing simulation models. More specifically, FMI is a tool independent standard that supports both model exchange (ME) and co-simulation (CS) of dynamic models. A model implementing the FMI standard is known as a Functional Mock-up Unit (FMU). Many tools support FMUs, and it has become the de-facto standard for ME and CS. However, it does not solve everything and itself brings some problems. These issues are:

- Open-source FMI implementations exist for relatively few programming languages, like C, C++, Java and Python.
- FMI is cross-platform in theory, but not necessarily in practice. It depends on the exporting tools' ability to cross-compile.
- An FMU may require a particular software or license.
- An FMU may only support instantiating a single model-instance per process.
- The binary code within an FMU may contain malicious code.
- Reluctance to share FMUs even if the source code is provided in binary form, due to IP concerns.

Fortunately, distributed access can solve these issues. In describing how and presenting a benchmark, this paper builds on the work presented in [10], which introduced a framework for accessing models compatible with FMI 2.0 for CS and ME in a language and platform-independent manner. This is achieved using well-established remote procedure call (RPC) technologies, allowing cross-platform clients and servers to be written in most major languages, overcoming the issues listed above. For instance, this kind of architecture protects IP and prevents unintended distribution [11]. Furthermore, it allows the use of FMUs with special requirements, such as pre-installed software and license requirements, from otherwise incompatible systems.

Server and client implementations have been realized for both C++ and the Java Virtual Machine (JVM). Proof of concept clients also exists for Python, JavaScript and MATLAB. Thanks to the stub generation capability of selected RPC systems, such as Apache Thrift and gRPC, additional implementations are easy to realize as the selected RPC's compiler will auto-generate most, if not all, of the code required to interact with the remote FMUs.

The rest of the paper is organized as follows. Section II introduces recent and related work on FMI and distributed co-simulation. A presentation of the high-level architecture of the framework, as well as an introduction of the necessary background on the RPC standards and technologies referenced in this work, is provided in Section III. Implementation details follows in Section IV. A case study is presented in Section V along with a discussion of relevant findings. Finally, Section VI concludes the paper and provides directions for future work.

II. RELATED WORK

This section presents a brief summary of the current state of the FMI standard and distributed FMI based co-simulation.

A. THE FUNCTIONAL MOCK-UP INTERFACE

FMI is a tool independent standard that supports both (ME) and (CS) of dynamic models. Currently at version 2.0, the standard was one of the results of the MODELISAR project and the Modelica Association manages it today. A key goal of FMI is to improve the exchange of simulation models between suppliers and original equipment manufactures (OEMs).

An FMU is a model that implements the FMI standard that 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 first version of the standard, FMI 1.0, was released in 2010. Version 2.0 of the standard, was released in 2014. This version merged the two types of FMI standards and incorporated some major enhancements compared to the initial release. As a result, version 2.0 is not backwards compatible with version 1.0. In December 2017, the Modelica Association released a preliminary feature list for version 3.0 that includes:

- Meta-data for ports and icons, allowing for a more consistent representation across tools.
- Support for multi-dimensional variables (arrays).
- Co-simulation with events.
- Inclusion of a binary data type.
- Access of intermediate output values between communication points.
- Better support for source code FMUs.

Since the inception of the FMI standard, a multitude of libraries and software tools that support it have been implemented. As of March 2018, the official FMI web page lists 108 such tools, 71 of which support invocation of FMI 2.0 compatible simulation models. Table. 1 provides a summary of open-source libraries with FMI import capabilities. Clearly, the standard is solving a real problem. However, practical challenges persist.

- FMI is cross platform in theory, but in practice can only be used cross-platform if the exporting tools have the ability to cross-compile native binaries. Many do not.
- While FMI has been implemented in several languages, such as C [12], [13], C++ [14], [15], Python [16], [17] and Java [18]–[20], out-of-the-box support for FMI is still missing in many languages.
- An FMU may require a license or pre-installed software on the target computer, making the FMU unavailable on many systems.
- Some FMI implementations only support CS, making parts of the standard unavailable. Others may also support ME but may not provide an easy way of solving them. Thus, some users may find the threshold for utilizing this feature too high.
- The standard does not cover IP protection. While, model exporters can implement protection as they see fit. Some model owners may worry about leaking IP and might be reluctant to share FMUs with others.
- As an FMU's application code can be delivered in binary form, end-users may be afraid to use it because it could contain malicious elements.

B. DISTRIBUTED FMI BASED CO-SIMULATION

Table. 2 provides a list of open-source tools for simulating FMUs. Among these, the ones that support distributed invocation of FMUs are as follows.

DACCOSIM (Distributed Architecture for Controlled CO-SIMulation) [21], is an FMI-based co-simulation environment written in Java. DACCOSIM lets the user design and execute a simulation requiring the collaboration of multiple FMUs on multi-core computation nodes or clusters. For complex scenarios with many FMUs and/or connections, a domain specific language can be used to replace the graphical user interface (GUI). It also includes a command line interface (CLI) for running co-simulations. JavaFMI [19] is used for simulating FMUs. DACCOSIM is released under the LGPLv3 license and is available for both Windows and Linux.

Coral [6] is a free and open-source software for distributed FMI based co-simulation. It supports FMI 1.0 and 2.0 for CS and is licensed under the MPL 2.0. Coral was developed as part of the R&D project Virtual Prototyping of Maritime Systems and Operations [5]. According to its creators, Coral is primarily a C++ library, but also acts as a tool as it requires setting up and running several programs in a distributed fashion. It also comes with a CLI for running simulations. Coral works by installing a server program called a *slave provider* on each of the machines that should participate in a simulation. This program is responsible for publishing information on which FMUs are available on that machine to the network, as well as loading and running FMUs at the request of the master software, which acts as a client. Coral relies on the FMI Library to interact with FMUs, while the ZeroMQ middleware facilitates networking. Google Protocol Buffers are used for encoding/decoding messages sent over the network.

FMI Go! [22] is an MIT-licensed software infrastructure designed to perform distributed simulations with FMI compatible components, that runs on Windows, Linux and Mac OS X. It supports CS as well as ME FMUs by wrapping

TABLE 1: Open-source software libraries providing FMI import capabilities.

