FMU-proxy: A Framework for Distributed Access to Functional Mock-up Units

Lars Ivar Hatledal[1] Houxiang Zhang[1] Arne Styve[2] Geir Hovland[3]

[1]Department of Ocean Operations and Civil Engineering, NTNU, Norway, {laht,hozh}@ntnu.no
[2]Department of ICT and Natural Sciences, NTNU, Norway, asty@ntnu.no
[3]Department of Engineering Sciences, UiA, Norway, geir.hovland@uia.no

Abstract

The main goal of the Functional Mock-up Interface (FMI) standard is to allow simulation models to be shared 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 an Functional Mock-up Unit (FMU) and uses the extension *.fmu*. In theory, an FMU can support multiple platforms, however this is not always the case and depends on the type of binaries the exporting tool was able to provide. Furthermore, a library providing FMI support may not be available in a particular language, and/or it may not support the whole standard. Another issue is related to the protection of Intellectual Property (IP). While an FMU is free to only provide the C-code in binary form, other resources shipped with the FMU may be unprotected.

In order to overcome these challenges, this paper presents FMU-proxy, an open-source framework for accessing FMUs across languages and platforms. This is done by wrapping one or more FMUs behind a server program supporting multiple language independent Remote Procedure Call (RPC) technologies over several network protocols. Currently, Apache Thrift (TCP/IP, HTTP), gRPC (HTTP/2) and JSON-RPC (HTTP, WebSockets, TPC/IP, ZeroMQ) are supported. Together, they allow FMUs to be invoked from virtually any language on any platform. As users don't have direct access to the FMU or the resources within it, IP is more effectively protected.

Keywords: RPC, FMI, Co-simulation, Model Exchange

1 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. How such components could be integrated in a standardized way is a problem the Function Mock-up Interface (FMI) (Blochwitz et al., 2012) aims to solve. More specifically, FMI is a tool independent standard to support both Model Exchange (ME) and Co-Simulation (CS) of dynamic models. Currently at version 2.0, the standard was one of the results of the MODELISAR project and is today managed by the Modelica Association.

A model implementing the FMI standard is known as an Functional Mock-up Unit (FMU), and is distributed as a zip-file with the extension *.fmu*. This archive contains:

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

The FMI standard consists of two main parts:

- *FMI for Model Exchange (ME):* Models are exported without solvers and are described by differential, algebraic and discrete equations with time-, state- and step-events.
- *FMI for Co-Simulation (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.

It's worth noting that a single FMU may support both ME and CS, and that the former may be wrapped by an importing tool into the latter.

FMI has seen high adaption rates since it's inception in 2011. The official tools page at fmi-standard.org/tools currently shows about 120 tools supporting FMI in one way or another. Clearly, the standard is solving a real problem. However, there are still some practical challenges related to it.

- FMI is cross platform in theory, but in practice this depends on the exporting tools ability to cross-compile native binaries. This is often not the case, making some FMUs unavailable for a certain platform.
- While FMI has been implemented in several languages, such as C (JModelica, 2017; QTronic, 2014), C++ (Widl et al., 2013; Hatledal, 2018), Python (Dassault Systems, 2017; Andersson et al.,

2016) and Java (Hatledal et al., 2018; Cortes Montenegro, 2014; Broman et al., 2013), 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 supports CS, making parts of the standard unavailable. Others may support ME also, but may not provide an easy way of solving them. Thus, some users may find the threshold for utilizing this feature too high.
- IP protection is not covered by the standard, however, model exporters are free to implement such mechanism as they see fit. Regardless, some model owners may worry about leaking IP and might be reluctant in sharing FMUs with others.

In order to resolve these issues, we present FMU-proxy, a framework for accessing FMUs compatible with FMI 2.0 for CS and ME in a language and platform independent way. The language and platform independent nature of the framework is achieved using well established RPC technologies, allowing clients and servers for FMU-proxy to be written in almost any language, on any platform. As noted by (Durling et al., 2017), server solutions such as presented in this paper are effective at protecting IP and unintended distribution. Furthermore, they allow FMUs with special requirements, such as pre-installed software and licence requirements, to be utilized on other systems.

