FMI4j: A Software Package for working with Functional Mock-up Units on the Java Virtual Machine

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

This paper introduces FMI4j, a software package for working with Functional Mock-up Units (FMUs) on the Java Virtual Machine (JVM). FMI4j is written in Kotlin, which is 100% interoperable with Java, and consists of programming APIs for parsing the meta-data associated with an FMU, as well as running them. FMI4j is compatible with FMI version 2.0 for Model Exchange (ME) and Co-Simulation (CS). Currently, FMI4j is the only software library targeting the JVM supporting ME 2.0. In addition to provide bare-bones access to such FMUs, it provides the means for solving them using a range of bundled fixed- and variable-step solvers. A command line tool named FMU2Jar is also provided, which is capable of turning any FMU into a Java library. The source code generated from this tool provides type-safe access to all FMU variables explicitly through the API (Application Programming Interface). Additionally, the API is documented with key information retrieved from the FMU meta-data, allowing essential information such as the description, causality and start value of each variable to be seamlessly exposed to the user through the Integrated Development Environment (IDE).

Keywords: FMI, Co-Simulation, Model Exchange, JVM

1 Introduction

Recently, several research projects at NTNU Aalesund (Hatledal et al., 2015; Chu et al., 2017, 2018) and others (Skjong et al., 2017; Sadjina et al., 2017) involve co-simulation and virtual prototyping. Virtual prototyping refers to a vision where models, or *virtual prototypes*, of complex systems can be developed, tested, and amended with a trial-and-error approach. As computer technology develops it becomes possible to make an increasing part of the necessary tests based on simulations. However, as complex models often require components from several different domains, perhaps developed in separate domain-specific tools, a standard is required to fit them all together.

FMI (Blochwitz et al., 2011, 2012) is a tool independent standard to support both Model Exchange (ME) and Co-Simulation (CS) of dynamic models. The first version of the standard, FMI 1.0, was released in 2010. In 2014, version 2.0 was released, which merged the two standards and incorporated some major enhancements compared to the initial release. As such, version 2.0 is not backwards compatible with version 1.x.

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

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

FMI4j, the software package introduced in this paper, aims to simplify interaction with FMUs, and consists of easy to use software APIs for parsing and simulating FMUs on the JVM, as well as a tool for wrapping FMUs into Java libraries, named FMU2Jar. Kotlin was chosen as the implementation language as it is 100% interoperable with Java, while offering several language improvements such as null safety and less boilerplate code. From a usability perspective, invoking FMI4j code from Java feels no different than calling any other Java library.

The source code is published online under the permissive open-source MIT license and can be accessed through GitHub[1]. Here, pre-compiled FMU2Jar binaries are also available. The APIs are available on maven central[2]. Only version 2.0 and upwards are planned to be supported.

The rest of the paper is organized as follows. First some related work is given, followed by a presentation of the FMI4j software package. Finally, a conclusion and future work are given.

2 Related work

Since the release of the FMI standard, several software libraries implementing the standard have been published. An overview of such libraries for importing/invoking FMUs is given in Table. 1.

[1]https://github.com/SFI-Mechatronics/fmi4j
[2]http://mvnrepository.com/artifact/no.mechatronics.sfi.fmi4j

Table 1. Software libraries providing FMI import

					FMI support					
	Language				*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	2.0.4	BSD
FMI++		x	x[a]	x[a]	x	x	x[b]	x[b]	-	BSD
PyFMI				x	x	x	x[b]	x[b]	2.4	LGPLv3
FMPy				x	x	x	x[b]	x[b]	0.2.5	BSD
JFMI			x		x		x		1.0.2	MIT
JavaFMI			x		x	x			2.24.5	LGPLv3

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

The FMI Library (FMIL) (JModelica, 2017) and FMU SDK (QTronic, 2014), written in C and C++ respectively, provide basic access to low-level FMI functions and is often used as base for creating more high-level libraries.