	Language				FMI support					
					CS		*ME*			
Name	*C*	*C++*	*Java*	*Python*	*v1.0*	*v.2.0*	*v1.0*	*v2.0*	Version	**License**
FMI Library	x				x	x	x	x	2.0.3	BSD
FMU SDK		x			x	x	x	x	2.0.6	BSD
FMI4cpp		x				x		x[b]	0.7.0	MIT
FMI++		x	x[a]	x[a]	x	x	x[b]	x[b]	-	BSD
PyFMI				x	x	x	x[b]	x[b]	2.5	LGPLv3
FMPy				x	x	x	x[b]	x[b]	0.2.11	BSD
JFMI			x		x		x		1.0.2	MIT
JavaFMI			x		x	x			2.25.3	LGPLv3
FMI4j			x			x		x[b]	0.22.1	MIT

[a] Through SWIG
[b] Can solve ME FMUs

TABLE 2: Open-source software tools for simulating FMUs.

	FMI support											
	CS		*ME*									
Name	*v1.0*	*v2.0*	*v1.0*	*v2.0*	**Standalone**	**Plugin**	**Distributed**	**API**	**CLI**	**GUI**	**Version**	**License**
Coral	x	x			x		x	x	x		0.10.0	MPLv2
DACCOSIM		x			x		x		x	x	2018	LGPLv3
FMI Go!	x	x	x	x	x		x		x		0.5.0	MIT
FIDE		x				x				x	-	-
FUMOLA	x	x	x	x		x		x		x	alpha	-
Hopsan	x	x								x	2.11.0	GPLv3
Maestro		x			x		x	x			1.0.2	ICAPL
MasterSim	x	x			x			x	x	x	0.5.3	LGPLv3
OpenModelica	x		x	x	x					x	1.13.0	OSMC-PL
OMSimulator		x		x	x	x		x	x	x	2.0.1	OSMC-PL
Ptolemy II	x	x	x	x	x			x		x	10.0.1	MIT
Xcos FMU wrapper	x		x			x				x	0.6	CeCILL

these into CS FMUs. ME FMUs are preferred, as they allow the FMI Go! run-time environment to provide rollback and directional derivatives of the FMU. In CS FMUs, these features are considered optional and are often absent, but in fact they may be required to achieve accurate and or stable simulations. FMI Go! uses a client-server architecture, where a server hosts an individual FMU. Google Protocol Buffers are used for mapping the various FMI functions to messages, which are transmitted using the ZeroMQ middleware. The message passing interface is also supported. The global stepper is then a client, consuming results produced by the FMUs. For applications that would want access to the simulation data, such as loggers, visualization etc., the global stepper serves also as a server. The system specification and parameterization (SSP) is used for defining the structure of a simulation. A bare-bones CLI for this purpose also exists.

λ-Sim is a tool implemented on top of Amazon Web Services (AWS) that converts FMI based simulation models into REST APIs. Provided with an FMU bundled with a JSON configuration file, λ-Sim builds a series of AWS that will run simulations upon requests from a RESTful API. Two services are provided. *Lambda*, a service that operates on-demand servers for running simulations and return meta-data associated to the requested model, and *Apigateway* - the service that exposes the server via a public REST API. A web-based GUI is available, allowing users to load the generated API, simulate the model and visualize the results.

A software architecture for simulation and visualization based on FMI and web technologies was presented in [23]. This work leveraged the Java specific RPC technology Remote Method Invocation [24] for distributed access to FMUs.

The proposed framework differs from the ones mentioned above in that it totally separates itself from the master algorithm. It is a completely standalone project that provides the infrastructure required to invoke FMI compatible models, such as FMUs, remotely using RPCs. Multiple RPC systems over several network protocols are supported. Time stepping, variable routing, plotting, and tasks typically performed by a master tool are left implemented by the integrating tool. This creates a lightweight framework that is easy to use and is re-usable.

Rather than having several tools implementing their own, perhaps non-modular or internal, distribution mechanism, we hope that the solution offered here can be considered as an alternative or drop-in replacement for existing solutions. However, this work can only be integrated into simulation masters with a centralized design. Data must flow through the master, and not directly between slaves.

Highly related to the work presented in this paper is the **Distributed Co-Simulation Protocol (DCP)** [25], which is a standard for real-time and non-real-time system integration and simulation. The DCP is compatible with FMI, and just like FMI, it defines only the slave. The design of a master is not in scope of the specification. Recently it was adopted by the Modelica Association as a Modelica Association Project (MAP)

This work is similar to the DSP in that both initiatives aim to enable and promote distributed co-simulation. However, this work does not define a standard, but mimics FMI for function definitions and leverages existing cross-platform RPC frameworks for serialization and networking. This makes it less complex, more accessible and easier to use. However, this work relies on reliable network communication and no special considerations have been made for real-time system or hardware-in-the-loop integration, making DSP more suited for these kinds of co-simulation tasks.

III. THE PROPOSED FRAMEWORK

This section introduces the high-level concepts of the proposed framework. The framework uses a client-server architecture and embraces cross-platform and language independent RPCs for communication between clients and servers. Such RPCs have several benefits compared to ad-hoc message passing systems, such as:

1) Tried and tested.
2) Not having to re-invent the wheel.
3) Built in serialization and networking.
4) Schema based code validation and generation.
5) Large open-source communities surrounding them.

In particular, Apache Thrift and gRPC are supported - both of which are schema based and available in a wide range of languages. Additionally, JSON-RPC is supported by one of the server implementations. JSON-RPC is language and transport agnostic and can be used to fill any gaps left by the other RPCs regarding language, transport and/or platform support, effectively making the framework accessible from virtually any client application.

A. KNOWLEDGE BACKGROUND

This section will introduce the necessary background on the RPC technologies and standards used by the proposed framework.

1) Remote Procedure Call

An RPC is an abstraction for executing a function call (or procedure) located in a different address space (e.g another computer). RPCs provide more structure than request-response message-passing systems. Typically, a RPC request demands a response and error handling is baked into the protocol. Many RPCs also rely on a pre-definition of available functions and types, either through schema definitions or language interfaces. This allows statically typed languages to verify the message-passing logic at compile time, making bugs less likely to appear in production code.

2) Protocol Buffers

Protocol buffers [26], or protobuf, are Google's mechanism for serializing structured data. Compared to common alternatives for data serialization over the wire, such as XML and JSON, protobuf generate much smaller data packages because they use a binary format. Messages are compiled using a predefined schema, allowing messages to be more compact. The schema is specified in a file with a *.proto* extension. Both regular messages and RPC services can be defined using the protobuf interface definitions language (IDL). However, the RPC feature requires a 3rd party plugin because the protobuf library itself does not implement it.