Server implementations already exist for C++ and for the Java Virtual Machine (JVM), while client implementations exist for C++, Python, JavaScript and the JVM. Thanks to the stub generation capability of selected RPC frameworks, additional implementations in other languages are easy to realize as most of the code will be generated by the RPC compiler.

FMU-proxy is different from other similar frameworks offering distributed execution of FMUs in that it completely separates itself from the master algorithm. It is a completely standalone project which provides the infrastructure required to invoke FMUs over the wire. And just that.

Rather than having a number of tools creating their own, perhaps non-modular or internal, distribution mechanism, we hope FMU-proxy can be considered as an alternative or drop-in replacement for existing solutions. Possibly, creating a eco-system of remotely available FMUs in the process.

The source code of FMU-proxy is available online[1] under a permissive MIT license.

The rest of the paper is organized as follows. First some related work is given, followed by a presentation of the high-level architecture of the framework and subsequent implementation notes. Finally, a conclusion and future works are given.

[1] https://github.com/NTNU-IHB/FMU-proxy

2 Related work

Since the inception of the FMI standard, a multitude of libraries and software tools supporting the standard has been implemented. As of November 2018, the official FMI web page lists 120 such tools. Most of which supports invocation of FMI 2.0 compatible simulation models. A list of open-source tools with FMI import capabilities are given in Table. 1. Of these tools, four support distributed invocation of FMUs. These are:

DACCOSIM (Distributed Architecture for Controlled CO-SIMulation) (Galtier et al., 2015; Dad et al., 2016), a FMI compatible master algorithm, that lets the user design and execute a simulation requiring the collaboration of multiple FMUs on multi-core computation nodes or clusters. DACCOSIM is implemented in Java and is built on-top of the Eclipse Rich Client Platform, which provides the user with a GUI for setting up and running co-simulations. For complex scenarios with many FMUs and/or connections, a DSL can be used to replace the GUI. JavaFMI (Cortes Montenegro, 2014) is used for simulating and building FMUs. For communications, the ZeroMQ middleware is used. DACCOSIM is released under the AGPL license and is available for both Windows and Linux.

Coral (Sadjina et al., 2017) is a free and open-source software for distributed FMI based co-simulation, licensed under the MPL 2.0. Coral support FMI 1.0 and 2.0 for CS and was developed as part of the R&D project Virtual Prototyping of Maritime Systems and Operations (ViProMa) (Hassani et al., 2016). According to the authors, 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. Additionally, it comes with a Command Line Interface (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, and exposes a subset of the FMI standard, compatible with both FMI 1.0 and 2.0, over the network. It also handles loading and running FMUs at the request of the master software, which acts as a client. Coral relies on the FMI Library (JModelica, 2017) to interact with FMUs, while networking is facilitated by the ZeroMQ middleware. Google Protocol Buffers are used for encoding/decoding messages sent over the network. A special feature of Coral is that slaves run in parallel, with variable values passed between them in a distributed fashion. Loggers and visualizers must therefore be implemented as FMUs themselves.

FMI Go! (Lacoursière and Härdin, 2017) is an open-source (MIT) distributed software infrastructure to perform distributed simulations with FMI compatible com-

DOI 10.3384/ecp1915779

Table 1. 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		0.9.0	MPLv2
DACCOSIM		x			x		x			x	2.1.0	AGPL
FMI Go!	x	x	x	x	x		x		x		-	MIT
FIDE		x				x				x	-	-
FUMOLA	x	x	x	x		x		x		x	alpha	-
Hopsan	x	x								x	2.10.0	GPLv3
INTO-CPS		x			x					x	-	MIT
MasterSim	x	x			x			x	x	x	0.5.0	LGPLv3
Ptolemy II	x	x	x	x	x			x		x	10.0.1	MIT
Xcos FMU wrapper	x	x	x			x				x	0.6	CeCILL
λ-Sim	x						x			x	-	-
OpenModelica	x		x	x	x					x	1.12.0	GPLv3