FMI++ (Widl et al., 2013) is a high level utility package for FMI based on FMIL for ME and CS, written in C++, that aims to bridge the gap between the basic FMI specification and the typical requirements of simulation tools. Interfaces for Python and Java can be generated using the Simplified Wrapper and Interface Generator (SWIG). While the Python interface for Windows comes pre-built, other packages must be built from source.

PyFMI (Andersson et al., 2016) is a high-level python library for interacting with FMUs, maintained by Modelon AB. It contains co-simulation masters for simulation of weakly coupled systems and provides a connection to the simulation package Assimulo (Andersson et al., 2015), a Python package for solving first or second order explicit ordinary differential equations (ODEs) or implicit ordinary differential equations (DAEs). PyFMI is available as a stand-alone package or as part of the JModelica.org distribution.

FMPy (Dassault Systems, 2017) is a free python library from Catia Systems for simulating FMUs. FMPy supports both FMI 1.0 and 2.0 for ME and CS. Using solvers from the Sundials package, FMPy can be used to solve ME FMUs. It also features both a command line utility and a GUI for running and presenting simulation results. FMPy and PyFMI may seem very similar, however there is a major difference in that FMPy is implemented in pure Python, whereas PyFMI acts as a wrapper for FMIL, with additional high-level features.

JFMI (Broman et al., 2013b) is a low-level wrapper for FMI 1.0 for CS and ME. The latest version, 1.0.2, was released in 2013. Although the library supports both FMI-CS and FMI-ME, a flexible solving mechanism for FMI-ME is not provided.

JavaFMI (Cortes Montenegro, 2014) is a set of components for working with the FMI standard using Java, developed by SIANI institute (Las Palmas University). JavaFMI is still actively maintained and offers cross platform support for FMI version 1.0 and 2.0 for CS. A neat feature of JavaFMI is the ability to export Java code as CS FMUs.

While several FMI implementations exist, also for the JVM. Only JavaFMI is maintained, however it lacks FMI for ME support. It could be argued that FMI++ is available on the JVM by means of SWIG bindings, however, the library must be built from source which is not straightforward and requires a number of native dependencies.

As such, it can be argued that there is still room for an alternative, easy to use FMI implementation for the JVM that supports both CS and ME FMUs.

3 FMI4j

This section introduces FMI4j, a software package for working with Functional Mock-up Units on the JVM, developed by researchers at NTNU Aalesund. It is implemented from scratch in Kotlin and provides a high-level API for interacting with FMUs on the JVM (Java, Scala, Groovy, Kotlin etc.) that implements FMI 2.0 for CS and/or ME. When provided with an solver, FMI4j is able to solve ME FMUs. Such instances share a common interface with ordinary CS FMUs, that expose the most important FMI functions related to stepping a FMU forward in time.

Furthermore, FMI4j through the FMU2Jar tool is, to the best of the authors knowledge, the only publicly available software that utilizes the provided meta-data in an FMU in order to generate a high-level API tailored towards it. E.g provide type-safe and documented access to named variables directly through the API.

The different components available in the package is:

1. *fmi-modeldescription* - A library for parsing the meta-data found in the *modelDescription.xml* located within an FMU.
2. *fmi-import* - A library for loading and running FMUs on the JVM. Supports FMI 2.0 for CS and ME.
3. *FMU2Jar* - A command line tool for turning an FMU into a Java library (.jar).

FMU2Jar is dependent on fmi-import, which again depend on fmi-modeldescription. Artifacts from both libraries are hosted on *The Central Repository*[3] hosted by Sonatype. A collection of the most notable FMI4j classes are shown in Figure. 1, some of which are described in more detail in the following sections.

3.1 fmi-modeldescription

fmi-modeldescription is a lightweight API for parsing the meta-data found in the *modelDescription.xml* located inside an FMU. Useful when only static information about the FMU is required. For instance if you only want to display static information about the FMU in a web-app or when generating source code tailored towards a particular FMU, as in the case for FMU2Jar.