3) gRPC

gRPC [27] is a language- and platform-neutral open-source RPC system, initially developed at Google, with support for a wide range of programming languages. Official support exists for C/C++, C#, Node.js, PHP, Ruby, Python, Go and Java. It relies on HTTP/2 for transport and protobuf for data serialization. gRPC is essentially an implementation of the protobuf RPC. Listing. 1 demonstrates an example RPC service definition using gRPC/protobuf.

Listing 1: Example protobuf schema with service definitions

```
message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}

service HelloService {
    rpc SayHello (HelloRequest) returns (
        HelloResponse);
}
```

4) Apache Thrift

Apache Thrift [28] is a cross-platform RPC framework that supports several protocols and transports, e.g. binary over TCP/IP and JSON over HTTP. Initially developed at Facebook, it is now an open source project maintained by the Apache Software Foundation. A variety of programming languages are supported, including C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi. It is schema-based, with definitions and services declared in *.thrift* files. A analogous example to the protobuf definition in Listing. 1 is shown in Listing 2.

Listing 2: Example Thrift schema

```
service HelloService {
    string sayHello(1: string greeting);
}
```

5) JSON-RPC

JSON-RPC [29] is a stateless, light-weight RPC protocol. The protocol uses JSON as the data format and is designed to be simple. JSON-RPC is only a specification and is totally transport agnostic. An example of a JSON-RPC call is given in Listing. 3. Here, a method called *sayHello* is given a single parameter *"World!"*. The result sent back to the invoking part is *"Hello, World!"*. In case of errors, the result part of a response is replaced by an error object containing a code and a explanatory message.

Listing 3: Example JSON RPC call

```
-> {"jsonrpc": "2.0", "method": "sayHello", "
    params": {"greeting": "World!"}, "id": 1}

 //on success
<- {"jsonrpc": "2.0", "result": "Hello, World
    !", "id": 1}

//on error
<- {"jsonrpc": "2.0", "error": {"code":
    -32601, "message": "Method not found"}, "
    id": 1}
```

B. FRAMEWORK OVERVIEW

The software architecture is shown in Fig. 2 and consists of three main parts, each of which is described in more detail below.

1) The Discovery Service

The discovery service is a web application whose main responsibility is to redistribute information about and the location of available FMUs. This information can be obtained visually through a web interface, or programmatically through HTTP requests. The following HTTP services are defined:

- */availablefmus*: Called by user applications. Returns a JSON formatted string containing information about all available FMUs registered with the discovery service. The information include data from the *modelDescription.xml* as well as the IP address of the host machine and the RPC port(s).
- */register*: Called by proxy-servers on start-up. Registers the server with this discovery service. Transmits network information and information about the *modelDescription.xml* for each locally available FMU.
- */ping*: Called by a proxy-server at regular intervals. Otherwise the discovery service will consider it to be offline.

The discovery service is an optional feature and is not required when the remote end-point of an RPC service is known to the client application, for instance when running the server on a physically accessible machine.

Multiple discovery services may be online at any given time. They may be public or used internally in a restricted network.

2) Proxy-Servers

A proxy-server is responsible for making available one or more FMUs over a set of RPCs. Implementations should support Thrift and or gRPC. Additional RPCs, such as JSON-RPC can also be supported.

In addition to the RPC support, a full implementation must be able to communicate with the discovery service over HTTP. Upon starting the server, the address of a discovery service should be specified. In order to ensure that the list of available FMUs is up to date, the server must ping the discovery service over HTTP, signaling that it is still online. When enough time has passed without such a notification, the server is considered offline and is removed from the discovery service.

The framework supports both ME and CS FMUs running on the back-end, but the user is only provided with a CS API, as ME models are wrapped. The user can configure which solver will be used for wrapping the ME model, subject to availability of certain solvers, which depends on the server implementation.

3) Proxy-Clients

A proxy-client is used to connect with the FMUs hosted by the remote server(s), and can be implemented in a wide variety of languages.

Using Thrift or gRPC, the process of generating the required source-code for interacting with an remote FMU is quite straightforward. Listing. 4 shows the command required for generating the required sources when targeting Thrift in JavaScript. Similarly, Listing. 5 shows how C++ sources for gRPC are generated. The same recipes apply to targeting other languages.

Listing 4: Generating JavaScript sources for interfacing with remote FMUs using Thrift.

```
thrift -js service.thrift
```

Listing 5: Generating C++ sources for interfacing with remote FMUs using gRPC.

```
protoc -I=. --plugin=protoc-gen-grpc=
    grpc_cpp_plugin --cpp_out=. --grpc_out=.
    service.proto
```

The framework accomplishes several things, such as:

- **Additional language support**. FMUs can be accessed in previously unsupported languages with low effort, as no XML must parsed and no C-code has to be interfaced. Depending on the RPC used, stubs are auto-generated.
- **Cross-platform access to any FMU**. FMUs can be invoked from unsupported platforms, i.e an FMU compiled only for Windows can be invoked from a Linux system. Naturally, a server running on a platform supported by the FMU must be available.
- **FMI compliance without FMU packaging**. It allows models to be compliant with the FMI standard without actually being packaged as an FMU. From a client's perspective, there is no difference between a "physically backed" FMU and one implemented in-memory. All the client sees is the RPC interface mimicking FMI.
- **Relaxed run-time constraints**. FMUs that require special software and/or licenses can be invoked from

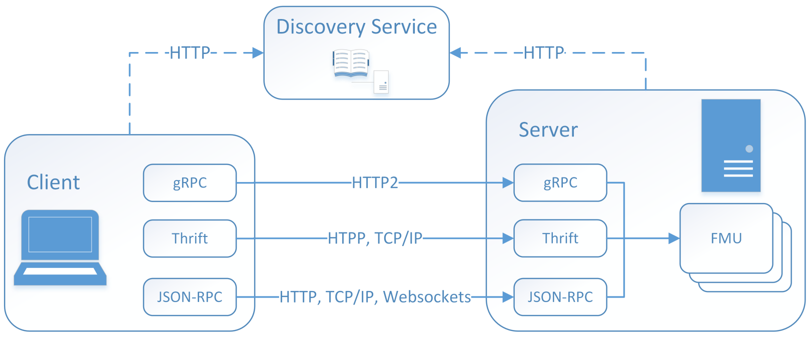

FIGURE 2: The high level software architecture of the proposed framework. The client-server architecture relies on RPCs for communication. The Discovery Service is optional, and serves as a centralized hub for locating available FMUs.