ponents, that runs on Windows, Linux and Mac OS X. Both CS and ME FMUs are supported, where ME FMUs are wrapped into CS FMUs. ME FMUs are preferred, as then the FMI Go! run-time environment can provide roll-back and directional derivatives of the FMU. In CS FMUs, these features are considered optional and are often lacking, but may be required to achieve accurate and or stable simulations. FMI Go! used a client-server architecture, where a server hosts an individual FMU. Google Protocol Buffers are used for mapping the various FMI functions to messages that are transmitted using the ZeroMQ middleware. The Message Passing Interface (MPI) 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) (Köhler et al., 2016) is used for defining the structure of a simulation. Additionally, a bare-bone CLI for this purpose also exists.

λ-Sim (Bonvini, 2016) 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 services that will run simulations upon requests from a RESTful API. A web-based GUI is available, allowing users to load the generated API, simulate the model and visualize the results.

In (Hatledal et al., 2015) a software architecture for simulation and visualization based on FMI and web technologies was presented, using the Java only Remote Method Invocation (RMI) system for distributed access of FMUs.

Efforts has also been made to integrate the High Level Architecture (HLA) (Dahmann et al., 1997) and FMI in the works of (Awais et al., 2013) and (Garro and Falcone, 2015).

Additionally, the emerging standard Distributed Co-Simulation Protocol (DCP) (Krammer et al., 2018) should be mentioned. It is subject to proposal as a standard for real-time and non-real-time system integration and simulation, and standardization as a Modelica Association Project (MAP). 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.

FMU-proxy is similar to the DSP in that it aims to enable distributed Co-Simulation. However, it does not define a standard, but mimics FMI for function definitions and leverages existing RPC frameworks and protocols for serialization and networking. It also makes no special considerations for real-time system integration like DSP does.

FMU-proxy differs from the other tools mentioned above as it does not actually simulate any FMUs. It merely provides access to the FMUs in a flexible way, supporting multiple RPCs and network protocols. Time stepping, variable routing, plotting etc. and other typical task performed by a master tool is left implemented by the integrating tool. This is a feature, allowing FMU-proxy to be lightweight, easy to use and re-usable in different software tools.

3 Software Architecture

This section introduces the high level concepts of FMU-proxy. The software architecture is shown in Fig. 1 and consists of three main parts:

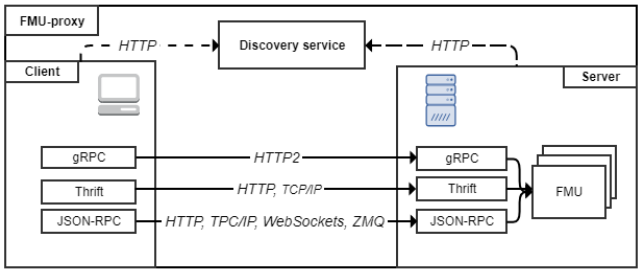

Figure 1. Software architecture.

1. **Discovery Services** A discovery service is a web application whose main responsibility is to communicate to users information about and the location of available FMUs. This information can be obtained visually through a web interface, or programmatically through an HTTP request.

   The discovery service has the following three HTTP services:

   - */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 the discovery server. Transmits network information, and information about the *modelDescription.xml* for each locally available FMU.
   - */ping*: Called by the proxy-servers at regular intervals, otherwise they will be considered to be offline by the discovery service.

   The discovery service is an optional feature and is not required when the remote end-point of an RPC service can be easily obtained. For instance when running the server on a physically accessible machine, allowing the IP address and RPC port(s) to be manually obtained. Another use case could be running both the client and server on *localhost* to enable invocations on FMUs from an otherwise unsupported language.

   Multiple discovery services may be online at any given time.

2. **Proxy-server**

   A proxy-server is responsible for making available one or more FMUs over a set of RPCs. At the very least, an implementation should support both Thrift and gRPC. Additional RPCs, such as JSON-RPC are optional.

