Doctoral theses at NTNU, 2021:81

Lars Ivar Hatledal

Protocols and Standards for Simulation and Co-simulation

For Demanding Maritime Operations

Doctoral thesis

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of
Philosophiae Doctor
Faculty of Engineering
Department of Ocean Operations and Civil
Engineering

Norwegian University of
Science and Technology

Lars Ivar Hatledal

Protocols and Standards for Simulation and Co-simulation

For Demanding Maritime Operations

Thesis for the Degree of Philosophiae Doctor

Trondheim, March 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Ocean Operations and Civil Engineering

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Engineering
Department of Ocean Operations and Civil Engineering

© Lars Ivar Hatledal

ISBN 978-82-326-5777-3 (printed ver.)
ISBN 978-82-326-5231-0 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2021:81

Printed by NTNU Grafisk senter

Abstract

There is a strong demand for innovation and efficiency within operations, life cycle services, and design of maritime systems. Modern vessels operate increasingly autonomously through strongly interacting sub-systems. These systems are dedicated to a specific, primary objective of the vessel or may be part of the general essential ship operations. The sub-systems exchange data and make coordinated operational decisions, ideally without any user interaction. The task of designing, operating, and integrating life cycle services for such vessels is a complex engineering task that requires an efficient development approach, which must consider the mutual interaction between the inherent multi-disciplinary on-board sub-systems. Digitalization thus has become a key aspect of making the maritime industry more innovative, efficient, and fit for future operations.

However, no one simulation tool is suitable for all purposes and the plethora of modeling tools within different disciplines exists for very good reasons. Issues related to integration of heterogeneous systems and hardware, memory, and CPU utilization makes implementing complex-cyber-physical systems, like vessels, in a monolithic or centralized manner undesirable. Co-simulation alleviates this issue, allowing different sub-systems to be modeled independently, but simulated together. 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. A crucial point is that it allows users to simulate models exported from different tools in a unified manner. Compared to more traditional monolithic simulations, co-simulation encourages re-usability, model sharing, and fusion of simulation domains.

Co-simulation can be expanded into the realm of digital twins by feeding sensor data measured from the real world into the models, which in turn closes the loop by providing actionable feedback. A digital twin can be defined as a virtual representation of a physical asset enabled through data and simulators for real-time prediction, optimization, monitoring, controlling, and improved decision making. As the digital twin mimics its physical counterpart, it can be used to estimate a vessels performance before running any tests in the real world. This not only offers flexibility, but also cuts down costs to a great extent. These proxies of the physical world will help companies in the maritime industry in developing enhancements to existing products, operations, and services, and can even help drive business innovation.

This dissertation aims to drive adoption of co-simulation standards and development of use-cases by providing software that makes co-simulation simpler and more intuitive. This includes enabling technology for building standard-conforming models and systems, and subsequent tools for simulating them. The case studies presented show the effectiveness of the proposed approach.

Acknowledgment

I conducted the research for this dissertation at the Norwegian University of Science and Technology in Ålesund within the Department of Ocean Operations and Civil Engineering (IHB). My Ph.D. position relied primarily on the project "SFI Offshore Mechatronics" for funding and the "SFI MOVE" and "Digital Twins For Vessel Life Cycle Service" projects provided ancillary support. These projects correspond to the Research Council of Norway grant nos. 280703, 237896, and 237929, respectively.

Beyond these funding sources, I'm grateful for the opportunity to pursue a Ph.D. degree under the supervision of Prof. Houxiang Zhang, Prof. Geir Hovland, and Arne Styve. The guidance and support I received during the last three and a half years are highly appreciated. Especially, I would like to thank my main supervisor, Prof. Houxiang Zhang, for shaping me into an independent researcher. Also, I would like to thank Prof. Hans Petter Hildre and Siri Schulerud for their administrative support.

Thanks to my colleagues at the Intelligent Systems Lab at NTNU in Ålesund. It has been a privilege working with you. Especially, I would like to thank Dr. Guoyuan Li and Robert Skulstad with their help on Paper A5. For valuable input with regards on this paper, I'd also like to thank Martin Rindarøy and Stian Skjong from SINTEF Ocean.

I'd also like to thank my friend, and former colleague, Dr. Yingguang Chu at SINTEF Ålesund for his help in realizing Paper A6, and also Frédéric Collonval for his help in making the PythonFMU package presented in Paper A4 more *pythonic*.

I've also learned a lot while working on the OSP project together with the OSP participants. In particular I'd like to mention Levi Jamt, Kristoffer Eide and Halvor Platou from DNV-GL, and Lars Kyllingstad from SINTEF Ocean.

A big thanks goes to the Offshore Simulator Centre (OSC) for providing a visualization friendly version of the Gunnerus 3D model used to enhance some of the simulations performed.

Finally, I give my most warmest thanks to my beloved partner, Caroline Remø Dahl, and our mesmerising twin sons Aron and Iver. You have truly shown me the meaning of life. Also, I give a special thanks to my supporting family and friends. Mom and dad, I love you. Making a run for a PhD title during a twin birth, COVID-19 and house building has been quite the experience.

Contents

Abstract i

Acknowledgment iii

List of Abbreviations vii

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Background and motivation 1
1.2 Objectives 4
1.3 List of publications 6
1.4 Structure of the Dissertation 7

2 Co-simulation to support demanding maritime operations 9

2.1 Co-simulation fundamentals 9
2.2 Literature review 12
2.2.1 Benefits and challenges of co-simulation 12
2.3 Scope of work 13
2.4 Model accumulation, limitations, and assumptions 15
2.4.1 Maritime reference models 15

3 Fundamental technologies for co-simulation 17

3.1 Open standards for co-simulation 17
3.1.1 The Functional Mock-up Interface 17
3.1.2 High Level Architecture 18
3.1.3 System Structure and Parameterization 18
3.1.4 Open Simulation Platform - Interface Specification 19
3.1.5 Open Simulation Platform - System Structure 19
3.1.6 Distributed Co-simulation Protocol 19
3.1.7 Standards considered in this work 19
3.2 FMI-based co-simulation libraries and tools 20
3.2.1 Overview of existing libraries 20

3.2.2 FMI4j . . . 20
3.2.3 FMI4cpp . . . 21
3.2.4 PythonFMU . . . 21
3.2.5 SSPgen . . . 22
3.2.6 FMU-proxy . . . 23

4 Open-source co-simulation platforms **27**
4.1 Overview of open-source FMI based co-simulation platforms . . . 27
4.2 The open simulation platform . . . 27
4.2.1 libcosim . . . 28
4.2.2 cosim4j . . . 29
4.3 Vico . . . 29
4.3.1 FMI & SSP support . . . 30
4.3.2 3D visuals . . . 31
4.3.3 Scenarios . . . 31
4.4 Differences between the OSP and Vico . . . 32

5 Case studies **33**
5.1 Accuracy and performance benchmark . . . 33
5.2 Co-simulation of the RV Gunnerus . . . 36

6 Conclusion **47**
6.1 Summary of contributions . . . 47
6.2 Directions for future work . . . 49

References **51**

Appendix **57**

A Paper A1 **59**

B Paper A2 **67**

C Paper A3 **77**

D Paper A4 **91**

E Paper A5 **97**

F Paper A6 **113**

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CS	Co-simulation
DSL	Domain Specific Language
ECS	Entity-Component-System
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
HLA	High Level Architecture
HTTP	Hypertext Transfer Protocol
IPR	Intellectual Property Rights
JSON	Javascript Object Notation
JVM	Java Virtual Machine
NTNU	Norwegian University of Science and Technology
ME	Model Exchange
OSP	Open Simulation Platform
OSP-IS	OSP System Structure
OSP-IS	OSP Interface Specification
RO	Research Objective
RMSE	Root Mean Square Error
RV	Research Vessel
RPC	Remote Procedure Call
SSP	System Structure and Parameterization
TCP/IP	Transmission Control Protocol / Internet Protocol
YAML	YAML Ain't Markup Language
XML	Extensible Markup Language

List of Figures

1.1 Open software framework and system architecture. . . . 2

1.2 A plausible development procedure of digital twins system for the maritime industry. . . . 4

2.1 Key properties of co-simulation. . . . 9

2.2 The vessel model depicted in the figure is an aggregate of several different sub-components, which are integrated and solved together using co-simulation. *Vessel sub-component figures courtesy of the Virtual Prototyping of Maritime Systems and Operations project (Research Council of Norway, grant nr. 225322).* . . 10

2.3 The components of a digital twin. . . . 11

2.4 Co-simulation layers considered in this work. . . . 14

2.5 Interconnection of published papers in the thesis. . . . 15

3.1 FMI for model exchange (from [1]). . . . 17

3.2 FMI for co-simulation (from [1]). . . . 17

3.3 Overview of the initial FMU-proxy structure. . . . 24

3.4 Overview of the current FMU-proxy structure. . . . 24

4.1 High-level overview of the ECS architecture. . . . 30

5.1 Illustration of the quarter-truck system. . . . 33

5.2 Wheel response when simulated at 100hz. . . . 35

5.3 Wheel response when simulated at 1000hz. . . . 35

5.4 Detailed view of the the first second of simulation presented in Fig. 5.2 (100hz). 36

5.8 Starboard view of the RV Gunnerus. . . . 36

5.5 Chassis response when simulated at 100hz. . . . 37

5.6 Chassis response when simulated at 1000hz. . . . 37

5.7 Performance of the various tools when considering the presented quarter-truck system. Simulation time=1000s, step-size=0.001s, number of runs=15. . . . 38

5.9 Diagram showing the logical relationship of the components involved with the case study presented in Paper A5. . . . 39

5.10 Connections between the components used in the Gunnerus system presented in Paper A5. . . . 40

5.11 Northeast plot showing the trajectory and heading of the vessels during the experiment. The blue arrow indicates the wind direction according to north and normalized magnitude of the speed. 41
5.12 Twin surge speed with respect to course changes by the Gunnerus. 42
5.13 Power consumption comparison. The power output shown is the sum of the two azimuths. 43
5.14 Simulation with synchronized video overlay of the real operation. 44
5.15 Demonstration of a vessel path following simulation running in Vico with 3D visualization and plotting enabled. 45
5.16 Performance of the various tools when considering the presented Gunnerus trajectory tracking scenario. Simulation time=1000s, step-size=0.05s, number of runs=15. 45

List of Tables

2.1 OSP reference models utilized by the Gunnerus case study presented in Paper A5. 16

3.1 Software libraries providing FMI import. 20

4.1 Open-source co-simulation platforms supporting the FMI. 28

5.1 Input and output variables of the quarter-truck models used for connections. . . 34
5.2 Summary of tools included in the case study. 34
5.3 Root mean square error of the computed vertical displacement of the wheel. . . 35
5.4 FMUs involved in the case study utilizing the RV Gunnerus presented in Paper A5. 37

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. Co-simulation refers to an enabling technique, where different sub-systems making up a global simulation are 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 output [2]. This dissertation is heavily application-oriented and focuses mainly on how to use co-simulation as a tool for building and simulating complex heterogeneous systems, like vessels, creating tools that make the process easier along the way.

1.1 Background and motivation

There is a strong demand for innovation and efficiency within operations, life-cycle services, and design of maritime systems. Modern vessels operate increasingly autonomously through strongly interacting sub-systems. These systems may be dedicated to a specific, primary objective of the vessel or may be part of the general essential ship operations. The sub-systems exchange data and make coordinated operational decisions, ideally without any user interaction. The task of designing, operating, and integrating life cycle services for such vessels is a complex engineering task that requires an efficient development approach, which must consider the mutual interaction between the inherent multi-disciplinary on-board sub-systems. Digitalization thus has become a key aspect of making the maritime industry more innovative, efficient, and fit for future operations [3, 4].

The concept of digital twins, characterized by the high fidelity with which they mimic their physical counterpart, provides a potential solution for the next generation of advanced ships. A digital twin can be defined as a virtual representation of a physical asset enabled through data and simulators for real-time prediction, optimization, monitoring, controlling, and improved decision making [5]. The digital twin should be able to take advantage of all digital information available for an asset, such as the system and data information models, 3D models, mathematical models, dependability models, condition and performance indicators, and data analytics. Digital twin technology, together with data-driven prognostics and health management systems, allows analysis of data and monitoring of maritime systems to detect faults before they occur through condition monitoring and predictive maintenance [6], and plan for the future by using simulations. However, issues related to integration of heterogeneous systems and hardware, memory, and CPU utilization makes implementing such a digital twin in a monolithic or centralized manner undesirable. Co-simulation as a technology mitigates some of these issues by facilitating an inherently distributed black-box modeling and simulation approach. Furthermore, as maritime systems such as ships are becoming increasingly complex and consist of many sub-systems from different engineering domains, traditional simulation approaches are too inflexible, too costly, and too inefficient [7].

The maritime industry will benefit from co-simulation as a tool for virtual prototyping [8, 9, 10] and as an enabler for digital twins [11]. These proxies of the physical world will help maritime companies in developing enhancements to existing products, operations, and services, and can even help drive innovation. In [12], the authors details additional benefits for the maritime industry as a whole.

The goal of this work is to develop auxiliary tools and an open framework to assist the development of digital twins that will aid users to more easily develop components or subsystem models, and combine them in a full system for the purpose of maritime industry design, operation, service, and maintenance, as shown in Fig. 1.1. The core framework, allowing simulations to be designed and carried out, as well as a number of reference models are open-sourced. Additional models, applications, tools and services designed for or built on top of the core framework may or may not be open-source depending on the intent of the provider.

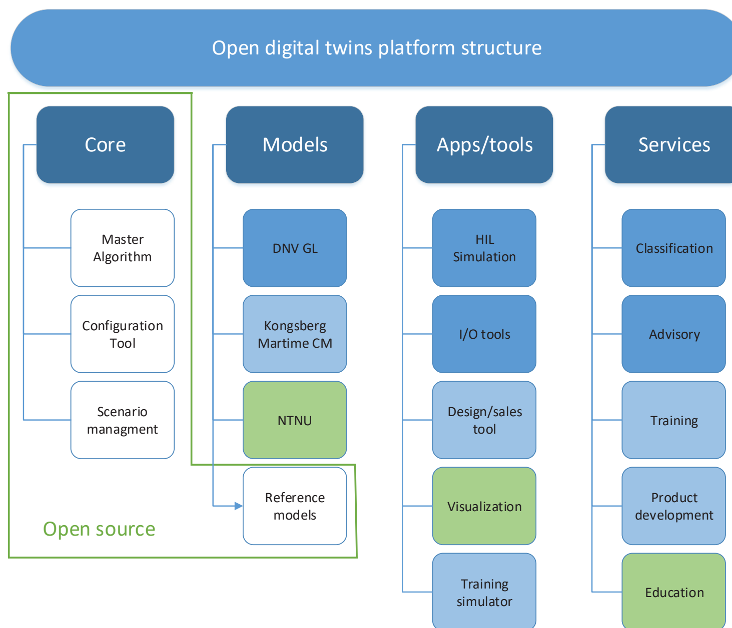


Figure 1.1: Open software framework and system architecture.

The flexibility of simulator frameworks available on the market today with respect to integration and mixing of models from different sources is very limited, and coupling with a physics engine may not be possible. Limited or no support for existing standards is also a major challenge in existing solutions. This prompts the first three research questions of this dissertation:

- ***Which standards related to co-simulation are applicable?***
- ***How can components from different sources be combined effectively?***

- ***How can co-simulation be made more accessible?***

In order to answer these questions it is necessary to investigate existing co-simulation standards and select the ones that are identified as the most promising in terms of enabling co-simulations for demanding maritime operations. Furthermore, it is necessary to establish what co-simulation can and cannot do, thus the following research question is raised:

- ***Which drawbacks comes with co-simulation?***

Some of the main challenges related to combining components from different sources are related to accuracy & stability [13, 14], timing & data-exchange, and security & protection of Intellectual Property Rights (IPR) [15]. This leads to the next research question:

- ***How to facilitate accurate and stable co-simulations?***

This area is well studied in literature; see for example [16, 17, 18, 19, 13] for work on master algorithms and [14] on stability. Thus further work may draw on the existing body of knowledge.

In order to ensure simulation scalability and model interoperability, simulation models may be distributed and processed within a dedicated process, either locally or remotely. Remote model execution also provides secondary benefits, as it alleviates the security concerns of the user and further protects the IPR of the model owner. However, existing solutions either lacks this ability or forces the user to commit fully to them, often in a non-transparent way. This leads the following research questions:

- ***How might we facilitate seamless distributed model execution?***

Ideally, the user should not be required to possess deep knowledge of the network stack in order to run distributed simulations, nor should the user be forced to either run all models in a single process or run all models in a distributed fashion.

Building upon the previous research questions, a more practical question related to usability emerges:

- ***How can a co-simulation approach be used to support maritime digital twins and simulation of demanding maritime operations?***

To address this, a plausible case-study must be developed to verify its applicability.

Moreover, most frameworks today, including 20Sim, SimulationX, MATLAB and other typical engineering tools, are focused on modeling behavior and behavior only. This makes it difficult, if not impossible, to change the fidelity or characteristics of the simulation in an intuitive way while at the same time preserving state. This prompts the following research question:

- ***How can simulation behavior and state be separated, effectively and intuitively?***

To address this question it is imperative to investigate and test different software architectures. The architecture should not only make it possible, but also provide the necessary usability to be practically viable.