FMI4j can parse the model-description given both a file and URL reference to the FMU location. It can also handle raw XML input. Usage is demonstrated in Listing. 1. For brevity, code snippets are provided in Kotlin.

As seen in Figure. 1 there are several different interfaces representing the model-description. The *CommonModelDescription* interface represents common meta-data found in both CS and ME FMUs, while the *SpecificModelDescription* interface contain additional common information found in the <*ModelExchange*> and <*CoSimulation*> XML elements for ME and CS FMUs respectively. Furthermore, the *ModelExchangeModelDescription* and *CoSimulationModelDescription* interfaces contains type-specific information located within the same entries.

Listing 1. Parsing the model-description file from an FMU.

```
File fmuFile = File("path/to/fmu.fmu")

//includes common FMI entries only
val md = ModelDescriptionParser.parse(
    fmuFile)

//includes also CS specific entries
val cs_md = md.
    asCoSimulationModelDescription()

//includes also ME specific entries
val me_md = md.
    asModelExchangeModelDescription()
```

3.2 fmi-import

fmi-import is responsible for loading and simulating FMUs. It relies on fmi-modeldescription for parsing and Java Native Access (JNA) for invoking the native FMI functions written in C. For integration of differential equations, it relies on the Apache Commons Math library (Apache, 2017).

The API for reading and writing variables is given in Listing. 2 and 3 respectively. For convenience, FMU variables can be accessed through the *ScalarVariable* instance representing the variable entry from the XML. As seen, variables can also be accessed in a more FMI idiomatic way using the *variableAccessor* handle found within an FMU implementation.

Listing 2. Read API.

```
val instance: FmiSimulation = ...
val speed: Double
    = instance.variableAccessor
    .readReal("speed")
```

Listing 3. Write API.

```
val instance: FmiSimulation = ...
val speedVariable: RealVariable = ...
val status: FmiStatus
    = speedVariable.write(1.0)

// or
val status: FmiStatus
    = instance.variableAccessor
    .writeReal("speed", 1.0)
```

A description of some of the most notable classes found within the fmi-import module are given below.

- *Fmu* - Represents an FMU file on disk. Responsible for extracting the FMU, and acts as a factory for new FMU instances. This allows extracted FMU content to be re-used across instances. On JVM shutdown, it will handle any necessary clean-up related to previously instantiated FMU instances and will also delete the extracted FMU contents.
- *FmuInstance* - Represents a generic FMU instance, exposing some of the most common functions.
- *FmiSimulation* - Extends the *FmuInstance* interface with time stepping. Common interface for CS FMUs and self-integrating ME FMUs.
- *AbstractFmuInstance* - Base class for all implemented FMU classes. Wraps the model description and a handle to the underlying native code belonging to the loaded FMU. Also contains common boilerplate code.
- *CoSimulationFmuInstance* - Represents a CS FMU instance. Implements the *FmiSimulation* interface. Example usage is given in Listing. 4. Implements the FMI extension for predictable step sizes proposed in (Broman et al., 2013a), enabling step-size negotiation between FMUs. More specifically, the extension adds the capability flag *canProvideMaxStepSize* and a CS specific C procedure, *fmiGetMaxStepSize*, which is an upper bound on the step-size that the FMU can accept.
- *ModelExchangeFmuInstance* - A bare-bones class for interacting with instances of ME FMUs. The responsibility of solving the FMU is left to the user, as the class simply provides access to the underlying FMU functions. Instantiated as seen in Listing. 5.
- *ModelExchangeFmuStepper* - Wraps an instance of a *ModelExchangeFmuInstance*, while implementing