   In addition to the RPC support, an implementation must be able to communicate with the discovery service over HTTP. Upon starting the server, the remote address of a discovery service should be specified. In order to ensure that the list of available FMUs are kept up to date, a heartbeat connection to the discovery service is established. At regular intervals, the server sends a ping - or heartbeat - over HTTP signalling that it is still online. When enough time has passed without such a notification, the server is considered offline and it's listing is subsequently removed from the discovery service.

   FMU-proxy 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. Which solver and parameters to use are configurable by the user, however the availability of certain solvers are dependent on the server implementation.

3. **Proxy-clients**

   Proxy clients are used to connect with the FMUs hosted by the remote server(s). FMU-proxy aims to provide flexibility, such that clients can be implemented in a wide variety of languages and platform.

   Using Thrift or gRPC, the process of generating the required source-code for interacting with an remote FMU is quite straightforward. Listing. 1 shows the command required for generating the required sources when targeting Thrift in JavaScript. Similarly, Listing. 2 shows how C++ sources for gRPC are generated.

   Listing 1. Generating JavaScript sources for interfacing with remote FMUs using Thrift.

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

   Listing 2. 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 has to be 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 is required.
- **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 otherwise incompatible systems.
- **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.

DOI 10.3384/ecp1915779

4 Implementation

This section describes some of the implementation details related to FMU-proxy. Currently, it comes with server implementations for C++ and the JVM. Client implementations exist also for C++ and the JVM. Additionally, proof of concept implementations for Python and JavaScript are bundled. In addition to the servers and clients, FMU-proxy comes bundled with an implementation of a discovery service.

4.1 The Discovery Service

The discovery service has been implemented in Kotlin, a statically typed language 100% interoperable with Java. The front-end seen in Fig. 2 has been implemented using PrimeFaces, a UI component framework for Java Server Faces (JSF). It 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.

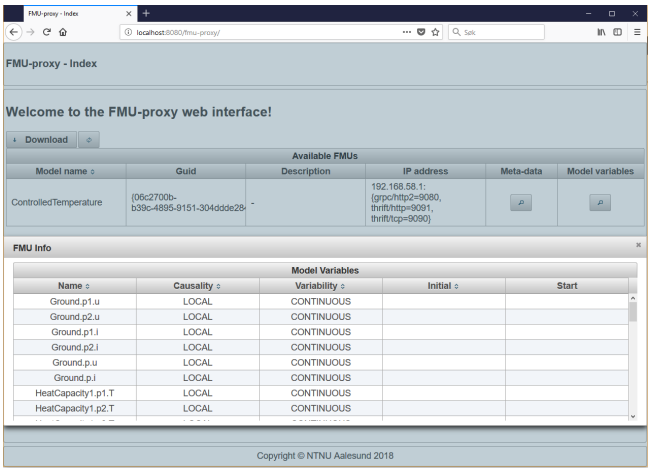

Figure 2. The discovery service's web interface. Here available FMUs are listed, showing network information and data from the *modelDescription.xml*.

4.2 Proxy-server

Two server implementations have been realized, each described more in detail below. Which one to deploy in production depends on the users need for RPCs supported, stability, stability, quality of the available ME solvers, memory foot-print and performance. No one implementation will excel at everything.

4.2.1 JVM

The JVM implementations is written in Kotlin and rely on FMI4j (Hatledal et al., 2018) for interacting with FMUs. FMI4j supports FMI 2.0 for CS and ME. ME models can be wrapped as CS ones using solvers from Apache Commons Math.

The implementation supports Thrift (TPC/IP - binary, HTTP - JSON), gRPC (HTTP2 - protocol buffers) as well as JSON-RPC (HTTP, TCP/IP, WebSockets, ZeroMQ). Of the two current implementations, this one is considered the most stable and feature rich.

4.2.2 C++

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

FMI4cpp (Hatledal, 2018) is used for interacting with FMUs. It supports FMI 2.0 for CS and ME. ME models can be wrapped as CS ones using solvers from Boost odeint.