Existing tools might not be very flexible when it comes to presenting and controlling a simulation. If possible at all, interaction and presentation of simulations is provided within that software, with limited or no options for custom access/control from the outside. This leads to the final research question:

- ***How can the simulation be presented to the user in an intuitive way?***

Answering these questions should lead to a sound conceptual framework as illustrated by Fig. 1.2, which should aid the development and simulation of maritime equipment, operations, and digital twins, thereby benefiting the surrounding maritime industry.

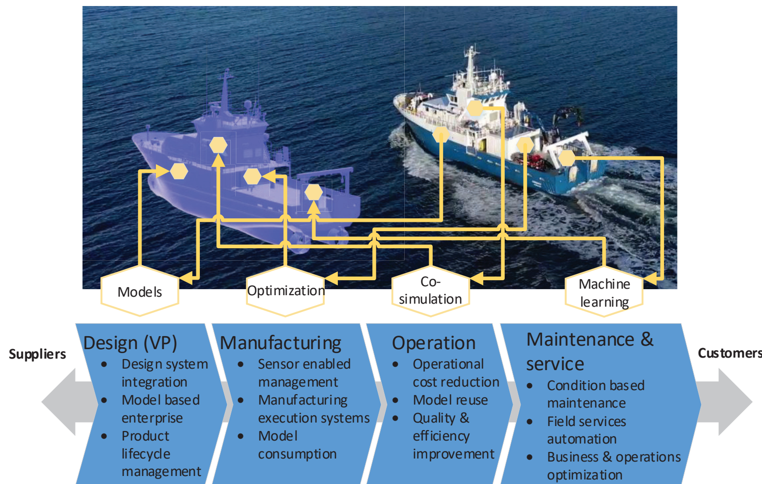


Figure 1.2: A plausible development procedure of digital twins system for the maritime industry.

1.2 Objectives

In seeking to answer the above research questions, this dissertation seeks to obtain the following research objectives (ROs):

✓ **RO1: Making co-simulation more accessible and user-friendly through the development of supplementary software packages.**

It is very useful that multiple standards for co-simulation has been developed. However, standards require software to actually implement them. While such implementations do exists for most sets of standards, they might be deficient, difficult to use, or only available in a limited number of languages. Therefore it is important to establish supplementary software packages to support further developments. The development of these packages are covered in part by papers A1, A2, A3, and A4. Together they act as fundamental enablers for the development and usage of the framework proposed by the next research objective:

✓ **RO2: Propose an open-source simulation framework focusing on co-simulation and digital twin technology, with the main goal of supporting maritime use cases.**

The goal is to create a generic open-source simulator framework, with a strong focus on maritime use-cases that can provide full flexibility for the user in terms of what to simulate, how to simulate, and how to present and control simulations. The solution should provide the ability to define simulation scenarios where the user can easily select/change which behavior models to be used. Behavior models may be be Functional Mock-up Units (FMUs) generated with third-party tools, sensor data from hardware devices, or provided as source code in selected languages.

The quality of any co-simulation framework is largely dependent on the provided master algorithms. However, implementing every state-of-the art master algorithm imaginable is complex, time-consuming, error-prone, and tedious work. In order to raise the quality of the overall system during the early development phases of a new framework, it might be beneficial to only offer a single or limited set of algorithms. Nonetheless, it is vital to facilitate the inclusion of such algorithms in the future. If this process is to attract outside collaborators, it should be as seamless as possible. Papers A5 and A6 covers the development and utilization of the co-simulation frameworks used in this work. However, an elaborate framework filled with advanced features loses some of its value if some of the models it is designed for cannot be run or if the system simply cannot handle the workload exerted on it in a timely manner. This leads to the following research objective:

✓ **RO3: Propose a method for enabling interoperable and scalable simulations through distributed model access.**

Obtaining a simulation model does not mean that it is necessarily viable in its current form. The model might not run on the desired platform or tool due to misaligned specifications or missing software components. Furthermore, a user may want to instantiate multiple instances of a model that lacks this ability. As the simulation grows large, scalability might also be an issue. Thus, papers A2 and A3 explore an innovative, flexible, and efficient way of solving these issues. Furthermore, improvements and refinements for this solution are laid out in Section 3.2.6.

✓ **RO4: Making co-simulations intuitively presentable to the user.**

Certain simulation tools offer little more than the ability to save time-series data to a file after the simulation has ended, while others offer built-in visuals and plotting capabilities. As long as simulation data is somehow accessible, no solution is necessarily better than any other; the purpose of the tool in question will determine what is most useful. However, the ability to present comprehensible data while a simulation is running certainly has advantages. The solution proposed in Paper A6 allows 3D visuals and 2D plots to be easily configured and shown during simulation. Furthermore, this presentation is available both locally on desktop and remotely using web technologies. To allow flexibility, advanced users may choose to implement their own hooks into the simulation, using either the Application Programming Interface (API) or remote end-point, instead of using the provided solutions.

1.3 List of publications

This thesis is based on the research published in three journal papers and three conference papers. The six papers are included in the appendix section of this thesis. In the following list of publications, the papers are listed chronologically by the date of publication, from the oldest one to the most recent.

[Paper A1] 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", *The 59th Conference on Simulation and Modelling (SIMS 59)*, vol. 153, no. 6, pp. 37–42, 2018.

This paper introduces FMI4j, a software package for dealing with FMUs on the Java Virtual Machine (JVM), which makes using and developing FMUs on this platform easier. It is the only open-source JVM library to offer support both for model exchange and co-simulation, and later revisions of the package offers much better performance than the alternatives.

[Paper A2] L. I. Hatledal, H. Zhang, A. Styve, and G. Hovland, "FMU-proxy: A Framework for Distributed Access to Functional Mock-up Units", *Proceedings of the 13th International Modelica Conference*, vol. 183, pp. 240–251, 2019.

This paper introduces FMU-proxy, a framework that aims to solve some of the practical deficiencies of the Functional Mock-up Interface (FMI) standard. In practice, an FMU may not run on a particular system due to an incompatible operating system, licensing issues, or missing software components. Furthermore, some FMUs can only be instantiated once per process. FMU-proxy solves these issues by wrapping FMUs in a server program that exposes the FMI API through language and platform independent Remote Procedure Call (RPC) interfaces over multiple protocols.

[Paper A3] L. I. Hatledal, A. Styve, G. Hovland, and H. Zhang, "A Language and Platform Independent Co-simulation Framework Based on the Functional Mock-up Interface", *IEEE Access*, vol. 7, pp. 109328-109339, 2019.

This paper expands on the work introduced in Paper A2, most notably providing the results of a performance benchmark that compares different RPC technologies in terms of performance in the context of FMU-proxy.

[Paper A4] L. I. Hatledal, F. Collonval, and H. Zhang, "Enabling Python Driven Co-Simulation Models with PythonFMU", *Proceedings of the 34th International ECMS-Conference on Modelling and Simulation-ECMS 2020*, vol. 34, no. 1, 2020.

This paper introduces PythonFMU, a framework for exporting Python code as FMI 2.0 compatible co-simulation FMUs. This work aims to lower the barrier of entry for model creators by providing an easy-to-use to use tool for exporting FMUs from source code. As Python and its vast ecosystem of libraries allows the development of models that are connected to web services or utilize machine learning, PythonFMU was specifically developed to allow data scientist with little or no background in co-simulation or software engineering to contribute with models related to the development of digital twins.

[Paper A5] L. I. Hatledal, R. Skulstad, G. Li, A. Styve, and H. Zhang, "Co-simulation as a Fundamental Technology for Twin Ships", in *Modelling Identification and Control*, vol. 41, no. 4, pp. 297-311, 2020.

This paper presents ongoing work related to the development towards a digital twin of the NTNU-owned research vessel (RV) Gunnerus. The work makes use of the FMI-based co-simulation library *libcosim*, which was developed as part of the Open Simulation Platform (OSP) initiative. The paper also introduces *cosim4j*, which makes it possible to interact with the software from within the JVM. A model of the Gunnerus was developed and described using the System Structure and Parameterization (SSP) standard, then subsequently simulated using the presented co-simulation software.

[Paper A6] L. I. Hatledal, Y. Chu, A. Styve, and H. Zhang, "Vico: An Entity-Component-System Based Co-simulation Framework", *Simulation Modelling Practice and Theory*, vol. 108, April 2021.

This paper introduces Vico, a novel co-simulation framework based on the Entity-Component-System (ECS) software architecture. This work may be regarded as the culmination of the work conducted as part of this dissertation, connecting the various software components together in one unified package. That is, PythonFMU and FMI4j may be used to aid in model development, SSPgen may be used to aid in the development of the system to be simulated, and FMU-proxy may be used to enable distributed execution of FMUs or merely as an enabling technology for adapting otherwise incompatible simulation models. Vico, which also provides a Command Line Interface (CLI), scriptable scenarios, 3D visuals, plotting, data export, and web access in one unified package, is then used to run the global simulation.

Other works

The following papers are not included in this thesis but might be considered relevant due to co-authorship and similar topics:

i Y. Chu, L I. Hatledal, V. Æsøy, E. Sören, and H. Zhang, "An Object-Oriented Modeling Approach to Virtual Prototyping of Marine Operation Systems Based on Functional Mock-Up Interface Co-Simulation", in *Journal of Offshore Mechanics and Arctic Engineering*, vol. 140, no. 2, 2018.

ii Y. Chu, L I. Hatledal, H. Zhang, V. Æsøy, and E. Sören, "Virtual prototyping for maritime crane design and operations", in *Journal of marine science and technology*, vol. 24, no. 4, pp. 754-766, 2018.

1.4 Structure of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 introduces co-simulation as a fundamental technology for enabling simulation and digital twins related to demanding maritime operations. Chapter 3 introduces established standards in the field and provides an overview of the standards selected for this work. It also presents on the various enabling tools that have been developed, expanding on the work published in Papers A1 through A4. Chapter 4 relates to Paper A5 and A6, and gives an overview of currently existing co-simulation platforms as well as an introduction to the two co-simulation platforms that have been developed, and contributed to, as part of this work. Chapter 5 presents the various case studies performed, which include a generic co-simulation case-study related to simulation accuracy and performance published in Paper A6, as well as case studies related to the research vessel Gunnerus, one of which appeared in Paper A5. Chapter 6 concludes the dissertation, summarizes the contributions, and indicates objectives for future work.

2

Co-simulation to support demanding maritime operations

This chapter introduces co-simulation as a fundamental technology for enabling effective simulation of demanding maritime operations. Effective, in this context, means that the process of creating the overall simulation should be easy enough, the simulation results should not produce unexpected results, and the run-time performance should be adequate. Finally, it should be possible to both passively and actively interact with the simulation. Passive interaction would mean interaction with tools that provides apprehensible feedback about the ongoing simulation, like plots and 3D visuals. Active interaction should allow the user to influence the simulation while it is running, in order to facilitate more than just pure data-centric use-cases.

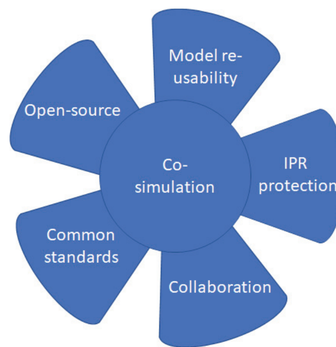


Figure 2.1: Key properties of co-simulation.

2.1 Co-simulation fundamentals

Co-simulation refers to an enabling technique, where different sub-systems making up a global simulation are being modeled and run in a (logically) 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 [20]. Key-properties affiliated with co-simulation are illustrated in Fig. 2.1. A crucial point is that co-simulation allows users to simulate models exported from different tools together. Compared to more traditional monolithic simulations, co-simulation encourages re-usability, model sharing and fusion of simulation domains. Thus, the idea of using co-simulation to simulate maritime vessels and auxiliary equipment seems promising. Modeling a cyber-physical-system (CPS) as complex as a vessel will naturally consist of components from several different domains [7]. The plethora of modeling tools within different disciplines exists for very good reasons, and there might never be a single tool that is suitable for every stage and

every branch of the design process [21]. Fig. 2.2 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 form other than a black-box model due to concerns around exposing IPR [15]. As the business logic can be distributed as binary code, the black-box approach should relieve this concern somewhat. However, additional resources bundled within the model may not be protected, so this approach may not completely hide IPR. On the other hand, it is easy to interface with black-box models that use standardized interfaces. Thus it is quite manageable to establish a remote interface to such models, which in turn makes it possible to hide unintentional IPR leaks by only exposing the API, and not the resources within.

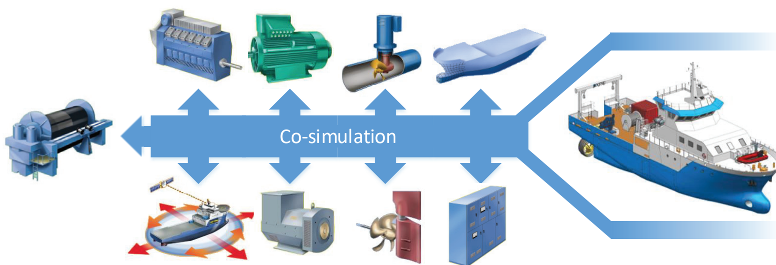


Figure 2.2: The vessel model depicted in the figure is an aggregate of several different sub-components, which are integrated and solved together using co-simulation. *Vessel sub-component figures courtesy of the Virtual Prototyping of Maritime Systems and Operations project (Research Council of Norway, grant nr. 225322).*

Co-simulation can be expanded into the realm of digital twins by feeding sensor data measured from the real world into the models, which in turn closes the loop by providing actionable feedback, as illustrated by Fig. 2.3. A crucial attribute of a digital twin model is that, since it mimics its physical counterpart, it can approximate a vessel's performance through simulation prior to running tests in the real world [9]. This not only offers flexibility; it also cuts down costs to a great extent. Performance, shipbuilding, and maintenance of the vessel are three vital areas where a digital twin acts as a valuable asset [22]. The maritime digital twin could be used to track information for all relevant parameters to define how each individual sub-module behaves over its entire useful life. Using digital twins to enhance maritime engineering could have the following advantages, and should be considered for future research.

1. ***Reduce development time and enhance production efficiency:*** Provide an integrated view on the vessel's various physical and behavioral aspects in all stages; allow simultaneous optimization of all functional performance requirements throughout the entire development cycle, from early concept to detailed engineering and commissioning; and reduce design tolerances, manufacturing uncertainties, and stochastic variabilities of vessel operation.

2. ***Improve operational flexibility and reduce cost***: Execute day-ahead and long-term planning for improved operations; and reuse the sub-system models, data, and operational sources.

3. ***Enhance the life-cycle value chain and improve the system performance and health condition***: Allow data exchange between different sub-systems, modules, and proprietary applications; enable remote access to on-board systems; assure system availability via digital twins; and utilize early warnings for short time technical support. For example to improve response time related to changing components and software updates.

4. ***Improve quality and efficiency of maritime product and operation approval and certification processes***: Combine simulation models and sensor data on an open platform; facilitate the design and verification of cyber-physical systems; and allow classification societies to replace periodical surveys with condition- and event-based inspections.

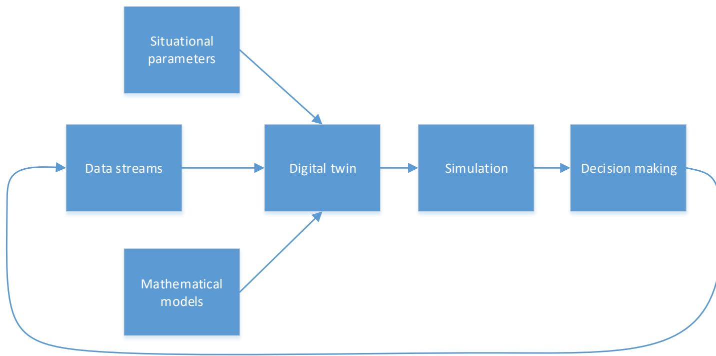


Figure 2.3: The components of a digital twin.

Co-simulation is, however, not a perfect solution to overcome all simulation challenges. In [2], the authors do a good job identifying common challenges related to co-simulation; like semantic adaptation, modular coupling, stability and accuracy, and finding a standard for hybrid co-simulation. A common issue with co-simulation is related to stability and accuracy [9, 14], in particular when strongly coupled models are involved. More advanced co-simulation algorithms with error-control, which are able to roll back and retry the simulation step from a previously known stable state using a lower time-step, may mitigate this issue [16]. However, models often do not support such capabilities and the larger the simulation the more likely it will encounter strongly coupled models. A pragmatic solution to handle the accuracy issue is to lower the time-step used to drive the simulation forward, at the cost of run-time performance. Unfortunately, this is not always sufficient and the models needs to be adapted by introducing elements such as spring-dampers systems at connection points to soften a stiff system. However, adapting the models naturally alters the characteristics of the intended simulation and thus it may not be an optimal solution. System designers then must re-think which parts of the overall simulation may be split into co-simulation units, making sure to avoid splitting parts of the simulation that are deemed too tightly coupled [21].

2.2 Literature review

A stand-alone literature review revolving around generic co-simulation technology and its application to virtual prototyping and digital twin development targeting for the maritime industry has not been published in conjunction with this thesis. Rather, literature search and reviews has been conducted and included in its individual papers. Papers A1 through A4 review the literature on enabling technologies for co-simulation; Paper A2 and A3 review the literature on the distributed part of co-simulation; and Paper A5 and A6, which are more application-oriented, include reviews of the literature on its use of the technology. Paper A5 in particular addresses digital twins.