otherwise incompatible systems, granted that a server fulfilling the needs is available.
- **Re-usability**. As the framework is decoupled from the master algorithm, it can be used by any software tool with a centralized master architecture that wants to support distributed execution of FMUs.
- **Protection against malicious code.** Non-source code FMUs could possibly contain malicious software. This framework makes it easy to place FMUs in a sand-boxed environment and invoke them remotely and safely.
- **Multiple instances of models that cannot share processes.** Some FMUs can only be instantiated once per process. One of the common reasons for this is the use of global variables. Distributed access allows the master to circumvent this restriction.

IV. IMPLEMENTATION DETAILS

This section describes some of the implementation details related to the proposed framework. Currently, it comes with server implementations for C++ and the JVM. Client implementations exist for C++ and the JVM. Additionally, proof of concept implementations for Python, JavaScript and MATLAB exists. A web-server for keeping track of available RPC servers, known as the discovery service, is also bundled.

A. THE DISCOVERY SERVICE

The discovery service is implemented in Kotlin, a statically typed language 100% interoperable with Java. The front-end offers basic functionality such as the ability for users to download available RPC schemas and to view information about available FMUs in a structured way. The user interface is somewhat crude but serves its purpose.

B. PROXY-SERVER

Two server implementations have been realized, each described more in detail below. Which one to deploy in production depends on factors like:

1) Which RPC to use.
2) Memory footprint and performance.
3) Maturity and stability of the implementation.
4) The quality of the available solvers for wrapping ME models.

No one implementation will excel at everything.

1) JVM

The JVM implementation is written in Kotlin and rely on FMI4j [18], an FMI implementation for JVM languages that supports FMI 1.0 and 2.0 for CS and ME. Out of the box, ME models can be wrapped as CS ones using solvers from the Apache Commons Math3 [30] package. Compared to other open-source FMI implementations targeting the JVM, such as JFMI [20] and JavaFMI [19], FMI4j is the only one to support ME for 2.0. Furthermore, FMI4j uses the Java Native Interface (JNI) rather than Java Native Access (JNA) for interfacing with the native FMI functions, which significantly improves performance. The calling overhead for a single native call using JNA can be an order of magnitude greater than equivalent JNI [31].

The implementation supports Thrift (TCP/IP - binary, HTTP - JSON), gRPC (HTTP2 - protocol buffers) as well as JSON-RPC (HTTP, TCP/IP and WebSockets), and is considered as the reference implementation.

2) C++

The C++ implementation is cross-platform and is written in modern C++17. All dependencies are available using the cross-platform package manager *conan*, making it easy to build. Currently, Thrift (TCP/IP - binary, HTTP - JSON) and gRPC (HTTP2 - protocol buffers) are supported RPCs.

FMI4cpp [15], an FMI 2.0 implementation for C++, is used for interacting with FMUs. It supports both CS and ME, where the latter can be wrapped as the former using solvers from *Boost odeint* [32]. The main goal of the FMI4cpp library is to be as easy to use and install as possible. To achieve this, it makes use of modern C++ features and supports installation using the *vcpkg* and *conan* package managers.

C. PROXY-CLIENTS

The framework comes bundled with client implementations for C++, the JVM, Python and JavaScript. The two latter are somewhat crude and ought to be considered as proof of concept. They are, however, bundled with the source code to showcase how easy it is to interface with the framework from new languages. A MATLAB demo using JSON-RPC over HTTP is also available. In the case of MATLAB, it is worth noting that one of the existing Java clients can be used.

The C++ and JVM implementations are more elaborate, providing a unified, higher level API for its users. No matter which RPC is used, there is no difference between a remote and local co-simulation slave for the user. As illustrated by Figure. 3, they all share the same interface, defined by FMI4cpp and FMI4j for C++ and JVM implementations respectively. Assuming a tool is using one of these FMI implementations, support for distributed execution can be seamlessly added with minimal changes to the existing code base. See Listing. 6 for an example.

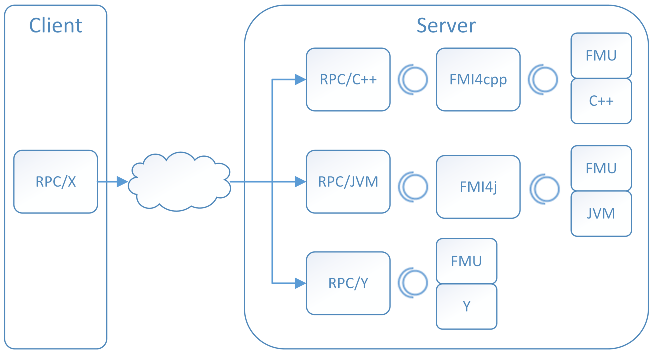

FIGURE 3: FMI4cpp and FMI4j's slave interface could hide slaves derived from either an in-memory implementation or an actual FMU. Slaves in any language supported by the chosen RPC could also be implemented directly behind the RPC layer.

Listing 6: JVM Thrift example, written in Kotlin

```
val localModel: Model = Fmu.from(<url or file>) //
    FMI4j API
val remoteModel: Model = ThriftFmuClient.
    socketClient(<host>, <port>).load(<guid, url,
    or file>)

val model = ... //one of the above

val stepSize = ...
val slave = model.newInstance()
slave.simpleSetup()
slave.doStep(stepSize)
slave.terminate()
```

After running the JavaScript code generation using the command shown earlier in Listing. 4, the code shown in Listing. 7 can be written. Here, Thrift is configured to use HTTP transport and JSON encoding. Subsequently an FMU slave is instantiated on the remote server and stepped for 1s until termination. The process is similar for the 14+ other languages supported by Thrift, as well as gRPC and its many supported languages.