4.3 Proxy-client

FMU-proxy comes bundled with client implementations for C++, the JVM, Python and JavaScript. The two latter are 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 FMU-proxy from new languages. A MATLAB demo using JSON-RPC over HTTP is also available.

The C++ and JVM implementations are more elaborate, providing a unified, higher level API for the users. No matter which RPC is used, there is no difference between a remote and local FMU 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.

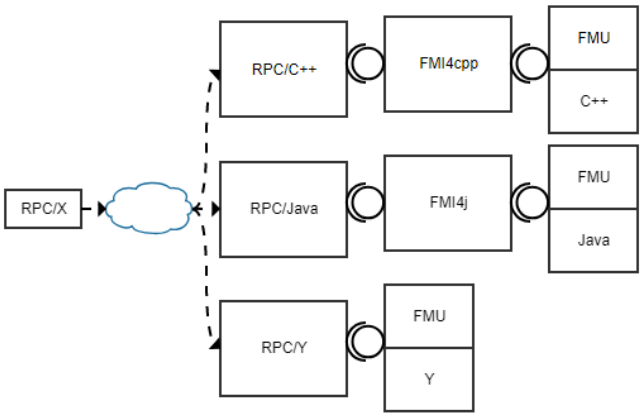

Figure 3. FMI4cpp and FMI4j's slave interface could hide slaves stemming from either an in-memory implementation or an actual FMU. A slave in any language supported by the chosen RPC could also be implemented directly behind the RPC layer.

5 Conclusion and Future Work

In this paper an open-source framework for working with FMUs across languages and platforms, named FMU-proxy, has been presented. It has been designed to allow

distributed execution of FMUs, which also enables access to FMUs in previously unsupported languages and on incompatible platforms. Since FMU-proxy is independent of the master algorithm, it can be re-used across software projects.

Some features of FMU-proxy include:

- 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 into FMUs.
- Allows code re-use between projects that requires distributed execution of FMUs, independent of implementation language.
- Enables companies to securely share FMUs. By hosting their own proxy server and directory service, neither the FMUs nor the knowledge about them leaves the company controlled servers.
- A unified slave interface for C++ and JVM users. On these platforms, local and remote slaves implement the same interface.

Server implementations exists for C++ and the JVM, while client implementations exists 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, further client implementations in additional languages require little effort.

Several enhancements to FMU-proxy is planned for the future, including:

1. Automatic distribution of FMUs over the network. It should be possible to upload an FMU to the Discovery Service, which in turn should find a suitable server for it to run on.
2. Manual distribution of FMUs over the network. It should be possible for the user to directly upload an FMU to an available proxy-server.
3. Publication of the C++ implementation to the cross-platform C++ library manager *vcpkg*.
4. Benchmark results, comparing the different implementations, RPCs and local vs. distributed execution of FMUs.
5. Once released, FMI 3.0 support will be added.

FMU-proxy is available from GitHub at `https://github.com/NTNU-IHB/FMU-proxy`. Here, pre-built server executables can be obtained. Client libraries for Java are available through *maven* at `https://jitpack.io/#NTNU-IHB/FMU-proxy`, while client libraries for C++ will be available through *vcpkg*.

6 Acknowledgement

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

References

Christian Andersson, Johan Åkesson, and Claus Führer. Pyfmi: A python package for simulation of coupled dynamic models with the functional mock-up interface. *Technical Report in Mathematical Sciences*, 2016(2), 2016.

Muhammad Usman Awais, Peter Palensky, Atiyah Elsheikh, Edmund Widl, and Stifter Matthias. The high level architecture rti as a master to the functional mock-up interface components. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 315–320. IEEE, 2013.

Torsten Blochwitz, Martin Otter, Johan Akesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 076, pages 173–184. Linköping University Electronic Press, 2012.

Marco Bonvini. Lambdasim, 2016. URL `https://github.com/mbonvini/LambdaSim`. (Date accessed 11-November-2018).