2.2.1 Benefits and challenges of co-simulation

Benefits

- Co-simulation allows models generated from different tools and designed for different domains to be coupled and simulated together [23]. Each black box incorporates its own simulation algorithm, which is usually the most appropriate for its domain [24]. This enables heterogeneous simulations to be realized, including complex cyber-physical-systems like vessels [11].
- Co-simulation allows competing companies to share models towards realizing a common goal, without having to expose IPR thanks to the black-box modeling approach [15, 24, 9].
- The use of open standards encourages re-usability and collaboration [24], which saves cost and drives innovation [25].
- Co-simulation enables simulation-based commissioning of vessels and virtual sea trials to remove design flaws and implementation errors at an early stage [9].
- The inherent property of distribution-friendliness makes it viable to execute the model on a remote resource, potentially protecting the user from malicious attacks and further protecting the model owner's IPR by not physically re-distributing the model [15]. This also allows for large-scale parallel simulations [26, 27].
- Co-simulation naturally enables the realization of digital-twins due to the inherent distributed and component-based structure as it allows models using different run-time systems to be executed together [28].

Challenges

- In order to achieve trustworthy, reliable, and relevant simulations, one naturally needs access to sufficiently good and relevant models. Additionally, acquired models need to be coupled. This process is often a source of error even if the models themselves are working as intended [25], especially if documentation is inadequate or missing altogether [24].
- Model interfaces often exposes a large number of variables, which, when coupled with non-intuitive variable names, makes the process of connecting variables difficult to reason about. This is can be illustrated using the quarter-truck system described in Section 5.1. Here, the output variable named *p.e* from the *chassis* model should be connected to the input variable named *p1.e* belonging to the *wheel* model. Deducting this just by looking at the model definitions is not necessarily obvious.

- When designing co-simulations, there might be issues related to causality when two models, which theoretically would be a good match for coupling, have inputs and outputs flowing in the wrong direction compared to each other [9, 2]. This can be alleviated by proper communication between model designers, but there might still be an issue when using legacy models.
- Naively connecting inputs to outputs between black-boxes does not necessarily imply that the resulting behavior mimics the actual couplings of the sub-system models [24]. Actually, co-simulation often leads to less accurate simulations than monolithic ones, as each sub-system is solved individually. This changes the eigenvalues of the total system, which can cause instabilities in the local solvers [29, 14]. This can be alleviated to some extent by good master algorithms, but such algorithms might not always be available, nor applicable due to lacking model implementations or real-time requirements.
- Challenges associated with the development of digital twins relates to insufficient synchronization between the physical and the digital world to establish closed loops, a lack of high-fidelity models for simulation and virtual testing at multiple scales, lacking uncertainty quantification for such models, difficulties related to the prediction of complex systems, and challenges related to the gathering and processing of large data sets [30].

2.3 Scope of work

The realization of the proposed co-simulation framework can be divided into three main parts, as illustrated by Fig. 2.4. The first layer represents the underlying standards for co-simulation, with the second layer representing the tools and utilities that enables actual usage of these standards. Finally, the third layer represents the integration layer, which is responsible for connecting all the pieces and perform the actual co-simulation. This dissertation is not aiming to propose new competing standards for co-simulation, which would likely lead to more fragmentation of the space or might just end up in the shadows of existing and more prominent standards. Rather, the objective is to select suitable standards to use as the foundation for realizing a streamlined co-simulation experience and then develop enabling tools around them. The novelty of this work lies in the enabling technologies that allows individual models and complete co-simulation systems to be more easily and intuitively realized and later simulated. These technologies are then used to prototype a plausible and coherent framework for the development of digital twins for the maritime industry. In order to create a suitable environment for co-simulation that can provide the necessary accuracy and stability required to support trustworthy simulations, it is necessary to investigate applicable master algorithms for co-simulation. Furthermore, it is imperative that the proposed solution can accommodate such algorithms to be seamlessly implemented by anyone who wishes to do so. The solution must therefore be extensible in this regard.

The interconnection between the research objectives and the papers published are shown in Fig. 2.5. In order to fulfill RO1, a set of libraries and tools are developed that simplify, enhance, and make the use of the underlying co-simulation standards more accessible. Paper A1 and A4 both introduce libraries that makes interaction with the FMI standard easier, using the Java and Python platforms respectively. Furthermore, Paper A2 and A3 propose a framework to improve interoperability between models and tools, and to enable scalable simulations through distributed model access as suggested by RO3. These libraries are subsequently used to aid development and usage of the framework set out to be established by RO2. This work is covered by Paper A5 and A6. The framework introduced in Paper A6 also tries to accommodate the

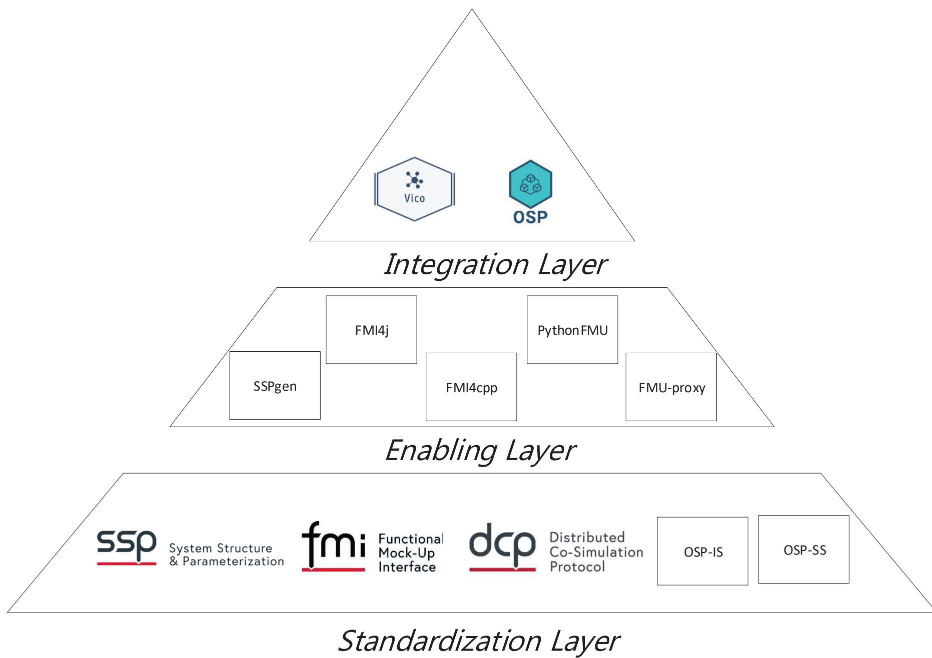


Figure 2.4: Co-simulation layers considered in this work.

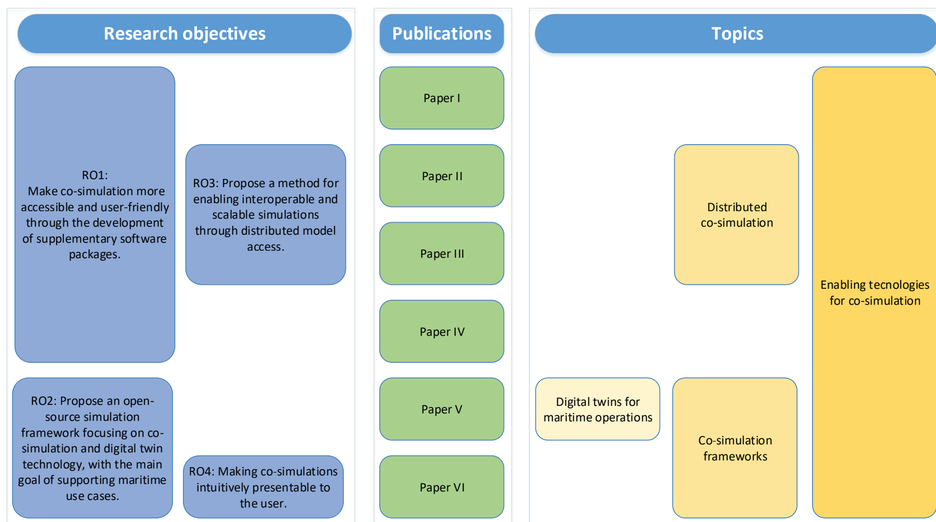


Figure 2.5: Interconnection of published papers in the thesis.

issue of presenting co-simulations as suggested by RO4, while Paper A5 looks more into how co-simulation technology can be adapted and used towards enabling digital twins for maritime operations.

2.4 Model accumulation, limitations, and assumptions

Access to relevant models are paramount in order to demonstrate the capabilities of the proposed co-simulation approach. Model development is, however, not within the scope of this work and models have been expected to already be available or to be provided by others, in a standardized and supported format. It has been assumed that the models provided are working as intended and does not provide erroneous results. In most cases models are provided as black-boxes without any attached sources. In the event that a model is not working as expected or the model interface needs adjustments, it must be updated by the body that supplied it. This might not be possible, or it could be time consuming. However, using the tools developed as part of this work, it has been possible to create supplementary models to be used in conjunction with the already provided models.

2.4.1 Maritime reference models

One of the goals of the OSP project was to produce a set of publicly available maritime reference models. These models have been published online, and an overview can be found at the following location: https://open-simulation-platform.github.io/demo-cases[1]. The models have been developed by different suppliers, including SINTEF, NTNU, Kongsberg and DNV-GL, using a diverse range of tools, targeting different versions of the FMI standard. Most relevant to this work are the models used in relationship with the RV Gunnerus, listed in Table. 2.1,

[1]Date accessed 15-Jan-2021

which were utilized, and further described, in the case study presented in Paper A5.

Table 2.1: OSP reference models utilized by the Gunnerus case study presented in Paper A5.

Component	Tool	Vendor	FMI version
ThrusterDrive	20sim	SINTEF Ocean	1.0
PowerPlant	20sim	SINTEF Ocean	1.0
VesselFmu	VeSim	SINTEF Ocean	1.0
PMAzimuth	VeSim	Kongsberg Maritime	1.0

3 Fundamental technologies for co-simulation

This chapter introduces existing standards for co-simulation as well libraries and tools developed as part of the dissertation to support the most promising standards.

3.1 Open standards for co-simulation

This section provides an overview of existing and open standards related to co-simulation that could serve to support demanding maritime operations. Both standards for executing (how) and defining (what) co-simulations are considered. The obvious benefit of taking advantage of established standards is that it builds on previous knowledge, fosters collaboration, and enables re-usability [24]. Ad-hoc solutions using some kind if messaging protocol is not considered as they are not standardized in terms of co-simulation functionality. Furthermore, the interfaces developed for ad-hoc co-simulation depend on the simulators it connects; therefore, the re-usability of models and interfaces is not always evident [31] and might contribute to vendor lock-in as the interfaces may not be sufficiently generalized.

3.1.1 The Functional Mock-up Interface

The FMI [23] 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 an FMU. The main goal of the 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. The FMI standard consists of two main parts, both of which a single FMU may support:

- *FMI for ME:* Models are exported without a solver, as illustrated by Fig. 3.1, and are described by differential, algebraic, and discrete equations with time-, state-, and step-events.
- *FMI for CS:* Models are exported with a solver, as illustrated by Fig. 3.2, and data is exchanged between sub-systems at discrete communication points. In the time between two communication points, the sub-systems are solved independently from each other.

Figure 3.1: FMI for model exchange (from [1]).

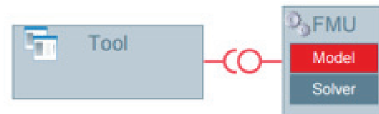


Figure 3.2: FMI for co-simulation (from [1]).

Complex cyber-physical systems, like vessels, require models from several different domains, likely developed in separate, domain-specific tools. The FMI enables such models to be integrated in a standardized way. Supported by over 140 tool vendors to date, it has become the de facto standard for co-simulation and model exchange. Moreover, a recent survey showed that experts consider the FMI standard to be the most promising standard for continuous time, discrete event, and hybrid co-simulation [24]. Although this standard has reached acceptance in industry, it provides only limited support for simulating systems that mix continuous and discrete behavior, which are typical for cyber-physical systems [32]. A future version of the standard (FMI 3.0) will introduce clocks for synchronization of variable changes across FMUs, allowing co-simulation with events.

3.1.2 High Level Architecture

The High Level Architecture [33] (HLA) is an open international IEEE standard for distributed simulation. It focuses on interoperability and re-usability of the components (called federates) and offers time management interoperability as well as sophisticated data distribution concepts [34]. HLA was initially developed in the 1990s under the leadership of the US Department of Defense. However, due to its openness and generic character it has also had large impact on non-military distributed simulation applications. While the FMI and the HLA seems to be competing standards, they are not mutually exclusive as demonstrated in [35, 36], where FMUs are incorporated as HLA federates. Similarly, it should be possible to wrap an HLA federate as an FMU using the federate object model (FOM) as the basis for the FMI *modelDescription.xml*. When HLA simulations are executed in time unconstrained mode, the resulting FMU might be treated as a live-stream of data in a real-time synchronized FMI simulation.

3.1.3 System Structure and Parameterization

As an extension of the FMI standard, the SSP [37] standard aims to facilitate the design, simulation and execution of a network of components (e.g., FMUs). It is a tool independent standard to define complete systems consisting of one or more components, including its parameterization, which can be transferred between simulation tools. The structure of a simulation is defined in an XML configuration file. At least one configuration file named *SystemStructure.ssd* must be present. However, additional configurations may optionally be defined, allowing a single SSP archive to contain multiple simulation configurations. Simply explained, an *.ssd* defines which models make up a simulation (components), which variables are exposed (connectors), how they are connected (connections), and how they are parameterized (parameter-sets). Annotations are used to define tool-specific features. The *.ssd* files are packed, together with any resources required, like FMUs, in a zip archive with an *.ssp* extension. The main use of this standard is to:

1. Define a standardized way to store and apply parameters to these components.
2. Define a standardized format for representing the interconnections between components.

Thus, the SSP allows complete systems to be defined, shared, and re-used in a tool-independent manner. Much as the FMI allows decoupling from the simulation tools, the SSP allows decoupling from the co-simulation master.

3.1.4 Open Simulation Platform - Interface Specification

The OSP interface specification (OSP-IS) is an addition to the FMI 2.0 standard for co-simulation, which provides a method for adding semantic meaning to model interface variables. The OSP-IS aims to enable faster construction of co-simulation system by simplifying the model connection process, and validation of semantically correct simulations. Basically the OSP-IS allows model creators to bundle an additional XML document alongside an FMU with the intent of easing the process of connecting inputs and outputs between two compatible models.

3.1.5 Open Simulation Platform - System Structure

The OSP system structure (OSP-SS) is an alternative to the SSP standard, developed alongside the *libcosim* library for co-simulation introduced in Section. 4.2.1. It re-uses parts of the SSP, like unit definitions, and uses a similar structure. A major difference lies in how connections are defined. The OSP-SS natively supports OSP-IS connections, allowing groups of variables to be connected between models that supports it. Furthermore, it enables the declaration of custom functions. Functions in the context of OSP-SS are pre-defined code blocks that are placed between FMUs in order increase or decrease the number of outgoing signals and/or to manipulate their value. This allows more advanced variable transformations than the few generic mapping functions found in SSP, but requires more effort to support.

3.1.6 Distributed Co-simulation Protocol

The Distributed Co-simulation Protocol (DCP) [38] is an FMI-compatible standard for real-time and non-real-time system integration and simulation. Thus, the DCP is a promising solution for realizing digital twin systems and other systems that require real-time interaction with physical components. However, the standard seemingly lacks adoption and is more involved than the FMI standard.

3.1.7 Standards considered in this work

Going forward, the subsequent work chose to focus on the FMI and SSP standards. Since its inception, the FMI has gained far more traction than the HLA with regards to co-simulation, as the sheer amount of tools that support it suggests [1]. As noted by [24] experts also consider it to be the most promising standard for continuous time, discrete event, and hybrid co-simulation. It is worth noting that some efforts have been devoted towards combining the FMI and HLA standards, as demonstrated in [35, 36]. However, this approach is too complex in terms of usability for the current study's purpose. The SSP is chosen as it fits naturally together with the FMI in order to define standardized co-simulation systems. It is chosen over the OSP-SS as it has been adopted by multiple tools, thus allowing a single SSP archive to be simulated in more than one tool as demonstrated in Chapter. 5. Furthermore, the SSPGen tool introduced in Section 3.2.5 minimizes verbosity and enables usage of the OSP-IS in an SSP context. Two major reasons for the OSP-SS to exist. Usage of the DCP is not considered at the moment. However, given its close relationship and compatibility with the FMI standard, future research should consider the possibility of adopting it in order to facilitate real-time requirements of distributed components.

[1] https://fmi-standard.org/tools/

3.2 FMI-based co-simulation libraries and tools

This section gives an overview of existing libraries enabling FMI import, as well as a presentation of the various libraries and tools that have been developed as part of this dissertation in order to promote, simplify, and improve upon FMI-based co-simulation. The ultimate goal, however, is to support the co-simulation platforms described in Chapter 4, and subsequent use-cases realized through their employment.