[3] https://search.maven.org/

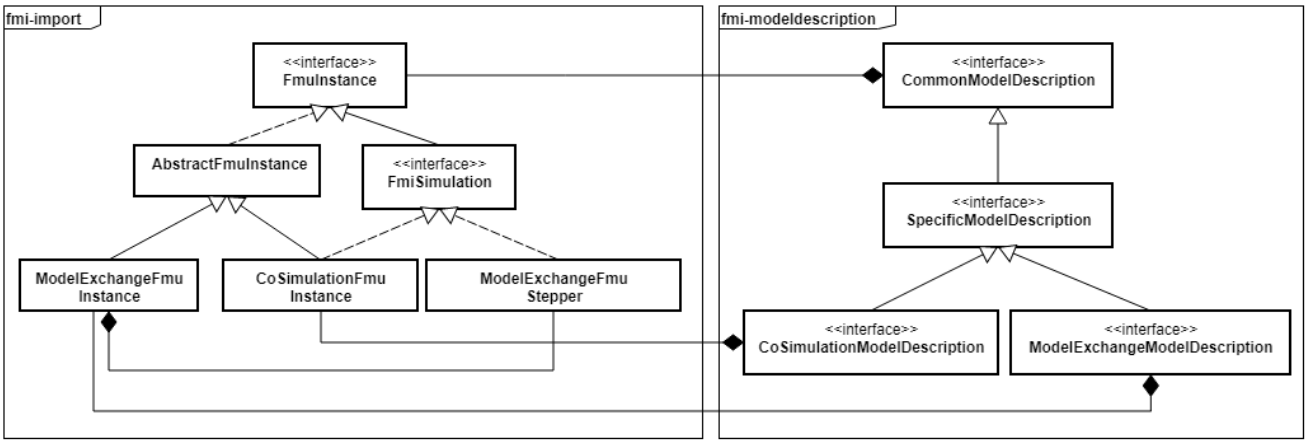

Figure 1. Simplified UML view of core FMI4j classes.

the *FmiSimulation* interface. Allows ME FMUs to be treated similar to CS FMUs. As seen in Listing. 6, it is instantiated very similarly to the *ModelExchangeFmu*, although a solver is required. For this purpose, FMI4j comes bundled with the Apache Commons Math package, which includes a range of both fixed and adaptive step-size solvers. A complete overview of the available solvers is given in Table. 2 and 3.

Listing 4. Loading and running an CS FMU

```
val fmuFile = File("path/to/fmu.fmu")
val slave = Fmu.from(fmuFile)
            .asCoSimulationFmu()
            .newInstance()

// assign start values here

slave.init() //throws on error

val dt = 1.0/100
val stop = 10.0
while (slave.currentTime < stop) {
    slave.doStep(dt)
}
slave.terminate()
```

Listing 5. Instantiating an ME FMU.

```
val file = File("path/to/fmu.fmu")
val slave = Fmu.from(file)
            .asModelExchangeFmu()
            .newInstance()
```

Listing 6. Instantiating an self-integrating ME FMU.

```
...
val solver = EulerIntegrator(1E-3)
val slave = Fmu.from(file)
            .asModelExchangeFmu(solver)
            .newInstance()
```

Table 2. Fixed-step solvers available in the Apache Commons Math package.

Name	Integration Order
Euler	1
Midpoint	2
Classical Runge-Kutta	4
Gill	4
3/8	4
Luther	6

Table 3. Adaptive step-size solvers available in the Apache Commons Math package.

Name	Order	Error Estimation Order
Higham and Hall	5	4
Dormand-Prince 5	5	4
Dormand-Prince 8	8	5 and 3
Gragg-Bulirsch-Stoer	variable	variable
Adams-Bashforth	variable	variable
Adams-Moulton	variable	variable

3.3 FMU2Jar

FMU2Jar is a command line tool for packaging an FMU into a Java library, allowing the FMU to be used as any other Java library. The generated library also exposes all variables from the FMU through a type-safe API. That is, named functions for getting and setting typed variable values are generated for each accessible variable in the FMU. These are documented with information retrieved from the associated entry in the model-description. This makes it easier to use the FMU, as all variables and associated documentation can be browsed from within an IDE, as seen in Figure. 2. Also, variables are grouped by causality for easier look-up. Both CS and ME FMUs are supported, with ME FMUs being wrapped as CS FMUs and subsequently solved using the solver provided on initialization, as seen in Listing. 7.