David Broman, Christopher Brooks, Edward A. Lee, Thierry S. Nouidui, Stavros Tripakis, and Michael Wetter. Jfmi - a java wrapper for the functional mock-up interface, 2013. URL `https://ptolemy.eecs.berkeley.edu/java/jfmi/`. (Date accessed 23-June-2018).

Johan Sebastian Cortes Montenegro. Javafmi una librería java para el estándar functional mockup interface. 2014.

Cherifa Dad, Stephane Vialle, Mathieu Caujolle, Jean-Philippe Tavella, and Michel Ianotto. Scaling of distributed multi-simulations on multi-core clusters. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on*, pages 142–147. IEEE, 2016.

Judith S Dahmann, Richard M Fujimoto, and Richard M Weatherly. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*, pages 142–149. IEEE Computer Society, 1997.

Dassault Systems. Fmpy, 2017. URL `https://github.com/CATIA-Systems/FMPy`. (Date accessed 23-June-2018).

Erik Durling, Elias Palmkvist, and Maria Henningsson. Fmi and ip protection of models: A survey of use cases and support in the standard. pages 329–335, 07 2017.

Virginie Galtier, Stephane Vialle, Cherifa Dad, Jean-Philippe Tavella, Jean-Philippe Lam-Yee-Mui, and Gilles Plessis. Fmi-based distributed multi-simulation with daccosim. In

Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, pages 39–46. Society for Computer Simulation International, 2015.

Alfredo Garro and Alberto Falcone. On the integration of hla and fmi for supporting interoperability and reusability in distributed simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 9–16. Society for Computer Simulation International, 2015.

Vahid Hassani, Martin Rindarøy, Lars T Kyllingstad, Jørgen B Nielsen, Severin Simon Sadjina, Stian Skjong, Dariusz Fathi, Trond Johnsen, Vilmar Æsøy, and Eilif Pedersen. Virtual prototyping of maritime systems and operations. In *ASME 2016 35th International Conference on Ocean, Offshore and Arctic Engineering*, pages V007T06A018–V007T06A018. American Society of Mechanical Engineers, 2016.

Lars Ivar Hatledal. Fmi4cpp, 2018. URL `https://github.com/SFI-Mechatronics/FMI4cpp`. (Date accessed 16-November-2018).

Lars Ivar Hatledal, Hans Georg Schaathun, and Houxiang 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.

Lars Ivar Hatledal, Houxiang Zhang, Arne Styve, and Geir 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*, number 153, pages 37–42. Linköping University Electronic Press, 2018.

JModelica. Fmi library, 2017. URL `http://www.jmodelica.org/FMILibrary`. (Date accessed 09-December-2017).

Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, and Mikio Nagasawa. Modelica-association-project "system structure and parameterization"–early insights. In *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, number 124, pages 35–42. Linköping University Electronic Press, 2016.

Martin Krammer, Martin Benedikt, Torsten Blochwitz, Khaled Alekeish, Nicolas Amringer, Christian Kater, Stefan Materne, Roberto Ruvalcaba, Klaus Schuch, Josef 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*, page 1. Society for Computer Simulation International, 2018.

Claude Lacoursière and Tomas 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*, number 132, pages 653–662. Linköping University Electronic Press, 2017.

QTronic. Fmu sdk, 2014. URL `http://www.qtronic.de/de/fmusdk.html`. (Date accessed 23-June-2018).

Severin Sadjina, Lars T Kyllingstad, Martin Rindarøy, Stian Skjong, Vilmar Æsøy, Dariusz Eirik Fathi, Vahid Hassani, Trond Johnsen, Jørgen Bremnes Nielsen, and Eilif Pedersen. Distributed co-simulation of maritime systems and operations. *arXiv preprint arXiv:1701.00997*, 2017.

Edmund Widl, Wolfgang Müller, Atiyah Elsheikh, Matthias Hörtenhuber, and Peter 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*, pages 1–6. IEEE, 2013.

DOI
10.3384/ecp1915779