3.2.1 Overview of existing libraries

Table. 3.1 shows an overview of existing open-source software libraries that enable import of FMUs. This means that the library should be able to unzip the FMU archive, parse the *modelDescription* XML and interact with the underlying C code distributed as platform-specific binaries.

Table 3.1: 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	2.2.3	BSD
FMU SDK		x			x	x	x	x	2.0.6	BSD
FMI++		x	x[a]	x[a]	x	x	x[b]	x[b]	-	BSD
FMI4cpp		x			x	x		x	0.8.0	MIT
PyFMI				x	x	x	x[b]	x[b]	2.8.5	LGPLv3
FMPy				x	x	x	x[b]	x[b]	0.2.26	BSD
JFMI			x		x		x		1.0.2	MIT
JavaFMI			x		x	x			2.26.3	LGPLv3
FMI4j			x		x	x		x	0.36.4	MIT

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

3.2.2 FMI4j

FMI4j, introduced in Paper A1, is a software package for dealing with FMUs on the JVM that supports import of models compatible with FMI 1.0 & FMI 2.0 for CS & ME as well as export of models compatible with FMI 2.0 for CS. Since the paper on FMI4j was published, the library has gone through some changes. Rather than using Java Native Access (JNA) to interact with native code, it now uses the Java Native Interface (JNI), which is significantly faster. Furthermore, FMI4j is now capable of exporting Java code as FMUs compatible with the FMI 2.0 for CS. Similarly, some features have also been removed: the ability to wrap ME models as CS models and FMU2Jar, a tool for transforming FMUs into regular Java libraries. The ability to wrap ME models was removed due to the fact that users themselves should perform this task to meet their specific needs, such as the use of specialized solvers or to solve multiple ME models together. Furthermore, the solution was dependent of the JVM, and standalone solutions exist that can turn ME FMUs into tool-independent CS FMUs. The point of FMU2Jar was to simplify the interaction with individual FMUs on the JVM by automatically generating a customized high-level API for interacting with it, for example to provide named getters and

setters for each exposed variable. However, this use-case is more or less nonexistent, as it is more natural to simulate FMUs together using some kind of readily available tool, where one does not interact with individual FMUs directly. However, the work provided some valuable insights into how one could leverage the build-system to inject auto-generated source code into Java applications. Compared to JavaFMI, which also runs on the JVM, FMI4j is much faster at interacting with imported FMUs and also exports FMUs that runs significantly faster, both of which are crucial for realizing efficient co-simulations. Listing. 3.1 shows how FMI4j may be used in order to develop FMI 2.0 compatible CS models using Java. FMI4j also integrates with Gradle, which makes the process of actually building the FMUs easier.

Listing 3.1: Minimal code example showing how to write FMI 2.0 compatible simulation models using FMI4j.

```
@SlaveInfo(author = "John Doe")
public class JavaSlave extends Fmi2Slave {

    @ScalarVariable(causality=Fmi2Causality.output)
    private double realOut = 2.0;

    public JavaSlave(Map<String, Object> args) {
        super(args);
    }

    @Override
    public void doStep(double currentTime, double dt) {
        realOut += dt;
    }

}
```

3.2.3 FMI4cpp

FMI4cpp is an open-source C++ library for importing models adhering to the FMI 2.0 for CS standard. Written in modern C++, it was developed as an easier to use alternative to the commonly used FMI Library written in C. Looking at the activity on its GitHub page, one could guesstimate that this library has been adopted by more than a couple of dozen users, which is a fair amount. Unfortunately, unlike the other tools presented here, it has not been used to support the thesis in any substantial way, although it was used in earlier stages by FMU-proxy, introduced in Section 3.2.6, to support a C++ based build. Nonetheless, the lessons learned while developing it have proven essential for supporting the *libcosim* library developed as part of the OSP, introduced as part of the following chapter.

3.2.4 PythonFMU

PythonFMU, introduced in Paper A4, is a framework for developing FMI 2.0-compatible models from regular Python 3.x code, which was mainly developed in order to enable data scientists with the Intelligent Systems Lab at NTNU in Ålesund to more easily contribute with models. Python benefits from a vast number of high quality software libraries, and PythonFMU allows the utilization of them within a co-simulation context. Listing. 3.2 shows how PythonFMU can be used in order to develop FMI 2.0 compatible simulations using Python.

Listing 3.2: Minimal code example showing how to write FMI 2.0 compatible simulation models using PythonFMU.

```
class PythonSlave(Fmi2Slave):

    author = "John Doe"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
```

```
        self.realOut = 3.0
        self.register_variable(Real("realOut", causality=Fmi2Causality.output))

    def do_step(self, current_time, step_size):
        return True
```

Another feature of PythonFMU is the ability to wrap comma-separated-values (CSV) files as co-simulation FMUs. While crude, this feature makes it trivial to supplement a co-simulation with historical or pre-genenerated inputs. Furthermore, FMUs generated from CSV data are able to interpolate real valued data points. PythonFMU also support more advanced, though context-dependent, FMI features like getting/setting and serializing/deserializing state, which allows it to be used in conjunction with more advanced co-simulation algorithms that depends on this feature.

3.2.5 SSPgen

SSPgen is a Kotlin-based Domain Specific Language (DSL) designed for creating SSP 1.0 compatible systems. Using SSPGen, SSP systems may be distributed as single, easily modifiable, and executable source files rather than zipped archives (.ssp). Instead of writing large, complicated XML documents defining the simulation structure, it allows the same document to be written using a simplified Kotlin DSL. This has several advantages over manually writing XML, such as the ability to compute parameter values. For example, it is possible to set the value of a parameter as the result of some function invocation, like $realParameter = PI/2.0$. Furthermore, entries may be copied and subsequently modified, leading to less verbose and error prone documents. SSPgen also automatically validates the content and handles the packaging of the required XML, together with any additional resources required, into a ready-to-use SSP archive. Resources can be both local files and URLs, thus allowing FMUs to be fetched directly from a remote repository. Listing. 3.3 demonstrates how SSPgen can be used to define the quarter-truck simulation system used as part of the case-study presented in Paper A6. As shown, SSPgen also allows OSP-IS connections to be formed, simplifying the model connection process, and enables validation of semantically correct simulations. This is achieved by transforming the OSP-IS variable groups into corresponding scalar connections supported by the SSP.

Listing 3.3: Complete SSPgen definition for the quarter-truck system.

```
@file:Repository("https://dl.bintray.com/ntnu-ihb/mvn")
@file:DependsOn("no.ntnu.ihb.sspgen:dsl:0.4.1")

import no.ntnu.ihb.sspgen.dsl.ssp

ssp("QuarterTruck") {

    resources {
        file("fmus/chassis.fmu")
        file("fmus/wheel.fmu")
        file("fmus/ground.fmu")
    }

    ssd("QuarterTruck") {

        author = "John Doe"
        description = "A quarter-truck co-simulation system"

        system("QuarterTruckSystem") {

            elements {
                component("chassis", "resources/chassis.fmu") {
                    connectors {
                        real("p.e", output)
                        real("p.f", input)
                    }
                    parameterBindings {
                        parameterSet("initialValues") {
```

```
                            real("C.mChassis", 400)
                            real("C.kChassis", 15000)
                            real("R.dChassis", 1000)
                        }
                    }
                }

                component("wheel", "resources/wheel.fmu") {
                    connectors {
                        real("p.f", input)
                        real("p1.e", input)
                        real("p.e", output)
                        real("p1.f", output)
                    }
                    parameterBindings {
                        parameterSet("initialValues") {
                            real("C.mWheel", 40)
                            real("C.kWheel", 150000)
                            real("R.dWheel", 0)
                        }
                    }
                }

                component("ground", "resources/ground.fmu") {
                    connectors {
                        real("p.e", input)
                        real("p.f", output)
                    }
                }
            }
            <-- SSPGen allows not only regular SSP connections, ... -->
            connections {
                "chassis.p.e" to "wheel.p1.f"
                "wheel.p1.f" to "chassis.p.f"
                "wheel.p.e" to "ground.p.e"
                "ground.p.f" to "wheel.p.f"
            }

            <-- but also OSP-IS type connections -->
            ospConnections {
                "chassis.linear mechanical" port to "wheel.chassis port"
                "wheel.ground.port" to "ground.linear mechanical port"
            }

        }

    }

}.build()
```

3.2.6 FMU-proxy

FMU-proxy, introduced in Paper A2 and further expanded on in Paper A3 is a framework that allows distributed execution of single FMUs. In some cases, users may encounter issues when trying to run an FMU that are related to one of the following:

1. The FMU does not support the current operating system.
2. The user has security concerns regarding the content of the FMU.
3. The FMU requires a license that is not available on the target computer, but is readily available on another.
4. The FMU may not instantiate more than one slave instance within a single process.
5. The FMU conforms to the FMI 1.0 specification, while the importing environment only supports version 2.0.

FMU-proxy can solve these issues by wrapping the original FMU in a server program, which is then accessed using platform-independent RPCs. In this way, the FMU may either run in a sandbox or on a different computer altogether that meets the requirements. This should solve issues 1-3, or run in a separate processes on the same computer, which would solve issue 4. The server program is able to handle FMUs targeting both version 1.0 and 2.0 of the FMI standard, which subsequently share the same remote API, thus resolving issue 5.

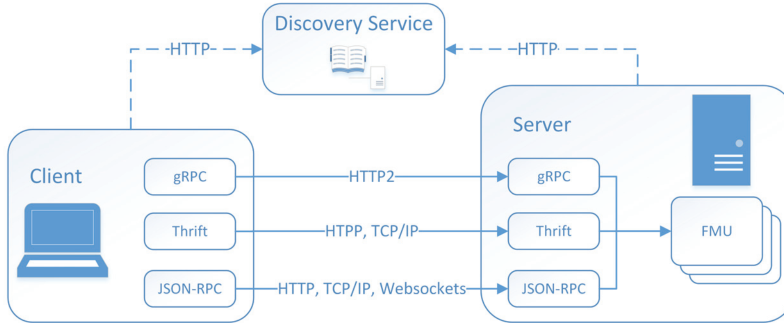


Figure 3.3: Overview of the initial FMU-proxy structure.

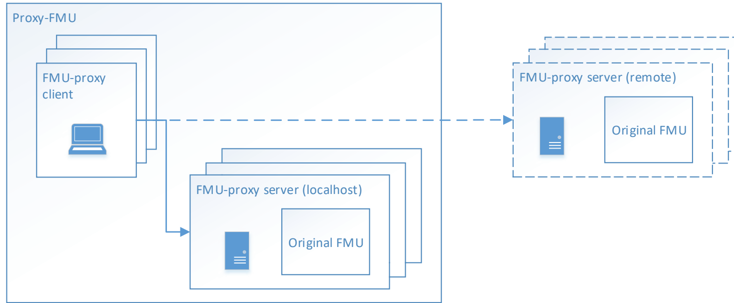


Figure 3.4: Overview of the current FMU-proxy structure.

The current version of FMU-proxy is quite different from the initial versions described in the papers published. An illustration of the initial structure can be seen in Fig. 3.3. Initially, users would interact with an FMU-proxy server using a client compatible with one of the available RPC technologies used. This approach required tools that wanted to interact with FMU-proxy to implement additional client-side code. Although FMU-proxy leverages schema-based RPCs that auto-generate most of this code, this approach is not plug-and-play. Currently, however, FMU-proxy itself is packaged as an FMU, as illustrated by Fig. 3.4. Using the *fmu-proxify* CLI tool, it is possible to specify an existing FMU, adhering to either FMI 1.0 or 2.0 for CS, and wrap it inside a new FMI 2.0-compatible FMU, which internally communicate with the original FMU. FMI4j is used to produce the FMU, thus a JVM is required to run the model. Networking is realized using Thrift over the TCP/IP protocol, and was chosen based on previous experience due to ease of use and performance. However, this is an internal implementation detail that can be changed without knowledge to the users. The generated FMU bundles a server, so the FMU is self-contained and works without any user configuration when targeting *localhost*. It is also possible to start an FMU-proxy server on a different computer and configure the generated FMU to connect to the remote server. This is achieved by modifying a configuration text file inside the FMU. This new solution is immensely powerful as it allows models to run distributed as an implementation detail of the model itself. Thus, the solution

can be used by any FMI 2.0-compatible tool without any modifications, allowing existing co-simulation systems to interact with previously unsupported models.

4 Open-source co-simulation platforms

The previous chapter introduced various standards, supporting libraries, and tools for co-simulation. However, complex interconnected co-simulations, such as simulations for demanding maritime operations, require a capable co-simulation platform. The platform should be able to import all models required, connect them, and simulate them together using some suitable master algorithm. This chapter provides an overview of available and open-source co-simulation platforms that supports the FMI. Additionally, it introduces the OSP and Vico platforms, the development of which has been a focus during the dissertation work.

4.1 Overview of open-source FMI based co-simulation platforms

Table. 4.1 shows an overview of currently existing and open-source co-simulation platforms. In order to count as a platform in this respect, the tool must be able to solve systems of interconnected FMUs using some form of master algorithm. The master is responsible for both coordinating the overall simulation and transferring data [16]. The FMI is flexible enough to allow both simple and advanced master algorithms to be implemented. Which algorithm to choose depends on the availability of algorithms on the platform in question as well as the capabilities of the models participating in the simulation. While more advanced co-simulation algorithms would be preferable for accuracy reasons, in practice very few models implement the optional capabilities that make such algorithms possible, like the ability to save and restore state and acquire directional derivatives. Thus, more basic algorithms like variations of a iterative fixed-step algorithm are often chosen. Together with libcosim and Vico introduced below, case-study based comparisons of the tools that support the SSP standard are provided in Chapter 5. Only SSP compliant tools are considered in order to ensure that the case-study is defined in a standardized manner and also to show the benefits of standardizing simulations on a system level.

4.2 The open simulation platform

The OSP [39] is an open-source industry initiative for co-simulation of maritime equipment, systems, and entire ships. Developed in collaboration between DNV-GL, Kongsberg Maritime, SINTEF Ocean, and NTNU, the project aims to create a maritime industry ecosystem for co-simulation of *black-box* simulation models and *plug and play* configuration of systems. The OSP relies on the FMI standard and the OSP-SS in conjunction with the OSP-IS for defining the simulation structure and connections. This facilitates the effective building of digital twins, which in turn can be used to solve challenges with designing, building, integrating, commissioning, and operating maritime systems. The OSP is an umbrella for several independently maintained projects, which are as follows:

1. *libcosim* – A C++ co-simulation library.

Table 4.1: Open-source co-simulation platforms supporting the FMI.

	FMI support										
	CS		*ME*								
Name	*v1.0*	*v2.0*	*v1.0*	*v2.0*	**SSP**	**Distri-buted**	**API**	**CLI**	**GUI**	**Last update**	**License**
Coral	✓	✓				✓	✓	✓		Dec. 2018	MPLv2
DACCOSIM		✓				✓			✓	Feb. 2020	AGPL
FMI Go!	✓	✓	✓	✓	✓[a]	✓		✓		Nov. 2019	MIT
Maestro		✓				✓	✓[b]		✓	Oct. 2020	GPLv3
MasterSim	✓	✓					✓	✓	✓	Oct. 2020	LGPLv3
Ptolemy II	✓	✓	✓	✓			✓		✓	Jun. 2018	MIT
FMPy	✓	✓	✓	✓	✓[a]		✓	✓	✓	Oct. 2020	BSD
OMSimulator		✓		✓	✓		✓	✓	✓	Jan. 2019	GPLv3

[a] Draft version
[b] HTTP API

2. *libcosimc* – A simplified C interface to *libcosim*.
3. *cosim4j* – A Java wrapper for *libcosim*.
4. *cosim* – A CLI for *libcosim*.
5. *cosim-demo-app* – Server-client demo application for *libcosim*.
6. *OSP-SS* – A standard for defining the structure of a co-simulation similar to the SSP.
7. *OSP-IS* – An addition to the FMI 2.0 standard for co-simulation, which provides a method for adding semantic meaning to model interface variables.
8. *osp-validator* — A tool for validating FMUs and simulation configurations against the OSP-IS.
9. *Kopl* – A graphical user interface for setting up simulations.

The above software modules are all open-source, with the exception of *Kopl*, which is closed-source, but free to use. With common tools, standards, and specifications, the OSP seeks to foster co-simulation collaboration within the industry.

In order to support this dissertation, a JVM wrapper for *libcosim*, named *cosim4j*, as well as improvements to the SSP support, were contributed to the project. Proper support for SSP is crucial in order to enable re-usability between tools, while *cosim4j* makes the underlying library more accessible to a wider audience.

4.2.1 libcosim

libcosim is a C++ library that orchestrates the co-simulation of models that conform to FMI 1.0 & 2.0 for CS and serves as the cornerstone of the OSP project. The design of *libcosim* is centralized, with all data flowing through the master. This makes for a less complicated, easier to maintain, easier to debug, and more flexible design compared to similar co-simulation engines such as Coral [7], where data flows directly between slaves. For instance, entities that want to observe or manipulate the simulation can do so directly as all data is available from a single source. Pure distributed co-simulation masters such as Coral and FMI Go! dictate that all slaves are to be run distributed, whereas *libcosim* makes this entirely optional. Support for this is currently implemented through integration with FMU-proxy. The structure of the

system to be simulated can be defined using either the SSP or the OSP-SS standards. The main benefit of the OSP-SS, provided that the FMU supports the OSP-IS, is the possibility to define FMU interconnections at a higher level, i.e. variable group connections rather than single scalar connections.