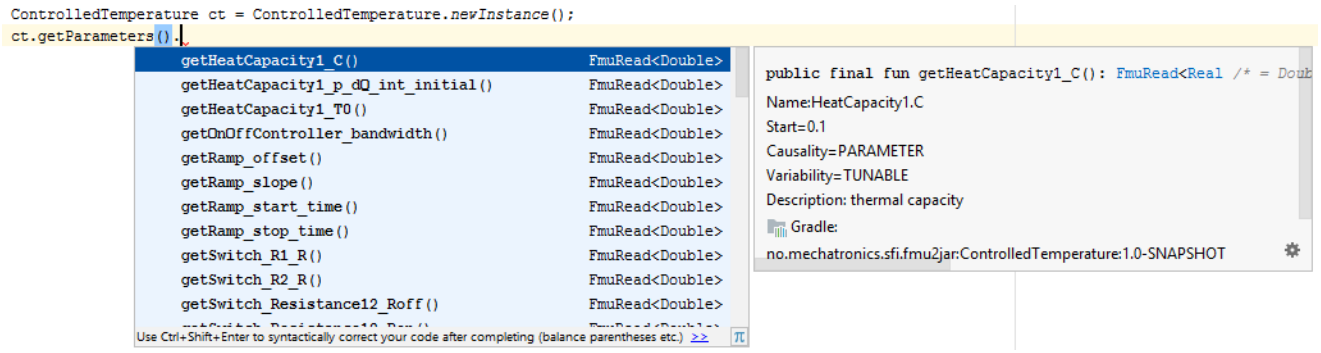

Figure 2. FMU named *ControlledTemperature* wrapped as a Java library using FMU2jar, then imported into IntelliJ IDE. From within the IDE, the user can browse and read documentation on all available variables.

Listing 7. Instantiating both a CS and a self-integrating ME FMU generated by FMU2Jar.

```
// Given an FMU that supports
// both CS & ME:
// First instantiate a CS FMU
ControlledTemperature
    .newInstance()
    .use { fmu ->
        ...
    }

// and then a self-integrating ME FMU
val solver = EulerIntegrator(1E-3)
ControlledTemperature
    .newInstance(solver)
    .use { fmu ->
        ...
    }
```

The Command Line Interface (CLI) is shown in Listing. 8. When supplying *-mavenLocal* as an argument, a maven artifact is published to the local maven repository (.m2 folder). This allows the user to easily include the library in a software project using a build system such as Apache Maven or Gradle. The user may also save the generated .jar into a specified folder and reference it explicitly.

Listing 8. FMU2Jar CLI

```
-fmu <arg>     Path to the FMU
-help          Prints this message
-mavenLocal    Should the library be
    published to maven local?
-out <arg>     Specify where to copy the
    generated .jar
```

FMU2Jar is most useful when working with FMUs programmatically, as its advantages such as variable look-up, type-safe variable access and in-IDE documentation has little to no function in common GUI based simulation environments such as OpenModelica, SimulationX, etc.

3.4 Performance

Table. 4 shows how FMI4j compares to some of the other FMI libraries in terms of performance. The table shows the time required in order to step two different test FMUs forward in time. Both FMUs implements the CS standard and was downloaded from the official SVN repository for test FMUs. A step-size of $1E-2$ and target time equal to 100 seconds is used for the *bouncingBall.fmu* exported from FMUSDK, while a step-size of $1E-5$ and target time equal to 12 seconds was used for the *TorsionBar.fmu* exported from 20Sim. For each time-step, a read call on a real-valued output variable is performed. The tests were performed on a i7-4770 CPU running Windows 10. From the results we see that the native FMIL library is faster than FMI4j by a good margin. This is to no surprise as there is some overhead related to calling native functions from Java (Kurzyniec and Sunderam, 2001). Performance wise, FMI4j and JavaFMI are practically identical as they both relies on JNA to handle native code execution. FMPy, which runs in an interpreted language, is slower in both test cases.