Listing 7: Invoking an FMU from JavaScript using Thrift over HTTP.

```
var transport = new Thrift.TXHRTransport("http://
    localhost:9091/thrift")
var protocol  = new Thrift.TJSONProtocol(transport
    )
var client    = new FmuServiceClient(protocol)

var fmu_id = client.loadFromXXX() //load from url
    or guid
var slave_id = client.createInstanceFromCS(fmu_id)

client.setupExperiment(slave_id)
client.enterInitializationMode(slave_id)
client.exitInitializationMode(slave_id)

var stop = 1.0
var step_size = 1.0/100
do {
    var result = client.step(slave_id, step_size)
    if (result.status != 0) {
        break
    }
} while (result.simulationTime <= stop)

client.terminate(slave_id)
```

V. CASE STUDY AND DISCUSSION

The following presents a case study to illustrate the performance of the various RPCs when running a somewhat representative selection of FMUs using different network topologies. These are:

1) Client and server running on *localhost*.
2) Client and server running on separate computers connected directly by Ethernet.
3) Client and server running on separate computers connected by Ethernet through a switch.

The different topologies are illustrated in Fig. 4.

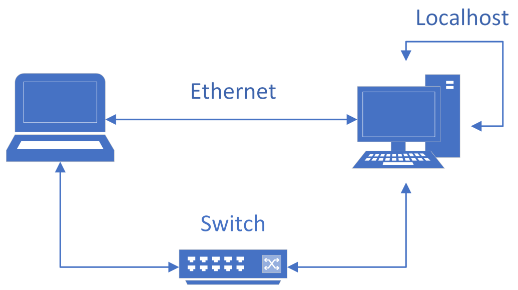

FIGURE 4: The different network topologies used in the case study.

The setup was as follows. A laptop running Ubuntu 18.04 and a desktop computer running Windows 10 was utilized. Both are 64-bit systems. The laptop is fitted with an Intel i7-6600U with four logical cores, while the desktop is equipped

TABLE 3: Performance of running the 33 FMUs listed in Table. 4 on the JVM. FMI4j is used to run the API version, which serves as a baseline. The execution time required to step the FMUs using the Thrift and gRPC RPCs over the different communication mediums are shown as a multitude of this.

		API	***Thrift***			***gRPC***		
		In-memory	*localhost*	*cable*	*switch*	*localhost*	*cable*	*switch*
Time[ms]	*Sequential*	1	9.5X	27.8X	29.3X	25.4X	50.6X	48.7X
	Parallel	1	3.6X	6.6X	8.3X	9.8X	11.4X	12.5X

with an Intel core i7-4770 with eight logical cores. As the desktop is the most powerful of the two, it was selected as the server. The switch used during the experiment was a ZyXEL GS-1055 v2 Gigabit Ethernet Switch. The JVM implementation of the proposed framework were used by both the client and server. While a C++ version is also available, there are two main reason for running the JVM implementation on both client and server. First, the JVM version is more mature and second, using a JVM language like Kotlin to set up the test case was deemed easier.

In order for an exporting tool to prove compliance with the FMI standard it must upload a number of FMUs to the FMI cross-check [33] repository. As these FMUs are publicly available and represent a wide variety of models, they are suited for testing in this experiment.

In this case-study, 33 of the 133 FMUs compatible with 64-bit Windows at the time of the test were selected. The requirements for selection were as follows.

1) A non-zero step-size must be defined.
2) In order to run on the test system, the FMU must require neither an execution tool nor a license.
3) In order not to skew the tests, the step-size must be greater or equal than 0.0001 with a stop time less than 20 seconds.
4) The FMUs must not write files to the current directory, as this proved to cause run-time issues in parallel and/or subsequent runs.

Some vendors provide many similar FMUs, exported only with different versions of the software. In order to keep a more well-balanced set of FMUs, exported FMUs from no more than two versions from the same vendor were included. All FMUs that were included in the experiment are listed in Table. 4.

The experiment was conducted as follows. For each configuration, all 33 FMUs were first run sequentially, then in parallel. Table. 4 also shows how long it took to step each FMU using the specified step-size and stop time when invoking the FMU directly using FMI4j (in-memory), as well as through the framework using Thrift and gRPC. Not surprisingly, calling the FMI API directly is much faster than distributed invocation. As would be expected, we observe that running both client and server on *localhost* is faster than a point-to-point Ethernet connection between two computers, which again is generally faster than having to go through a network switch.

A more compact representation of the results are shown in Table. 3, which also features results from simulating the FMUs in parallel. Figure. 5 presents the data shown in this table as well. From the results, it is clear that, at least on the JVM and for this particular set of FMUs, Thrift is a considerably faster than gRPC. However, even when running the client and server on the same machine Thrift is about 9.5x slower on average than in-memory API calls.

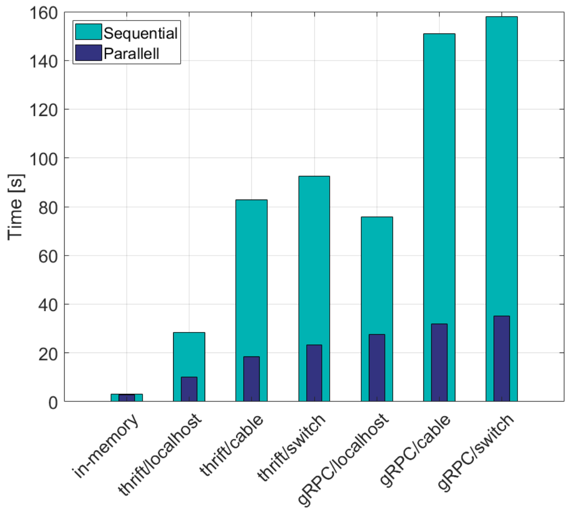

FIGURE 5: Bar plot of the results shown in Table. 3

Running in parallel provides quite a significant performance gain, moving from a ~ 9.5x to a ~ 3.6x performance loss compared to local API calls. By parallelizing the test case onto a computer cluster with the same per-FMU computational power as the desktop used in this particular test, one could in theory achieve similar or even better results than running in-memory. It took $87.5s$ to run the Thrift case sequentially using a network switch. Using a computer cluster, one could distribute each FMU onto a computation node. Theoretically, this should yield a total computation time of $87.5s/33 = 2.65s$, which in this case is comparable to running non-distributed.