4.2.2 cosim4j

cosim4j is a JVM wrapper for *libcosim* that has been developed by the author and contributed to the OSP specifically to support the work conducted during this dissertation. C++ can be a challenging language to learn. Especially compared to higher-level languages like Python or Java. For instance Java has fewer features to learn, is garbage-collected and comes with a richer standard library. Additionally, the tooling, in the form of integrated development environments (IDEs), build systems, and package managers, is state-of-the-art. All of this makes developing for the JVM generally easier than developing for C/C++. The goal of the Java API is to be generally easier to use and provide more high-level features than its native counterpart. To make the library more accessible, it is made available through Maven. Furthermore, the artifacts include pre-built native binaries for 64-bit Windows and Linux systems, which means that no prior installation of *libcosim* is required.

4.3 Vico

Vico, presented in Paper A6, is a generic co-simulation platform developed by the author that is based on the ECS software architecture [40, 41, 42, 43]. The platform has been under development in various forms for many years, serving as a test-bed for different architectural approaches to enable flexible co-simulations. For example an early prototype was presented in [44] and further developed in [45]. In the end, the ECS architectural pattern, illustrated by Fig. 4.1, was chosen, as it offers a powerful, intuitive, and clean way of separating behavior and state. Separating behavior and state allows, for example, workflows where the fidelity of the simulation can be more easily changed, even during run-time. The ECS follows the composition-over-inheritance principle, which allows for greater flexibility in terms of defining simulation objects than alternatives afford. Rather than having objects inheriting data and functionality from a parent object (object-oriented programming), the object (entity) is composed of data (components). Every entity consists of one or more components which contain data. Therefore, the behavior of an entity can be changed during run-time by systems that add, remove, or mutate components. This eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain, and/or extend. For a more in-depth description of the employed ECS architecture, the reader is referred to Appendix. F.

Vico allows co-simulation not only between FMI components, but also between physics components and other generic components. Additionally, 3D visualization and 2D plotting are integrated as part of the framework, and time-series data can be exported as CSV files. Some of the highlights of this framework are as follows:

1. Separation between state (components) and behavior (systems).
2. Support for FMI 1.0 & 2.0 for co-simulation.
3. Support for SSP 1.0.
4. Integrated 2D plotting.
5. Integrated (implementation independent) 3D visualization and physics.

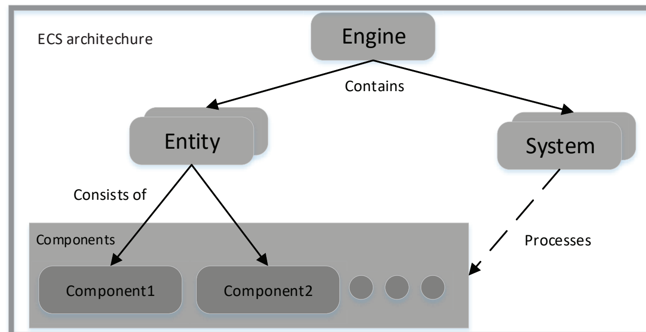


Figure 4.1: High-level overview of the ECS architecture.

6. Scriptable scenarios using a Kotlin DSL.
7. Integration with FMU-proxy, providing support for (optional) distributed model execution.

Vico is written in Kotlin, a JVM language that is 100% interoperable with Java. This means that Vico benefits from the vast number of libraries developed for the JVM as well as industry-leading tooling. Kotlin also provides scripting capabilities, which means that Vico can be run in a scripting context. Vico provides support for scriptable scenarios that utilizes this feature. In the context of Vico, scenarios are pre-configured actions that should be executed at a certain time or are triggered by an event. Vico can be used both as a library or as an application through the available CLI. This CLI can be used to run single FMUs or a system of FMUs described using the SSP standard. In both cases, configurations for visuals, plots, and data export as well as scenario files can be supplied in order to enhance the simulation.

4.3.1 FMI & SSP support

A module named *fmi* adds support for FMI 1.0 & 2.0-based co-simulation, and relies on FMI4j for interacting with FMUs. Since FMI4j was initially released, it has changed the way it interacts with native code, making it the fastest open-source JVM library for simulating FMUs. The library also supports export of FMUs compatible with FMI 2.0 for co-simulation and provides a Gradle plugin to simplify the usage of this feature. This allows for a workflow where slaves can be automatically exported to FMUs during the build process and loaded by Vico, or any other JVM based framework, within the same project. The *fmi* module adds a system named *SlaveSystem* that takes an instance of *MasterAlgorithm*, which is an interface, as a constructor parameter. The idea is that users should be free to develop their own master algorithm with as little cognitive overhead as possible. However, the module also provides a ready-to-use implementation of a fixed step-size master algorithm, which allows users to configure slaves to run at different rates. Due to the fact that FMUs consists of both behavior and state, they are difficult to fit into an ECS architecture, as an FMU is neither a component nor a system. This is solved in Vico by creating a component that represents the location of an FMU as well as a variable buffer. The system then loads the FMU from the component path

and continuously updates the buffer. As an optimization, only variables the user explicitly requested or that are used in connections are buffered. The use of I/O buffers significantly improves performance when dealing with distributed models. This is evident by comparing the results of the Gunnerus case-study presented in Section 5.2.

SSP support is also provided by the *fmi* module. Like OMSimulator and libcosim, Vico supports a limited set of the SSP 1.0 standard, where additional features might be implemented as use-cases appear. Worth noting is that both Vico and libcosim support a special kind of component that enables distributed execution of FMUs using FMU-proxy. Distributed execution of FMUs is required in cases where the FMU for example cannot be instantiated more than once per process or when it has been built for an incompatible system architecture. This particular feature was required in order for FMU-proxy to work prior to its update. With the current version of FMU-proxy, however, it is no longer necessary to implement explicit FMU-proxy support in code. It is retained for backwards compatibility only.

4.3.2 3D visuals

The 3D capabilities of Vico are quite powerful and allow visualization of primitives as well as generic tri-meshes, water planes, lines, tubes, and various helper objects like axes and arrows. 3D models in selected formats, including textures, may also be imported. Furthermore, the API is agnostic to the underlying 3D rendering framework used, allowing different rendering backends to be employed. Data related to visualization is available through a web interface, allowing 3D graphics to be shown in the browser as well as on the desktop. Naturally, visualization properties may be configured in code, but it is also possible to use XML. Thus simulations loaded using the CLI can also be visualized. Listing. 4.1 shows how the variables of an FMU may be mapped to a visual entity.

Listing 4.1: Configuring visualization properties for use by Vico using XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<vico:VisualConfig xmlns:vico="http://github.com/NTNU-IHB/Vico/schema/VisualConfig">
    <vico:Transform name="vesselModel">
        <vico:Geometry>
            <vico:Shape>
                <vico:Mesh source="../obj/Gunnerus.obj"/>
            </vico:Shape>
        </vico:Geometry>
        <vico:PositionRef>
            <vico:xRef name="cgShipMotion.ned.east"/>
            <vico:yRef name="cgShipMotion.ned.down">
                <vico:LinearTransformation factor="-1"/>
            </vico:yRef>
            <vico:zRef name="cgShipMotion.ned.north"/>
        </vico:PositionRef>
        <vico:RotationRef>
            <vico:xRef name="cgShipMotion.angularDisplacement.pitch"/>
            <vico:yRef name="cgShipMotion.angularDisplacement.yaw">
                <vico:LinearTransformation offset="180"/>
            </vico:yRef>
            <vico:zRef name="cgShipMotion.angularDisplacement.roll"/>
        </vico:RotationRef>
    </vico:Transform>
</vico:VisualConfig>
```

4.3.3 Scenarios

Scenarios in Vico are predefined actions to be executed at specific points in time or when certain conditions are met. Scenarios can be written inline using the API or supplied as standalone files. These files are written using an intuitive Kotlin DSL, which are parsed and compiled by the environment and passed to the simulation as if it was written as regular code. That is, there is no conversion between the passed-in script and the running byte code, which would be

necessary if the scenario were written using a typical configuration format like JSON, XML, or YAML. This enables the script to access simulation objects and evaluate expressions. This is immensely powerful, and allows script expressions to be evaluated during run-time with little or no additional performance overhead compared to inline configuration of the simulation. An example scenario script is shown in Listing. 4.2 for demonstration purposes.

Listing 4.2: Example scenario configuration.

```
@file:Repository("https://dl.bintray.com/ntnu-ihb/mvn")
@file:DependsOn("no.ntnu.ihb.vico:core:0.3.3")

import no.ntnu.ihb.vico.dsl.scenario

scenario {

    invokeAt(1.0) {
        real("e1.localPosition.x").set(1.0)
    }

    invokeWhen {
        predicate {
            real("e1.localPosition.x") > 3.0
        }.`do` {
            real("e2.speed").set(5.0)
        }
    }