Table 4. Performance comparison

	Execution time [ms]	
Library	*bouncingBall.fmu*	*TorsionBar.fmu*
FMIL	4	2801
JavaFMI	54	5843
FMI4j	53	5979
FMPy	60	9662

4 Conclusion and Future Work

This paper presents a high-level software package for working with FMUs on the JVM platform. It includes both a library for parsing the model-description file and also for running the FMUs, as well as a tool for wrapping FMUs as Java libraries, named FMU2Jar. Both FMI 2.0 for Co-simulation and Model-Exchange is supported. Currently, it is the only library implemented for the JVM to support version 2.0 of the ME standard. Using one of the bundled solvers from the Apache Commons Math library, such FMUs can be solved directly by the library.

The FMU2Jar tool makes it easier to work with a specific FMU by wrapping it as a Java library, and generate maven artifacts for it, which facilitates easy integration

with popular build tools such as *Maven* and *Gradle*. Furthermore, variables are exposed through the API as type-safe method calls with documentation retrieved from the model-description.

Recently, the FMI steering committee released a feature list for version 3.0 of the FMI standard (FMI steering committee, 2018). As a future work, we aim to support this standard some time after it has been officially released.

In the future FMI4j may also include the option to export FMUs from Java byte-code.

A request to list FMI4j on the official FMI tools page has been submitted, and is pending approval. If or when new features are added to the software, the capabilities shown in this entry will be updated accordingly.

5 Acknowledgement

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

References

Christian Andersson, Claus Führer, and Johan Åkesson. Assimulo: A unified framework for ode solvers. *Mathematics and Computers in Simulation*, 116:26–43, 2015.

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.

Apache. Apache commons math, 2017. URL `http://commons.apache.org/proper/commons-math/`. (Date accessed 23-June-2018).

Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, H Elmqvist, A Junghanns, J Mauß, M Monteiro, T Neidhold, D Neumerkel, et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 063, pages 105–114. Linköping University Electronic Press, 2011.

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.

David Broman, Christopher Brooks, Lev Greenberg, Edward A Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for co-simulation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 2. IEEE Press, 2013a.

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, 2013b. URL `https://ptolemy.eecs.berkeley.edu/java/jfmi/`. (Date accessed 23-June-2018).

Yingguang Chu, Lars Ivar Hatledal, Houxiang Zhang, Vilmar Æsøy, and Sören Ehlers. Virtual prototyping for maritime crane design and operations. *Journal of Marine Science and Technology*, pages 1–13, 2017.

Yingguang Chu, Lars Ivar Hatledal, Vilmar Æsøy, Sören Ehlers, and Houxiang Zhang. An object-oriented modeling approach to virtual prototyping of marine operation systems based on functional mock-up interface co-simulation. *Journal of Offshore Mechanics and Arctic Engineering*, 140(2):021601, 2018.

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

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

FMI steering committee. Fmi version 3.0: Status, 2018. URL `https://fmi-standard.org/downloads/`. (Date accessed 23-June-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 56th Conference on Simulation and Modelling (SIMS 56), October, 7-9, 2015, Linköping University, Sweden*, number 119, pages 123–129. Linköping University Electronic Press, 2015.

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

Dawid Kurzyniec and Vaidy Sunderam. Efficient cooperation between java and native codes–jni performance benchmark. In *The 2001 international conference on parallel and distributed processing techniques and applications*. Citeseer, 2001.

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.

Stian Skjong, Martin Rindarøy, Lars T Kyllingstad, Vilmar Æsøy, and Eilif Pedersen. Virtual prototyping of maritime systems and operations: applications of distributed co-simulations. *Journal of Marine Science and Technology*, pages 1–19, 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.