Although distributed co-simulation in general comes with a significant performance overhead, it's worth remembering that this approach is required to accommodate certain use-cases, such as overcoming license and software requirements, access from unsupported platforms or languages and safe invocation of an FMU by running it in a sandboxed environment. And as pointed out above, in cases were performance is crucial, the FMUs can be distributed to several computational nodes and stepped in parallel, provided the models involved allows the simulation to be parallelized.

Also worth noting is how FMUs that are computational heavy, such as the 20Sim *TorsionBar* were only marginally slower to run distributed. This makes such FMUs prime candidates for distribution. With a more powerful host system, the overall performance would actually increase compared to local execution. For FMUs that require low step-sizes the results tell another story though. In such cases, such as for the SimulationX *DoublePendelum* model, where 30000 invocations is required to simulate $3s$, the overhead of a network call becomes painfully obvious. Compare this to the 20Sim model, which only requires 126 invocations to simulate $12.56s$. As a result, distributed execution of models that require low time-steps should ideally be avoided when performance is important.

VI. CONCLUSION AND FUTURE WORK

This paper has presented a language- and platform- independent co-simulation framework based on the Functional Mock-up Interface.

It has been designed to easily allow distributed execution of FMI compatible models such as FMUs. The client server architecture allows FMUs to be invoked from previously unsupported languages and on incompatible platforms. It also makes it possible to shield the user from malicious code, while still being able to integrate models on a local machine. Since the framework is independent of the master algorithm, it can be re-used in different software projects.

Some of the highlighted features of the presented framework are:

- Brings FMI capabilities to previously unsupported languages and otherwise incompatible platforms.
- By implementing the RPC functions directly, FMI compliant models can be implemented without having to package them as FMUs.
- Allows code re-use between projects that requires distributed execution of FMUs, independent of implementation language.
- By hosting their own FMUs, companies may share their models without worrying about leaking IP.
- A unified slave interface for C++ and JVM users. On these platforms, local and remote slaves implement the same interface. This makes it trivial to switch between local and remote execution of a particular FMU.

The results provided in Section V clearly show that there is some considerable performance overhead related to distributed co-simulation. However, parallelizing the work make it possible to minimize this overhead. In any case, one should not decide to run distributed co-simulations for its own sake. Running the scenario locally, using regular API calls, should be the preferred approach. This framework provides an alternative when that's not feasible.

Server implementations exist for C++ and the JVM, while client implementations exist for JavaScript, Python, C++ and the JVM. Due to the language independent nature of the RPC frameworks and protocols used, and especially the code-generation feature of selected RPC frameworks, additional client implementations require little effort. For instance, FMU-proxy was recently integrated into one of the deliverables of the Open Simulation Platform, a joint industry project initiated by DNV GL, Kongsberg maritime, SINTEF Ocean and NTNU [34]. Using the Thrift RPC, integration was easily and quickly realized by taking the generated RPC code from the Thrift compiler and writing a thin wrapper, stitching the two APIs together. Furthermore, this integration supports the up-and-coming SSP standard [35].

Several enhancements to the framework are planned for the future, including:

- *Authentication*. Some form of authentication should be added, restricting who may interact with a particular proxy-server and or discovery service.
- *Wrap client as FMU*. It would be beneficial to be able to wrap one of the available clients as an FMU. This would allow FMI compliant tools to benefit from distributed simulation with zero modifications.
- *FMI 3.0 support*. Support for the next version of the standard will be added.

Additionally, the framework should be more thoroughly documented and continuously maintained.

Pre-built server executables for Linux and Windows can be found at https://github.com/NTNU-IHB/FMU-proxy. Client libraries for Java are available through *maven* at https://jitpack.io/#NTNU-IHB/FMU-proxy, while C++ artifacts are available as *conan* recipes. There are no immediate plans to publish the Python and JavaScript clients through any type of package managers. However, they are easily obtained from the publicly available source repository.

REFERENCES

[1] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-simulation: a survey," ACM Computing Surveys (CSUR), vol. 51, no. 3, p. 49, 2018.

[2] Z. Zhang, E. Eyisi, X. Koutsoukos, J. Porter, G. Karsai, and J. Sztipanovits, "A co-simulation framework for design of time-triggered automotive cyber physical systems," Simulation modelling practice and theory, vol. 43, pp. 16–33, 2014.

[3] R. L. Bücs, L. Murillo, E. Korotcenko, G. Dugge, R. Leupers, G. Ascheid, A. Ropers, M. Wedler, and A. Hoffmann, "Virtual hardware-in-the-loop co-simulation for multi-domain automotive systems via the functional mock-up interface," in Languages, Design Methods, and Tools for Electronic System Design. Springer, 2016, pp. 3–28.

[4] Y. Chu, L. I. Hatledal, F. Sanfilippo, V. Æs, H. Zhang, H. G. Schaathun et al., "Virtual prototyping system for maritime crane design and operation based on functional mock-up interface," in OCEANS 2015-Genova. IEEE, 2015, pp. 1–4.

[5] 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, 2016, pp. V007T06A018–V007T06A018.

[6] S. Sadjina, L. T. Kyllingstad, M. Rindarøy, S. Skjong, V. Æsøy, D. E. Fathi, V. Hassani, T. Johnsen, J. B. Nielsen, and E. Pedersen, "Distributed co-simulation of maritime systems and operations," arXiv preprint arXiv:1701.00997, 2017.