}
```

4.4 Differences between the OSP and Vico

Vico is a product of this thesis, serving as a constraint-free environment for implementing various approaches towards realizing a generic and flexible co-simulation platform. OSP, on the other hand, is a collaboration project involving several industrial partners, including NTNU, that aims to realize a more conservative goal of enabling FMI based co-simulation. There are clear differences between the two, like implementation language and scope, but both the OSP and Vico are co-simulation platforms suitable for simulating systems of FMUs, and they are not necessarily mutually exclusive. Vico is a generic high-level co-simulation framework built upon the ECS software architecture, while the OSP is more closely dependent on FMI and uses a more traditional object-oriented software architecture. In fact, the core Vico module does not know how to process FMUs at all. Rather, it provides functionality that enable generic co-simulations to be orchestrated using the ECS architectural pattern. Support for FMI & SSP is explicitly opt-in unless the CLI is used, which for convenience includes all available functionality. Due to the generic nature of Vico, *libcosim*, or rather *cosim4j*, may actually be integrated into it. Furthermore, Vico provides access to features like a CLI, web-access, and visualizations as part of the same project that are updated in unison. The OSP is a set of packages, individually versioned, that collectively provides some of the same features. A feature only the OSP possess, however, is support for the OSP-SS, an alternative to the SSP, featuring native support for the OSP-IS connection scheme and functions. However, as mentioned, SSPgen allows the OSP-IS to be applied in a SSP context, making the difference between the SSP and the OSP-SS less apparent.

5 Case studies

This chapter present the results of a set of co-simulation case studies, namely an accuracy and performance benchmark using a quarter-truck simulation system and various case studies involving the RV Gunnerus. The simulations are defined using the SSP standard, thus only tools that support this standard are included in the case studies. The use of SSP ensures that the simulation setup is standardized across the participating tools.

5.1 Accuracy and performance benchmark

This section presents a summary of the simulations results provided in Paper A6 in terms of accuracy and performance. The tools listed in Table. 5.2 are used to load and simulate the same set of models that collectively represent a simplified quarter-truck system, also known in the literature as a quarter-car system [46, 47, 48]. The system for simulation is defined using the FMI and SSP standards in order to test performance in terms of accuracy and efficiency and was chosen as it has been previously studied in the literature for the same purpose [49, 13]. The co-simulation system representing the quarter-truck is comprised of three models: the *chassis* including the suspension, the *wheel* including the tyre, and the *ground*. The system is illustrated by Fig. 5.1, with m_w and m_c representing the mass of wheel and chassis, respectively. Both masses have a single vertical degree of freedom coupled with a linear spring-damper system representing the chassis suspension and wheel tyres. The ground profile is given as external input and is excited by a jump of $0.1m$ in vertical direction at $1s$. The input and output variables used to connect these models are given in Table. 5.1. The analytical model is further derived in Appendix F.

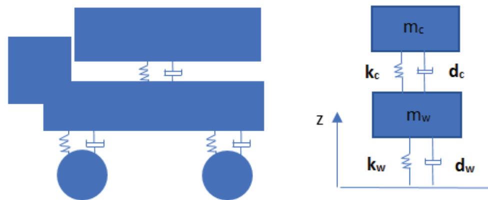


Figure 5.1: Illustration of the quarter-truck system.

The suspension force and the tyre force are given by Eq. 5.1, while the equations of motion for the chassis and wheel are given by Eq. 5.2.

$$
\begin{aligned}
F_{susp} &= k_c(z_w - z_c) + d_c(\dot{z}_w - \dot{z}_c) \\
F_{tyre} &= k_w(z_g - z_w) + d_w(\dot{z}_g - \dot{z}_w)
\end{aligned}
\tag{5.1}
$$

Table 5.1: Input and output variables of the quarter-truck models used for connections.

FMU	Variable	Input/output	Description
chassis	F_{susp}	output	Chassis suspension force applied to the wheel.
	$\dot{z}_w$	input	Velocity of the wheel from the wheel model.
wheel	F_{susp}	input	Chassis suspension force from the chassis part.
	$\dot{z}_w$	output	Velocity of the wheel sent to the chassis part.
	F_{tyre}	output	Tyre force applied to the ground.
	$\dot{z}_g$	input	Ground profile, given as vertical velocity variation from the ground model.
ground	F_{tyre}	input	Tyre force from the truck wheel.
	$\dot{z}_g$	output	Ground profile, given as vertical velocity variation sent to the wheel.

Table 5.2: Summary of tools included in the case study.

Tool	Main impl. language	Platform support	FMI version	SSP version
FMPy	Python	Win, Linux, Mac	1.0 & 2.0 for CS	Draft20171219
FMIGo!	C++	Win, Linux, Mac	2.0 for CS & ME	Draft20170606
libcosim	C/C++	Win, Linux	1.0 & 2.0 for CS	1.0
OMSimulator	C/C++	Win, Linux, Mac	1.0 & 2.0 for ME & CS	1.0
Vico	Kotlin/JVM	Win, Linux	1.0 & 2.0 for CS	1.0

$$\begin{aligned} m_c\ddot{z}_c &= F_{susp} - m_c g \\ m_w\ddot{z}_w &= F_{susp} - F_{tyre} - m_w g \end{aligned} \tag{5.2}$$

A reference solution was computed using a Runge Kutta 4 solver, with the integration time step set to $0.001s$. Additionally, the FMUs are exported with the same type of solver. Simulation results are shown using both a 100hz and 1000hz fixed-step-size for the master algorithms used. Chassis responses are shown in Fig. 5.5 and Fig. 5.6, and wheel responses are shown in Fig. 5.2 and Fig. 5.3. Except for the first second of the 100Hz simulation, the trend is practically the same for each participating tool. However, FMPy appears to constantly provide output timestamped one time-step earlier than the other tools and libcosim and FMPy both appear to generate stronger oscillations during this time period. This response can be seen more in detail through Fig. 5.4. The authors of libcosim have been made aware of this issue, and it should be fixed in a later release if it turns out to be some kind of initialization issue. Simulating the system at 1000Hz show a clear improvement in accuracy over using only 100Hz. In this case there are only small differences regarding simulation results between the tools and the reference solution. The improvement with respect to the root mean square error (RMSE) can be seen in Table. 5.3. The increase in accuracy comes, however, with a run-time cost. Nevertheless, using the co-simulation approach still provides less accuracy than the reference solution, even when using the same step-size. This shows one of the inherent weaknesses of co-simulation compared to monolithic simulations. A distributed system has trade-offs and will likely exhibit different behavior than the original system [14]. While it is clear that the simulation results differentiates

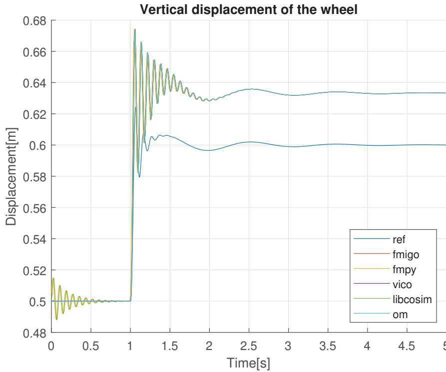


Figure 5.2: Wheel response when simulated at 100hz.

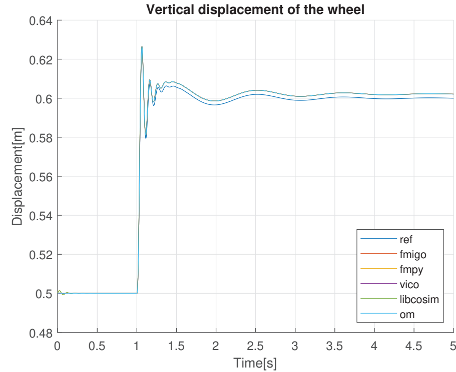


Figure 5.3: Wheel response when simulated at 1000hz.

more from the reference as the time-step grows, none of the other tools demonstrated better results than Vico using their default solver.

Table 5.3: Root mean square error of the computed vertical displacement of the wheel.

Tool	**RMSE**	
	100hz	*1000hz*
FMPy	0.0300358	0.0019367
FMIGo!	0.030062	0.0018814
libcosim	0.030109	0.0018815
OMSimulator	0.030062	0.0018814
Vico	0.030062	0.0018814

The results of a performance benchmark appear in Fig. 5.7 in the form of box plots. The benchmark is performed on a 64-bit Windows 10 system equipped with a Intel Core i5-3570 CPU with four logical processors. Each tool has been run 15 times, simulating the system for $1000s$ with a step-size of $0.001s$. FMI Go! and FMPy both exports a handful of variables to CSV. libcosim, OMSimulator and Vico is run both with and without exporting all 121 available variables to CSV. Additionally, OMSimulator also exports in MATLAB format. Vico is implemented on the JVM, which involves some overhead because it must cross the native bridge when it communicates with FMUs, but is nevertheless the fastest of the tools participating in the benchmark. OMsimulator is the second fastest, ahead of FMIGo!. The results of FMIGo! are quite impressive, considering that it is the only one of the tools to run distributed. Next is libcosim, followed by FMPy. It is not surprising that FMPy is the slowest tool, as Python is not known to be a particularly fast language. OMSimulator and Vico are configured to run this particular system single-threaded, which libcosim has no option to do, which may explain its poor performance. As the individual models in the system are computationally light, it would seem that the inherent overhead of handling threads/fibers/co-routines is actually degrading performance. Both OMSimulator and Vico were tested with multiple threads, and Vico in particular showed over a 2x performance increase when running

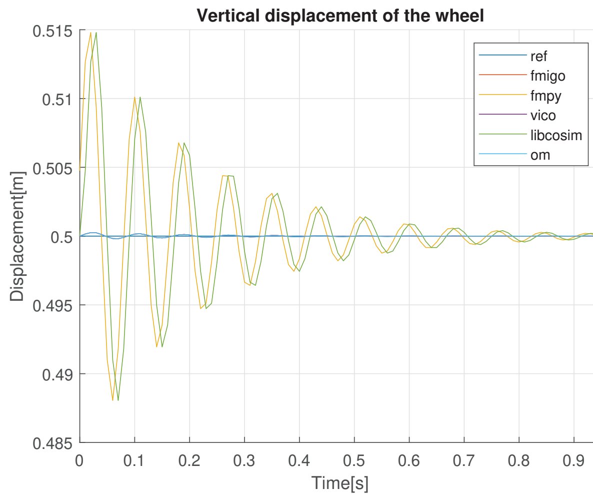


Figure 5.4: Detailed view of the the first second of simulation presented in Fig. 5.2 (100hz).

single-threaded. Note, however, that the performance indicators presented here are only valid for this particular system and should not be used as a general pointer to how well the various tools perform.

5.2 Co-simulation of the RV Gunnerus

This section presents the main research findings and important discussions concerning the conducted simulations utilizing a model of, and data from, the NTNU-owned RV Gunnerus. The vessel, which can be seen in Fig. 5.8, is equipped with the latest technology for a variety of research activities within biology, technology, geology, archaeology, oceanography, and fisheries research. In addition to research, the ship is used for educational purposes and is an important platform for maritime courses at all levels and disciplines.

Figure 5.8: Starboard view of the RV Gunnerus.

This dissertation has included extensive experimentation with creating co-simulations involving the Gunnerus. Given the existence of high-quality maritime reference models and the considerable potential that lay in modeling and simulation of the Gunnerus, it is clearly an excellent testbed to demonstrate and test co-simulation capabilities. The aggregated co-simulation model of the Gunnerus makes extensive use of readily available models, which are provided as part of the OSP repository of reference models. Additional, complementary, models have been developed as needed. Actually, the need to simplify model creation for the data scientist in the team was one of the main reasons PythonFMU was created. The various simulation configurations are described using SSP, which allows the system to be imported and simulated in a standardized way. This encourages re-usability and it allows relative validation of simulation results by comparing the output from different tools.

Paper A5 presents some of this work by introducing the strategy and preliminary results

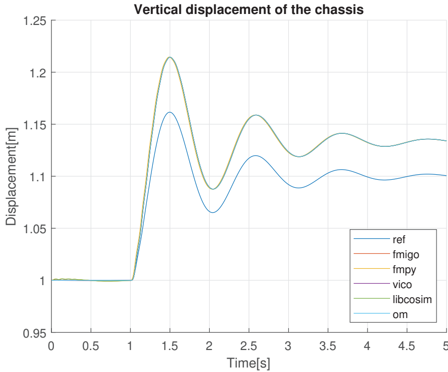


Figure 5.5: Chassis response when simulated at 100hz.

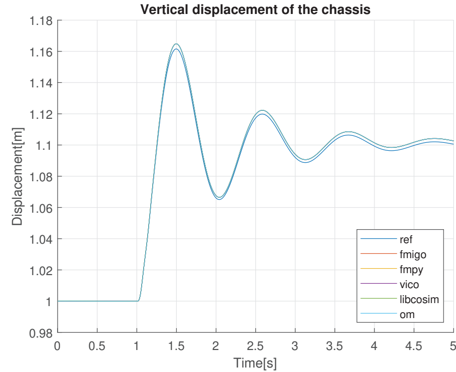


Figure 5.6: Chassis response when simulated at 1000hz.

Table 5.4: FMUs involved in the case study utilizing the RV Gunnerus presented in Paper A5.

Component	Tool	Vendor	FMI version	canBeInstantiated-OnlyOncePerProcess
HeadingController	FMI4j	NTNU	2.0	False
SpeedController	FMI4j	NTNU	2.0	False
Gunnerus	FMI4j	NTNU	2.0	False
VesselModelObserver	FMI4j	NTNU	2.0	False
ThrusterDrive[a]	20sim	SINTEF Ocean	1.0	True
PowerPlant[a]	20sim	SINTEF Ocean	1.0	True
VesselModel[a]	VeSim	SINTEF Ocean	1.0	True[b]
PMAzimuth[a]	VeSim	Kongsberg Maritime	1.0	True[b]

[a] OSP reference model.

[b] Additionally, only one instance of any model generated by this tool may be instantiated by the same process.

towards realizing a digital twin of the Gunnerus. The point of this case study is not to go into detail about how these models are implemented, which in general are black-boxes that could hide proprietary information. Rather, the point is to showcase how co-simulation technology, open-source software, open standards and a library of readily available maritime reference models can be used to develop a digital twin scenario. An overview of the models used to implement the case study is shown an Table. 5.4. Furthermore, each of the models are described in more detail below:

1. **Gunnerus -** This model contains previously recorded sensor data measured during operation of the Gunnerus. The time-series data is sampled at 1Hz and includes information such as:

    - Heading angle and percent-wise commanded RPM of the tunnel-thruster in the bow as well as the two azimuth thrusters in the aft.
    - Longitude and latitude.

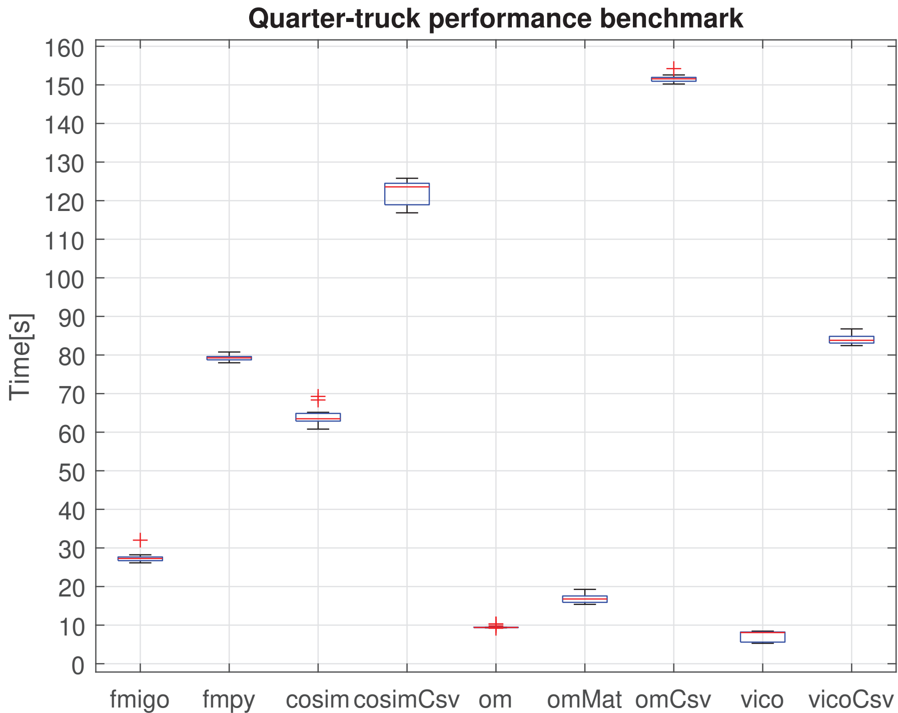


Figure 5.7: Performance of the various tools when considering the presented quarter-truck system. Simulation time=1000s, step-size=0.001s, number of runs=15.

- Surge, sway, and heave.
- Yaw, pitch, and roll.
- Wind direction and speed.
- Positional and rotational velocities.

The FMU implements linear interpolation of the recorded data, which is convenient given the low sample rate of the sensor data relative to the simulation, which runs at 20 Hz. In this work, the model acts as a stand-in for what eventually should become a stream of data originating from the real asset.

2. **VesselModel -** This model computes the vessel hydrodynamics such as the radiation forces, mass, and restoring forces as well as manoeuvring forces (resistance and cross flow drag as well as semi-empirical corrections). The equations of motions are solved by this model, summing up all the external forces acting on the vessel. SINTEF Ocean originally implemented the *VesselModel* to model the Gunnerus as part of the SimVal [50] project. It was later updated to better approximate an elongated Gunnerus vessel as part of the MAROFF Knowlege-building Project for Industry (KPN): Digital Twins for Vessel Life Cycle Service (TwinShip). While the model was validated during the SimVal project, it has yet to be validated against the elongated version of the vessel.

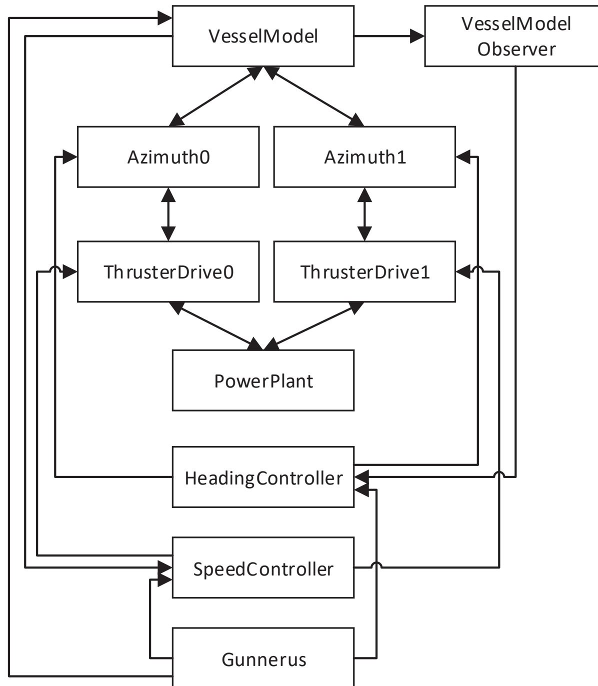


Figure 5.9: Diagram showing the logical relationship of the components involved with the case study presented in Paper A5.

3. **VesselModelObserver -** A simple model that computes the direction of travel and speed over ground of the *VesselModel* based on its current and previous position.

4. **SpeedController -** A general-purpose proportional-integral-derivative (PID) controller. It is used to regulate the force required by the *ThrusterDrives* so that the speed of the *VesselModel* and the *Gunnerus* are aligned.

5. **HeadingController -** A special-purpose proportional–integral–derivative controller where the input data used to compute the controller error is treated as angles in the range $[-180°, 180°]$. This unwinds any input angles that lie outside of the specified range.

6. **PMAzimuth -** The hydrodynamic model of the azimuth thrusters without actuator/-motor, implemented by Kongsberg Maritime using VeSim as part of the ViProMa project. Given a certain RPM command (issued by the *ThrusterDrive* FMU), location on the hull, azimuth angle, vessel speed, and the loss factor, the model will output the 3DOF (surge, sway, heave) force generated.

7. **ThrusterDrive -** A drive that converts force commands from the *SpeedController* into

RPMs for the *PMAzimuth*, developed as part of the ViProMa project [9].

8. **PowerPlant -** A maritime power plant with two equally large gensets, including auxiliary load and circuit breakers, developed as part of the ViProMa project. Further implementation notes can be found in [51].

Their logical relationship are illustrated by Fig. 5.9. As illustrated by Fig. 5.10, the system is far from trivial, with a total of 48 variable connections between the models involved.