[7] C. Shum, W.-H. Lau, T. Mao, H. S.-H. Chung, K.-F. Tsang, N. C.-F. Tse, and L. L. Lai, “Co-simulation of distributed smart grid software using direct-execution simulation,” IEEE Access, vol. 6, pp. 20 531–20 544, 2018.
[8] M. Faruque, V. Dinavahi, M. Steurer, A. Monti, K. Strunz, J. Martinez, G. Chang, J. Jatskevich, R. Iravani, and A. Davoudi, “Interfacing issues in multi-domain simulation tools,” IEEE Transactions on Power Delivery, vol. 27, no. 1, pp. 439–448, 2012.
[9] 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, no. 076. Linköping University Electronic Press, 2012, pp. 173–184.
[10] L. I. Hatledal, H. Zhang, A. Styve, and G. Hovland, “Fmu-proxy: A framework for distributed access to functional mock-up units,” in Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019, no. 157. Linköping University Electronic Press, 2019.
[11] E. Durling, E. Palmkvist, and M. Henningsson, “Fmi and ip protection of models: a survey of use cases and support in the standard,” in Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, no. 132. Linköping University Electronic Press, 2017, pp. 329–335.
[12] JModelica, “Fmi library,” 2017, (Date accessed 16-May-2019). [Online]. Available: http://www.jmodelica.org/FMILibrary
[13] QTronic, “Fmu sdk,” 2014, (Date accessed 16-May-2019). [Online]. Available: https://github.com/qtronic/fmusdk
[14] E. Widl, W. Müller, A. Elsheikh, M. Hörtenhuber, and P. Palensky, “The fmi++ library: A high-level utility package for fmi for model exchange,” in Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on. IEEE, 2013, pp. 1–6.
[15] L. I. Hatledal, “Fmi4cpp,” 2018, (Date accessed 16-May-2019). [Online]. Available: https://github.com/FMU-proxy/FMI4cpp
[16] Dassault Systems, “Fmpy,” 2017, (Date accessed 16-May-2019). [Online]. Available: https://github.com/CATIA-Systems/FMPy
[17] C. Andersson, J. Åkesson, and C. Führer, “Pyfmi: A python package for simulation of coupled dynamic models with the functional mock-up interface,” Technical Report in Mathematical Sciences, vol. 2016, no. 2, 2016.
[18] 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 Proceedings of The 59th Conference on Simulation and Modelling (SIMS 59), 26-28 September 2018, Oslo Metropolitan University, Norway, no. 153. Linköping University Electronic Press, 2018, pp. 37–42.
[19] J. S. Cortes Montenegro, “Javafmi una librería java para el estándar functional mockup interface,” 2014.
[20] D. Broman, C. Brooks, E. A. Lee, T. S. Nouidui, S. Tripakis, and M. Wetter, “Jfmi - a java wrapper for the functional mock-up interface,” 2013, (Date accessed 16-May-2019). [Online]. Available: https://ptolemy.eecs.berkeley.edu/java/jfmi/
[21] J. É. Gómez, J. J. H. Cabrera, J.-P. Tavella, S. Vialle, E. Kremers, and L. Frayssinet, “Daccosim ng: co-simulation made simpler and faster,” in Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019, no. 157. Linköping University Electronic Press, 2019.
[22] C. Lacoursière and T. Härdin, “Fmi go! a simulation runtime environment with a client server architecture over multiple protocols,” in Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017, no. 132. Linköping University Electronic Press, 2017, pp. 653–662.
[23] L. I. Hatledal, H. G. Schaathun, and H. Zhang, “A software architecture for simulation and visualisation based on the functional mock-up interface and web technologies,” in Proceedings of The 57th Conference on Simulation and Modelling (SIMS 56): October, 7-9, 2015, Linköping University, Sweden. Linköping University Electronic Press, Linköpings universitet, 2015.
[24] E. Pitt and K. McNiff, Java. rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc., 2001.
[25] M. Krammer, M. Benedikt, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner et al., “The distributed co-simulation protocol for the integration of real-time systems and simulation environments,” in Proceedings of the 50th Computer Simulation Conference. Society for Computer Simulation International, 2018, p. 1.
[26] K. Varda, “Protocol buffers: Google’s data interchange format,” Google Open Source Blog, Available at least as early as Jul, 2008.
[27] gRPC, “grpc,” 2018, (Date accessed 16-May-2019). [Online]. Available: https://grpc.io/
[28] Apache Software Foundation, “Apache thrift,” 2019, (Date accessed 16-May-2019). [Online]. Available: https://thrift.apache.org/
[29] JSON-RPC Working Group and others, “Json-rpc 2.0 specification,” 2012, (Date accessed 27-March-2019). [Online]. Available: https://www.jsonrpc.org/specification
[30] Apache foundation, “Apache commons math3,” 2019, (Date accessed 27-March-2019). [Online]. Available: http://commons.apache.org/proper/commons-math/
[31] JNA authors, “Jna faq,” 2018, (Date accessed 27-March-2019). [Online]. Available: https://github.com/java-native-access/jna/blob/5.2.0/www/FrequentlyAskedQuestions.md
[32] Boost developers, “Apache commons math3,” 2019, (Date accessed 27-March-2019). [Online]. Available: http://headmyshoulder.github.io/odeint-v2/
[33] C. Bertsch, E. Ahle, and U. Schulmeister, “The functional mockup interface-seen from an industrial perspective,” in Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden, no. 096. Linköping University Electronic Press, 2014, pp. 27–33.
[34] OSP, “Open simulator platform,” 2019, (Date accessed 16-May-2019). [Online]. Available: https://opensimulationplatform.com/
[35] J. Köhler, H.-M. Heinkel, P. Mai, J. Krasser, M. Deppe, and M. Nagasawa, “Modelica-association-project “system structure and parameterization”– early insights,” in The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan, no. 124. Linköping University Electronic Press, 2016, pp. 35–42.

LARS IVAR HATLEDAL received the B.Sc. degree in automation from NTNU, Aalesund, Norway, 2013. After his graduation he started working part-time as a research assistant with the mechatronics lab at NTNU Ålesund, Department of Ocean Operations and Civil Engineering. In 2017 he received a M.Sc. in Simulation and Visualisation also from NTNU, where he is currently working towards a PhD degree with the Department of Ocean Operations and Civil Engineering. His research interests include simulation, artificial intelligence and 3D visualisation.

ARNE STYVE received the B.E. degree (honors) in microelectronics and software engineering from the University of Newcastle upon Tyne, Newcastle upon Tyne, U.K., in 1991. He is an Assistant Professor with the Department of ICT and Natural Sciences, NTNU, Ålesund, Norway. He has more than 20 years of experience in the SW industry, having worked in areas like fire detection systems, the Norwegian defence industry and digital television broadcasting systems (Tandberg Television). In 2004, he joined what later became the Offshore Simulator Centre AS (OSC), where he held the position of R&D Manager until his return to NTNU Ålesund in 2014.

GEIR HOVLAND received the M.Sc. degree in engineering cybernetics from the Norwegian University of Science and Technology, Trondheim, Norway, in 1993, and the Ph.D. degree in robotics from the Australian National University, Canberra, ACT, Australia, in 1997. He was a Research Engineer at ABB Norway, Sweden, and Switzerland (Oslo, Västerås, and Baden, respectively) during 1997-2003, and took part in the development of ABBs control system for industrial robots. He was a Senior Lecturer in Mechatronics at the University of Queensland, Brisbane, QLD, Australia, during 2004-2006, and has been a Professor in Robotics and Control Systems with the University of Agder, Grimstad, Norway, since 2007. He is currently the Director of the Centre for Research-based Innovation Offshore Mechatronics and Technical Manager of the Norwegian Motion Lab (both in Grimstad, Norway). Dr. Hovland is the Chief Editor of the MIC Journal.