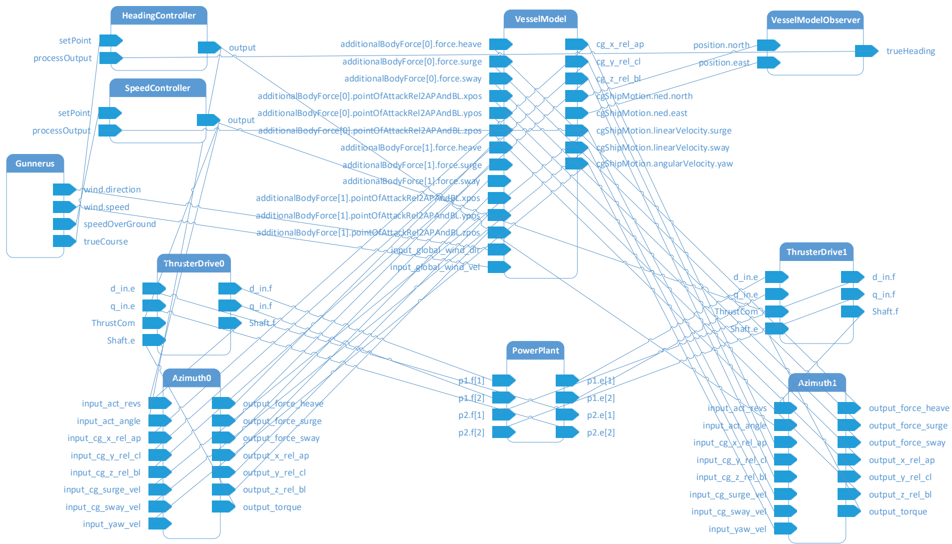


Figure 5.10: Connections between the components used in the Gunnerus system presented in Paper A5.

Fig. 5.11 shows the real and simulated path of by the Gunnerus during the presented case study. The blue line represents the real path undertaken by the Gunnerus, while the red line represents the motion of the twin ship. The arrow represent the relative magnitude of the wind speed and wind direction measured by the real vessel. The real data is a 33 minutes excerpt from the data recorded from the Gunnerus while performing maneuvers outside of Trondheim, in which it performs a U-turn. The idea of the case study is to compare the power consumption of the real vessel and the preliminary digital twin. This is done by feeding the speed and true heading of the real vessel into a set of controllers used to regulate the motion of the twin. To simplify the case study, the equipped thunnel-thruster is not utilized and the command signals to both azimuth thrusters in the aft are equal. Ideally, the power consumption should be comparable, which would indicate a good model fit. However, environmental effects such as current and waves, which are very difficult to measure, could introduce discrepancies between the real and simulated vessel. Under the assumption of constant or slowly varying environmental effect, a possible way to mitigate this in future works could be to estimate these effects using a Kalman filter [52, 53, 54].

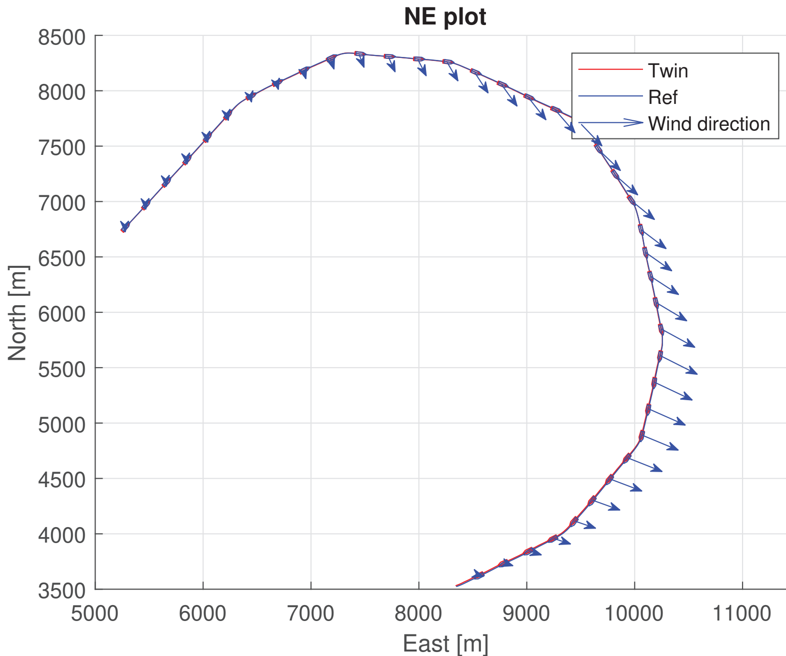


Figure 5.11: Northeast plot showing the trajectory and heading of the vessels during the experiment. The blue arrow indicates the wind direction according to north and normalized magnitude of the speed.

The power consumption is shown in Fig. 5.13, while both the course and speed can be observed in Fig. 5.12. As seen from the last figure, speed transients for the twin directly relates to changes in course. Interestingly, the power consumption calculated from the twin is showing higher correlation with the speed than that of the real vessel. After approximately 1100s, the power measurement for the real vessel is actually reduced as the speed increases. This could indicate that the vessel is affected by external forces that the model is not aware of, such as current. Therefore, the full magnitude of the observed discrepancy does not necessarily indicate a weakness in the model, but actually provides potentially valuable information regarding external environmental forces acting on the real hull. Nonetheless, it is clear that some of the underlying models could be more accurately tuned to better reflect the current vessel design. The case study makes use of a hull model, supplied by an external entity, that has yet to be thoroughly validated after the Gunnerus underwent an elongation prior to 2019. Doing so might improve the observed difference in terms of overall power consumption. Recently, model towing tests in the scale 1:9.13 were performed on the hull of the Gunnerus at SINTEF Oceans towing tank in Trondheim. The resulting data might be used to further improve this model. Additionally, other teams members are investigating additional ways of improving the overall simulation model.

Work has also been conducted in order to test models for ship maneuvering and other tasks. Fig. 5.14 shows an attempt to see how the digital model of the Gunnerus would react

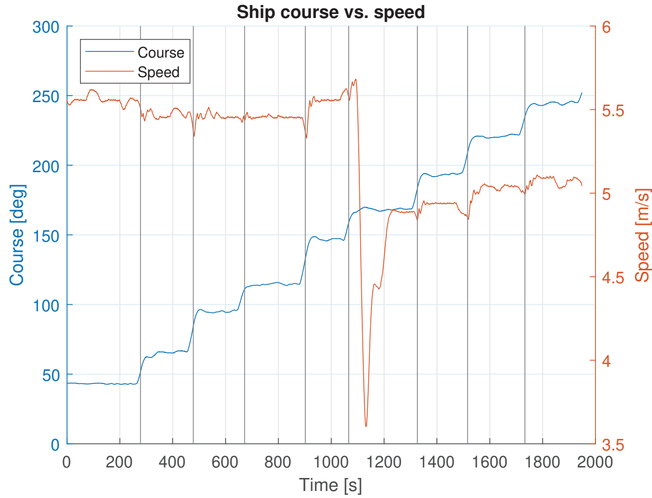


Figure 5.12: Twin surge speed with respect to course changes by the Gunnerus.

when provided with thruster commands recorded by the Gunnerus during real operations. This was done using data from a number of different docking attempts recorded at the harbor in Ålesund. During this time, a drone video-recorded the Gunnerus. The resulting video, with matching time-stamp, is subsequently overlaid on-top of the simulation rendering. The green line represents the actual path recorded by the Gunnerus, while the blue line and silhouette of the vessel represents the motion of the modeled ship. The structure of the simulation is similar to that found in Fig. 5.9, but without the *PowerPlant*, *ThrusterDrive(s)*, *HeadingController*, *SpeedController*, and *VesselModelObserver*. Rather, the measured values from the Gunnerus is fed directly into the model without the use of intermediate controllers. From the simulations, it was observed that some cases would provide better fits than others. However, this approach has some limitations. Because there are so many unknown or incomplete variables from the real world that the model cannot detect, it is not plausible to validate a model using this approach. Even with a perfect model of the vessel and environment, a small deviation in initial values due to low resolution sampling or noise in the recording would make the two trajectories diverge. However, this case highlights the usefulness and effectiveness of the visual capabilities of the Vico system, which provides immediate visual feedback to aid the user. The underlying uncertainties, however, leads to the power-consumption approach to the model validation, as demonstrated earlier. However, the general trend observed in the docking simulation was that the model moved slower than the real vessel, which actually corresponds with the results from the power consumption case study, where it was observed that in order to keep the same speed, the model consistently consumes more power than the real vessel. This enforces the belief that the Gunnerus model is in need of adjustments.

Another simulation case makes use of the Gunnerus hull model to a perform trajectory tracking simulation, as illustrated by Fig. 5.15. The visualization displayed is generated using the XML shown in the previous chapter through Listing. 4.1. The green cylinder represents the current way-point that the Gunnerus should navigate towards, while the blue line trailing the vessel shows the path used to get there. Once the vessel is within a set radius of the current

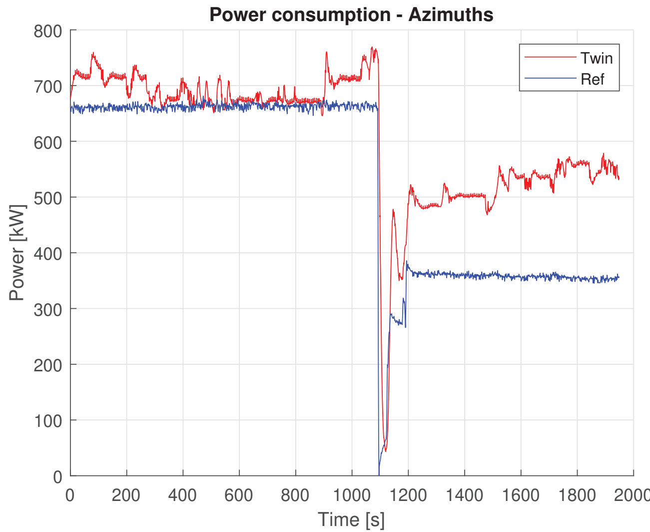


Figure 5.13: Power consumption comparison. The power output shown is the sum of the two azimuths.

way-point, another way-point appears that the Gunnerus should now try to reach. The list of way-points are specified in advance through a text file. This simulation of the Gunnerus is simpler than the ones introduced above as it does not make use of real data. The purpose of the simulation is to test the basic capabilities of the Gunnerus model in order to learn how the model behaves. In this example, the simulation makes use of the visual capabilities of Vico, which provides immediate and comprehensible feedback on how the model behaves. Furthermore, no coding is involved in making this simulation. The system is described using SSP and the visualization properties are configured by means of XML. These files are then passed to the Vico CLI and subsequently simulated.

Fig. 5.16 shows the performance of Vico compared to libcosim and OMSimulator when simulating the Gunnerus system. FMU-proxy is used in order to make the system, which originally consisted of both FMI 1.0 & 2.0 FMUs, compatible with OMSimulator. An attempt were made in order to run the system in the same set of tools as for the quarter-truck, but adopting the SSP file to the obsolete versions used by FMPy and FMIGo! proved difficult and attempts to simulate the system in those frameworks were unsuccessful. The benchmark is performed on a 64-bit Windows 10 system equipped with a Intel Core i7-8700 CPU with twelve logical processors. The simulation is run 10 times, simulating the system for $1000s$ with a step-size of $0.05s$. Vico and libcosim performs the simulation both with and without exporting available time-series data, while OMSimulator is configured to not record such data. The reason for this is that the system contains a total of 3006 variable values that must be retrieved from the various model instances at each time-step and later written to disk. Furthermore, the use of FMU-proxy means that networking is involved. Both Vico and libcosim implements a strategy to optimize variable reads and writes, however, it seems OMSimulator does not. Because of this, OMSimulator is not able to simulate the system in a timely manner

Figure 5.14: Simulation with synchronized video overlay of the real operation.

when also set to export time-series data. For example, it took OMSimulator approx. 250s to simulate 40s. To compare, Vico used approx. 58s to simulate 1000s. Furthermore, Vico runs the simulation both single- and multi-threaded. Compared to the quarter-truck system, this simulation benefits from parallel execution in terms of performance. The difference between Vico and libcosim is less in this case, but Vico still performs better when utilizing multiple threads. Even with the additional overhead of exporting time-series data, both Vico and libcosim perform better than OMSimulator. This is related to how variable reads and writes are handled by the frameworks. Basically, OMSimulator seems to perform variable reads and writes on individual variables, while libcosim and Vico execute these operations in bulk. This puts the performance of OMsimulator, which runs in parallel, in the vicinity of Vico in single-threaded mode. Assuming that accuracy and stability is maintained, higher performance is naturally beneficial as it allows more demanding simulations to be run in real-time mode, for example when used as, or as part of, a training simulator, for the purpose of integrating with sensor measurements, or when performing many simulations in bulk, saving time and decreasing cost.

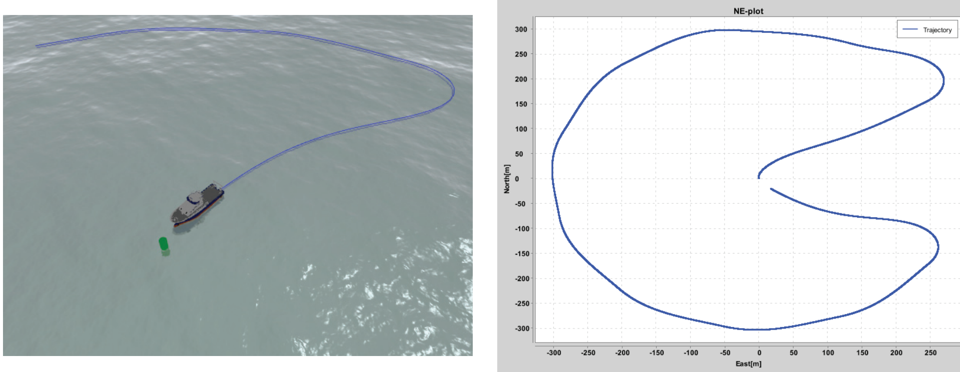


Figure 5.15: Demonstration of a vessel path following simulation running in Vico with 3D visualization and plotting enabled.

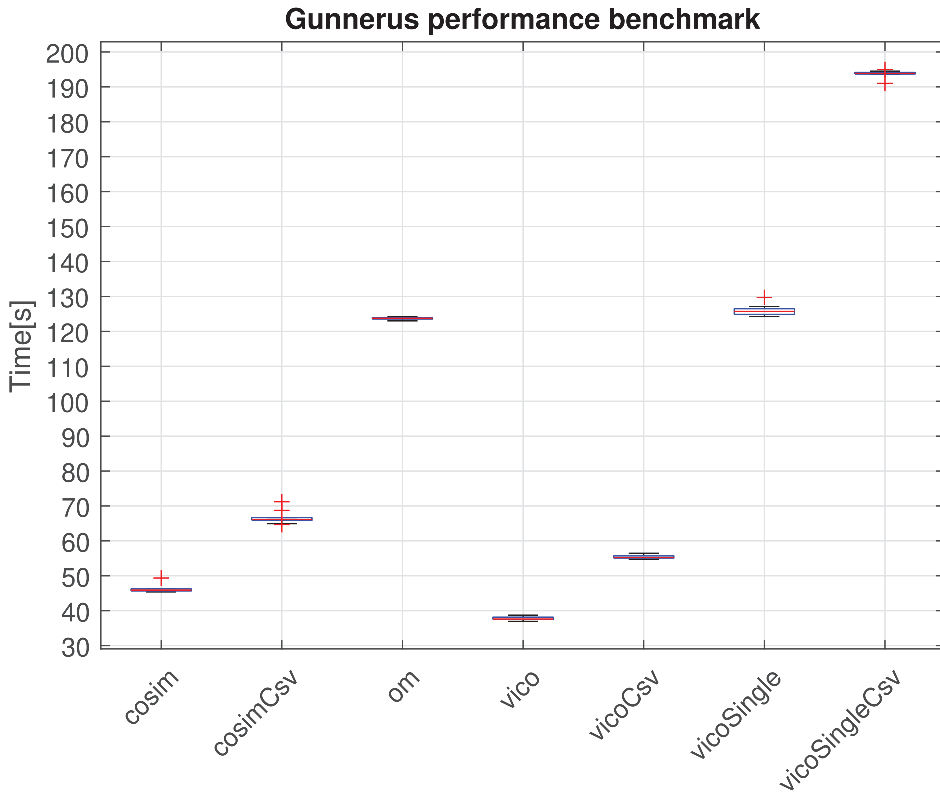


Figure 5.16: Performance of the various tools when considering the presented Gunnerus trajectory tracking scenario. Simulation time=1000s, step-size=0.05s, number of runs=15.

6 Conclusion

This dissertation has proposed and discussed enabling tools and frameworks for co-simulation driven development. All contributions in this dissertation aim to enhance the area of co-simulation and further aid towards the development of simulations related to demanding maritime operations, including digital-twins, using co-simulation. While co-simulation is not a new technology, the advances in and adoption of open standards in recent years has made it more relevant. The work presented here hopes to further drive adoption in order to keep the momentum going.

6.1 Summary of contributions

Making co-simulation more accessible and user-friendly through the development of supplementary software packages as stated by RO1 has been a key goal of this dissertation. Not only to serve this work, but any tool that in some way make use of the FMI and SSP standards. This has been realized through the development of FMI4j, FMI4cpp, PythonFMU, FMU-proxy and SSPgen. These are all open and standalone tools that in some way make realizing the building blocks needed by co-simulations easier. SSPgen allows standardized systems to be defined using an intuitive DSL. These system may then be simulated in a growing number of SSP compatible tools. FMI4j and PythonFMU enables source code, written in Java and Python respectively, to be seamlessly exported as FMUs. FMI4j in particular serves as the foundation for realizing Vico. The open-source co-simulation framework, with the main goal of enabling digital twins and supporting maritime use cases proposed by RO2. Vico aims to serve this goal by providing a novel and flexible framework for orchestrating and running co-simulations. The OSP, which the author also has contributed to, may also be used for this purpose. However, the novel ECS architecture applied by Vico additionally allows models from different sources to be more easily and dynamically combined, and provides an intuitive way to separate behavior and state, which will allow the fidelity of simulations to be changed during run-time. Vico also tries to accommodate the goal of making co-simulations intuitively presentable to the user, as stated by RO4, by enabling user configurable 3D visualizations and 2D plots. It also allows the simulation to be accessed from a web interface, all in one unified package. Furthermore, Vico allows users to interact with the simulation using peripheral devices, enabling user controllable and dynamic simulations. However, even with open standards, like the FMI, model re-use, interoperability and scalability is still an issue in practice. FMU-proxy aims to solve these challenges, thus fulfilling RO3. By packaging existing FMUs into enhanced FMUs with network capabilities, models otherwise incompatible with some tool becomes usable. This feature is shown to be necessary in order to realize the Gunnerus system presented in Chapter 5, not only in Vico, but other frameworks like the OSP and OMSimulator.

The main contributions of this thesis are as follows:

- ✓ Development of tools that simplify interaction with the FMI and SSP standards, driving adoption:
  - (a) FMI4j introduced in Paper A1 enables import and export of FMUs on the JVM, adding very little performance overhead compared to native solutions.
  - (b) PythonFMU introduced in Paper A4 allows exporting of FMUs from Python code, lowering the barrier for model creation.
  - (c) SSPgen makes creating systems that conform to the SSP standard easier through a type-safe scripting context. Additionally, it allows the OSP-IS features, currently only available to libcosim users, to be applied within an SSP context.
  - (d) FMI4cpp eases the process of simulating FMUs using C++, by providing a library that is both easy to build and use, due to usage of modern C++ idioms.
- ✓ FMU-proxy, covered by Papers A2 and A3, solves practical challenges related to FMUs, including
  - (a) instantiating multiple instances of a model that otherwise would not allow it;
  - (b) interaction with models that would otherwise not work on a particular system, due to an incompatible operating system, missing software components or missing software licences;
  - (c) interaction with FMI 1.0 models in tools that only support version 2.0;
  - (d) effectively hiding IP by potentially only providing access to the model trough a remote API; and
  - (e) remote execution of FMUs, enabling scalability and protection from potential malicious code.
- ✓ Contributions to the OSP project that include general insights into discussions and issues in addition to
  - (a) minor contributions to the overall code-base;
  - (b) integration with FMU-proxy, enabling additional use-cases;
  - (c) enhancements to the SSP support; and
  - (d) development of the *cosim4j* package, enabling use of the underlying *libcosim* library from within the JVM.
- ✓ Developments toward realizing a digital twin of the R/V Gunnerus using open co-simulation standards as shown in Paper A5 that presents
  - (a) usage of pre-existing reference models in combination with supplementary models developed as part of this work; and
  - (b) preliminary simulation results utilizing the OSP co-simulation platform.
- ✓ Development of novel co-simulation framework based on the ECS architecture presented in Paper A6 that
  - (a) supports the FMI 1.0 & 2.0 for co-simulation and SSP 1.0;

(b) enables generic co-simulation couplings between components;
(c) intuitively separates behaviour and state;
(d) integrates with physics engines;
(e) provides 3D visual and plotting capabilities; and
(f) allows fully scriptable scenarios.

6.2 Directions for future work