HOUXIANG ZHANG (M'04–SM'12) Prof. Houxiang Zhang received the Ph.D. degree on Mechanical and Electronic Engineering in 2003. From 2004, he worked at the Institute of Technical Aspects of Multimodal Systems (TAMS), Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, University of Hamburg, Germany. In Feb. 2011, he finished the Habilitation on Informatics at University of Hamburg. Dr. Zhang joined the NTNU, Norway in April 2011 where he is a Professor on Robotics and Cybernetics. The focus of his research lies on two areas. One is on biological robots and modular robotics. The second focus is on virtual prototyping and maritime mechatronics. He has applied for and coordinated more than 20 projects supported by Norwegian Research Council (NFR), German Research Council (DFG), and industry. In these areas, he has published over 160 journal and conference papers as author or co-author. Dr. Zhang has received four best paper awards, and four finalist awards for best conference paper at International conference on Robotics and Automation.

TABLE 4: Overview of FMUs and settings used for the experiment, as well as performance results.

						Time elapsed [ms]						
						API	***Thrift***			***gRPC***		
Tool	**Version**	**Name**	**Step size [s]**	**Stop time [s]**	**No. calls**	*In-memory*	*localhost*	*cable*	*switch*	*localhost*	*cable*	*switch*
20sim	4.6.4.8004	TorsionBar	0.1	12.56	126	*1971*	*2062*	*2202*	*2152*	*2322*	*2296*	*2383*
ASim	2019FD01	Circle_SWC	0.1	4.0	40	*66*	*69*	*63*	*66*	*109*	*142*	*162*
		Speed_SWC	0.01	0.4	40	*21*	*32*	*53*	*57*	*96*	*111*	*143*
dSPACE TargetLink	Release 2018-B	Fmucontroller	0.001	0.35	350	*20*	*83*	*239*	*253*	*428*	*731*	*730*
		Fmufuelratecontroller	0.01	20.0	2000	*21*	*324*	*1058*	*1066*	*1487*	*2427*	*2450*
		FmuTL_VelocityController	0.001	0.2	200	*19*	*51*	*123*	*176*	*153*	*505*	*247*
DS FMU Export from Simulink	2.1	BouncingBalls_sf	0.001	10.0	10000	*25*	*1516*	*4196*	*5033*	*4750*	*9077*	*8970*
		TestModel1_sf	0.001	10.0	10000	*22*	*1496*	*4823*	*4959*	*4254*	*9123*	*8531*
		TriggeredSubsystems_sf	0.001	10.1	10100	*23*	*1523*	*4300*	*4980*	*4228*	*7975*	*8333*
	2.3.0	BouncingBalls_sf	0.001	10.0	10000	*23*	*1495*	*4544*	*4927*	*4245*	*8662*	*8180*
		TestModel1_sf	0.001	10.0	10000	*27*	*1511*	*4701*	*4999*	*4197*	*8319*	*8032*
		TestModel2_sf	0.001	10.0	10000	*39*	*1479*	*4109*	*4921*	*4207*	*8700*	*8286*
		TriggeredSubsystems_sf	0.001	10.0	10000	*27*	*1490*	*4715*	*4950*	*4207*	*8845*	*8217*
FMIToolbox MATLAB	2.1	Continuous	0.01	10.0	1000	*21*	*169*	*529*	*524*	*459*	*874*	*869*
		Discontinuities	0.01	10.0	1000	*20*	*167*	*739*	*526*	*460*	*795*	*875*
		EmbeddedCode	0.01	10.0	1000	*29*	*168*	*500*	*506*	*456*	*1143*	*855*
		IntegrateSignal	0.01	10.0	1000	*19*	*171*	*640*	*521*	*463*	*1051*	*865*
		Signal_Attributes	0.01	10.0	1000	*21*	*168*	*473*	*512*	*461*	*1117*	*857*
	2.3	Continuous	0.01	10.0	1000	*22*	*169*	*570*	*521*	*457*	*809*	*838*
		Discontinuities	0.01	10.0	1000	*24*	*171*	*499*	*512*	*452*	*1049*	*857*
		EmbeddedCode	0.01	10.0	1000	*26*	*171*	*605*	*527*	*449*	*812*	*865*
		IntegrateSignal	0.01	10.0	1000	*24*	*173*	*723*	*506*	*449*	*999*	*851*
MapleSim	2015.1	ControlledTemperature	0.001	10.0	10000	*29*	*1484*	*4610*	*4986*	*4187*	*7626*	*8038*
		CoupledClutches	0.01	1.5	150	*21*	*46*	*92*	*122*	*99*	*147*	*177*
	2018	ControlledTemperature	0.001	10.0	10000	*34*	*1496*	*4899*	*4955*	*4219*	*7851*	*8125*
		CoupledClutches	0.01	1.5	150	*20*	*50*	*87*	*110*	*94*	*150*	*168*
MWorks	2016	BouncingBall	0.01	10.0	1000	*23*	*171*	*891*	*523*	*452*	*1026*	*846*
		ControlledTemperature	0.001	10.0	10000	*27*	*1510*	*4913*	*4936*	*4145*	*10333*	*8218*
		CoupledClutches	0.001	1.5	1500	*28*	*251*	*923*	*775*	*651*	*1433*	*1261*
		MixtureGases	0.001	1.0	1000	*32*	*178*	*641*	*522*	*449*	*1034*	*859*
SimulationX	3.7.41138	ControlledTemperature	0.001	10.0	10000	*38*	*1520*	*4272*	*4916*	*4132*	*7971*	*7987*
		CoupledClutches	0.0001	1.5	15000	*95*	*2428*	*7257*	*7488*	*6209*	*13921*	*12427*
		DoublePendulum	0.0001	3.0	30000	*124*	*4630*	*13912*	*14962*	*12270*	*23932*	*24707*
			SUM	**257.61**		**2982**	**28422**	**82901**	**87489**	**75696**	**150986**	**145209**