- Among other things, this work has introduced FMI4j and PythonFMU, which enables the building of FMUs in Java and Python respectively. While easy to use, they are limited to their respective languages. Additionally, FMUs implemented in Java and especially Python are slower than comparable models implemented in C/C++. However, no open-source alternative currently exists that allows FMUs to be seamlessly written in C/C++. FMUSDK [55] and CPPFMU [56] make this process easier, but neither delivers a complete package. Rather, they simplify the process of generating the shared library required, but do not generate the *modelDescription.xml* or package the model into a ready-to-use FMU archive. Therefore it would be beneficial to expand the scope of FMI4cpp to also include FMU export, which would further a key goal of the thesis, which is to drive adoption and deliver enabling technology for co-simulation.

- The co-simulation platform Vico, developed by the author to support this thesis, is written in Kotlin and is targeting the JVM. However, Kotlin is actually a multi-platform language that also supports native and web targets. With some additional work, it would be possible to also make the core parts of Vico multi-platform, meaning that one could utilize Vico in a native application using Kotlin/Native and in a browser using Kotlin/JS in addition to Kotlin/JVM. This would certainly make Vico stand out, however there should be use-cases available to support such an effort.

- The effectiveness of a co-simulation platform is largely dependent on the quality of the master algorithm(s) provided. So far, both Vico and libcosim only provides a fixed-step type algorithm with no ability to roll-back the simulation. Further work should focus on adding more advanced algorithms in order to take advantage of the potential benefits for systems that supports it.

- FMI 3.0 has been under development for quite some time and its release is expected in the near future. At this point the tools developed here should be updated to support the new standard. The changes between version 3.0 and 2.0 are, however, much bigger than those between version 1.0 and 2.0, so implementing 3.0 while retaining compatibility with older versions will be challenging. Nonetheless, version 3.0 brings some much awaited features, like arrays, that should make working with FMUs more natural.

References

[1] Modelica Association, “Fmi 2.0 specification,” 2012. (Date accessed 05-October-2020).

[2] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: a survey,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–33, 2018.

[3] P.-L. Sanchez-Gonzalez, D. Díaz-Gutiérrez, T. J. Leo, and L. R. Núñez-Rivas, “Toward digitalization of maritime transport?,” *Sensors*, vol. 19, no. 4, p. 926, 2019.

[4] B. P. Sullivan, S. Desai, J. Sole, M. Rossi, L. Ramundo, and S. Terzi, “Maritime 4.0–opportunities in digitalization and advanced manufacturing for vessel development,” *Procedia Manufacturing*, vol. 42, pp. 246–253, 2020.

[5] A. Rasheed, O. San, and T. Kvamsdal, “Digital twin: Values, challenges and enablers from a modeling perspective,” *IEEE Access*, vol. 8, pp. 21980–22012, 2020.

[6] A. L. Ellefsen, “A data-driven prognostics and health management system for autonomous and semi-autonomous ships,” 2020.

[7] S. Sadjina, L. T. Kyllingstad, M. Rindarøy, S. Skjong, V. Æsøy, and E. Pedersen, “Distributed co-simulation of maritime systems and operations,” *Journal of Offshore Mechanics and Arctic Engineering*, vol. 141, no. 1, 2019.

[8] N. Pedersen, J. Madsen, and M. Vejlgaard-Laursen, “Co-simulation of distributed engine control system and network model using fmi & scnsl,” *IFAC-PapersOnLine*, vol. 48, no. 16, pp. 261–266, 2015.

[9] S. Skjong, M. Rindarøy, L. T. Kyllingstad, V. Æsøy, and E. Pedersen, “Virtual prototyping of maritime systems and operations: applications of distributed co-simulations,” *Journal of Marine Science and Technology*, vol. 23, no. 4, pp. 835–853, 2018.

[10] Y. Chu, L. I. Hatledal, H. Zhang, V. Æsøy, and S. Ehlers, “Virtual prototyping for maritime crane design and operations,” *Journal of marine science and technology*, vol. 23, no. 4, pp. 754–766, 2018.

[11] F. Perabo, D. Park, M. K. Zadeh, Ø. Smogeli, and L. Jamt, “Digital twin modelling of ship power and propulsion systems: Application of the open simulation platform (osp),” in *2020 IEEE 29th International Symposium on Industrial Electronics (ISIE)*, pp. 1265–1270, IEEE, 2020.

[12] A. Bekker, “Exploring the blue skies potential of digital twin technology for a polar supply and research vessel,” in *Proceedings of the 13th International Marine Design Conference Marine Design XIII (IMDC 2018)*, vol. 1, pp. 135–146, 2018.

[13] S. Sadjina, L. T. Kyllingstad, S. Skjong, and E. Pedersen, "Energy conservation and power bonds in co-simulations: non-iterative adaptive step size control and error estimation," *Engineering with Computers*, vol. 33, no. 3, pp. 607–620, 2017.

[14] S. Skjong and E. Pedersen, "On the numerical stability in dynamical distributed simulations," *Mathematics and Computers in Simulation*, vol. 163, pp. 183–203, 2019.

[15] 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, pp. 329–335, Linköping University Electronic Press, 2017.

[16] J. Bastian, C. Clauß, S. Wolf, and P. Schneider, "Master for co-simulation using fmi," in *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, no. 63, pp. 115–120, Linköping University Electronic Press, 2011.

[17] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–12, IEEE, 2013.

[18] T. Schierz, M. Arnold, and C. Clauß, "Co-simulation with communication step size control in an fmi compatible master algorithm," in *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, no. 076, pp. 205–214, Linköping University Electronic Press, 2012.

[19] B. Van Acker, J. Denil, H. Vangheluwe, and P. De Meulenaere, "Generation of an optimised master algorithm for fmi co-simulation.," in *SpringSim (TMS-DEVS)*, pp. 205–212, 2015.

[20] C. Gomes and H. Vangheluwe, "Co-simulation of continuous systems: a hands-on approach," in *2019 Winter Simulation Conference (WSC)*, pp. 1469–1481, IEEE, 2019.

[21] Y. Chu, "Virtual prototyping for marine crane design and operations," 2018.

[22] B. Mishra, "Applying digital twin in shipping and maritime," 2020. (Date accessed 15-October-2020).

[23] T. Blockwitz, 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*, 2012.

[24] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui, "An empirical survey on co-simulation: Promising standards, challenges and research needs," *Simulation modelling practice and theory*, vol. 95, pp. 148–163, 2019.

[25] Ø. Smogeli, L. Bruun, L. Jamt, B. Vik, H. Nordahl, L. Kyllingstad, K. Yum, and H. Zhang, "Open simulation platform – an open-source project for maritime system co-simulation," *19th International Conference on Computer and IT Applications in the Maritime Industries*, pp. 239–253, 2020.

[26] C. Dad, S. Vialle, M. Caujolle, J.-P. Tavella, and M. Ianotto, "Parallelization, distribution and scaling of multi-simulations on multi-core clusters, with daccosim environment," 2016.

[27] R. M. Fujimoto, "Parallel and distributed simulation systems," in *Proceeding of the 2001 Winter Simulation Conference (Cat. No. 01CH37304)*, vol. 1, pp. 147–157, IEEE, 2001.

[28] A. Becue, E. Maia, L. Feeken, P. Borchers, and I. Praca, "A new concept of digital twin supporting optimization and resilience of factories of the future," *Applied Sciences*, vol. 10, no. 13, p. 4482, 2020.

[29] S. Skjong, "Modeling and simulation of maritime systems and operations for virtual prototyping using co-simulations," 2017.

[30] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, "Shaping the digital twin for design and production engineering," *CIRP Annals - Manufacturing Technology*, vol. 66, no. 1, pp. 141–144, 2017.

[31] V. H. Nguyen, Y. Besanger, Q. T. Tran, T. L. Nguyen, *et al.*, "On conceptual structuration and coupling methods of co-simulation frameworks in cyber-physical energy system validation," *Energies*, vol. 10, no. 12, p. 1977, 2017.

[32] F. Cremona, E. Lee, M. Lohstroh, M. Masin, D. Broman, and S. Tripakis, "Hybrid co-simulation: It's about time," in *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, 14 October 2018 through 19 October 2018*, Association for Computing Machinery, Inc, 2018.

[33] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, "The department of defense high level architecture," in *Proceedings of the 29th conference on Winter simulation*, pp. 142–149, 1997.

[34] S. Straßburger, "Overview about the high level architecture for modelling and simulation and recent developments," *Simulation News Europe*, vol. 16, no. 2, pp. 5–14, 2006.

[35] F. Yilmaz, U. Durak, K. Taylan, and H. Oğuztüzün, "Adapting functional mockup units for hla-compliant distributed simulation," in *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, no. 096, pp. 247–257, Linköping University Electronic Press, 2014.

[36] A. Falcone and A. Garro, "Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface," *Simulation Modelling Practice and Theory*, vol. 97, p. 101967, 2019.

[37] 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, pp. 35–42, Linköping University Electronic Press, 2016.

[38] 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*, pp. 1–14, 2018.

[39] Open Simulation Platform, "Open simulation platform - joint industry project for the maritime industry," 2020. (Date accessed 01-October-2020).

[40] D. Adam, "Entity systems are the future of mmog development," 2007. (Date accessed 31-August-2020).

[41] D. Wiebusch and M. E. Latoschik, "Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems," in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 25–32, IEEE, 2015.

[42] P. Lange, R. Weller, and G. Zachmann, "Wait-free hash maps in the entity-component-system pattern for realtime interactive systems," in *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 1–8, IEEE, 2016.

[43] T. Raffaillac and S. Huot, "Polyphony: Programming interfaces and interactions with the entity-component-system model," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. EICS, pp. 1–22, 2019.

[44] 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.

[45] L. I. Hatledal, "A flexible network interface for a real-time simulation framework," Master's thesis, NTNU, 2017.

[46] M. Gull, O. F. Ergin, and S. Savas, "Control of quarter car model by co-simulation with adams and matlab," *International Journal for Research in Applied Science & Engineering Technology*, vol. 6, 2018.

[47] T. Lundberg, "Analysis of simplified dynamic truck models for parameter evaluation," 2013.

[48] P. Li and Q. Yuan, "Influence of coupling approximation on the numerical stability of explicit co-simulation," *Journal of Mechanical Science and Technology*, pp. 1–10, 2020.

[49] M. Arnold, C. Clauß, and T. Schierz, "Error analysis and error estimates for co-simulation in fmi for model exchange and co-simulation v2. 0," in *Progress in Differential-Algebraic Equations*, pp. 107–125, Springer, 2014.

[50] B. Rokseth, S. Skjong, and E. Pedersen, "Modeling of generic offshore vessel in crane operations with focus on strong rigid body connections," *IEEE Journal of Oceanic Engineering*, vol. 42, no. 4, pp. 846–868, 2016.

[51] S. Skjong and E. Pedersen, "A real-time simulator framework for marine power plants with weak power grids," *Mechatronics*, vol. 47, pp. 24–36, 2017.

[52] T. I. Fossen and T. Perez, "Kalman filtering for positioning and heading control of ships and offshore rigs," *IEEE control systems magazine*, vol. 29, no. 6, pp. 32–46, 2009.

[53] R. Pascoal and C. G. Soares, "Kalman filtering of vessel motions for ocean wave directional spectrum estimation," *Ocean Engineering*, vol. 36, no. 6-7, pp. 477–488, 2009.

[54] T. Iseki, D. Terada, *et al.*, "Bayesian estimation of directional wave spectra for ship guidance system," *International Journal of Offshore and Polar Engineering*, vol. 12, no. 01, 2002.

[55] QTronic, "Fmu sdk," 2017. (Date accessed 05-October-2020).

[56] Viproma, "cppfmu," 2016. (Date accessed 05-October-2020).

Appendix

59

A

Paper A1

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

	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	2.0.4	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/`

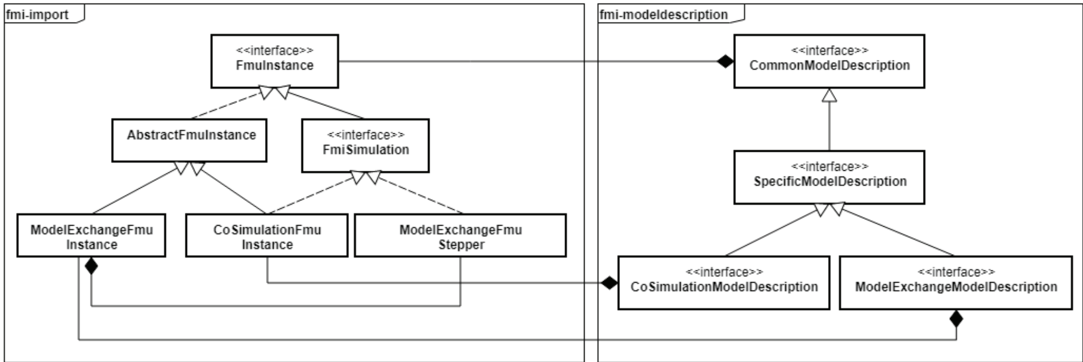


Figure 1. Simplified UML view of core FMI4j classes.

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.

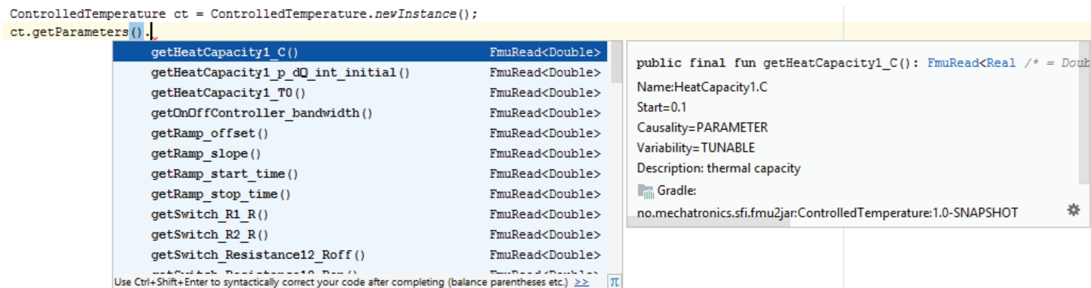


Figure 2. FMU named *ControlledTemperature* wrapped as a Java library using FMU2jar, then imported into IntelliJ IDE. From within the IDE, the user can browse and read documentation on all available variables.

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.

B

Paper A2

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 rollback 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:

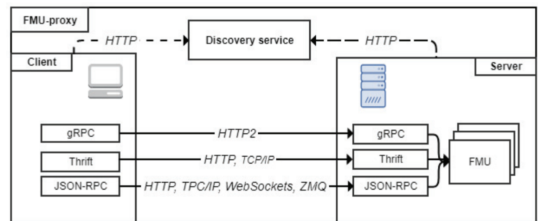


Figure 1. Software architecture.

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.

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.

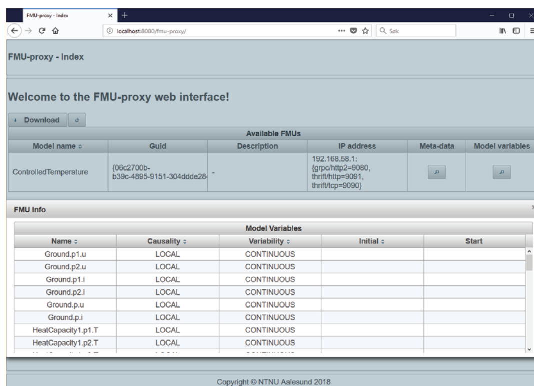


Figure 2. The discovery service's web interface. Here available FMUs are listed, showing network information and data from the *modelDescription.xml*.

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.

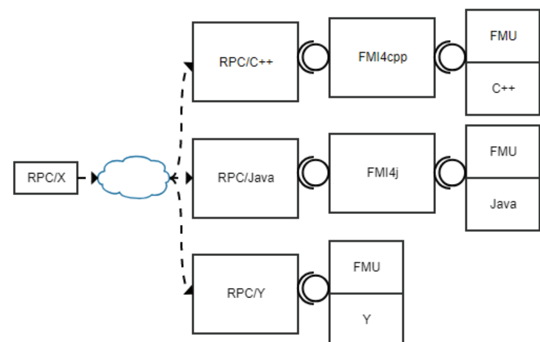


Figure 3. FMI4cpp and FMI4j's slave interface could hide slaves stemming from either an in-memory implementation or an actual FMU. A slave in any language supported by the chosen RPC could also be implemented directly behind the RPC layer.

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

DOI
10.3384/ecp1915779

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

C

Paper A3

IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received July 1, 2019, accepted July 29, 2019, date of publication August 5, 2019, date of current version August 21, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2933275

A Language and Platform Independent Co-Simulation Framework Based on the Functional Mock-Up Interface

LARS IVAR 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, 6009 Ålesund, Norway
[2]Department of ICT and Natural Sciences, Norwegian University of Science and Technology, 6009 Ålesund, Norway
[3]Department of Engineering Sciences, University of Agder, 4879 Grimstad, Norway

Corresponding author: Lars Ivar Hatledal (laht@ntnu.no)

This work was supported by the Norwegian Research Council, SFI Offshore Mechatronics, under Project 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 instance, a simulator may impose one or more requirements

The associate editor coordinating the review of this manuscript and approving it for publication was Orazio Gambino.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

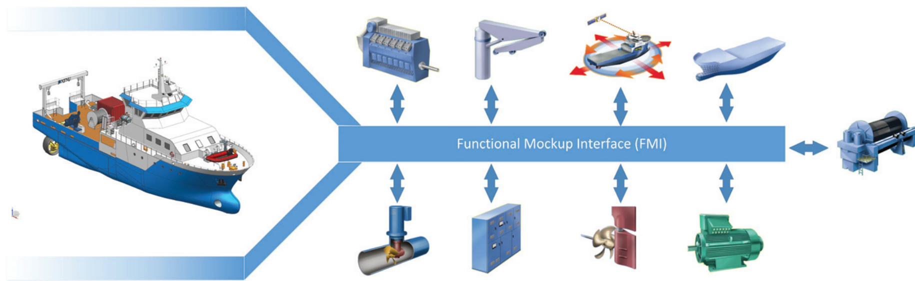


FIGURE 1. Simulation of a complex cyber-physical system in the maritime domain. The complete vessel model is constituted by the individual sub-modules connected through FMI for Co-simulation. *Sub-model figures courtesy of the Virtual Prototyping of Maritime Systems and Operations project (Research Council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

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 sandboxed 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

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

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

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

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 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 barebones 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 metadata 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.

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

message HelloResponse {
  string reply = 1;
}

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

Listing. 1. Example protobuf schema with service definitions.

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.

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

Listing. 2. Example Thrift schema.

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

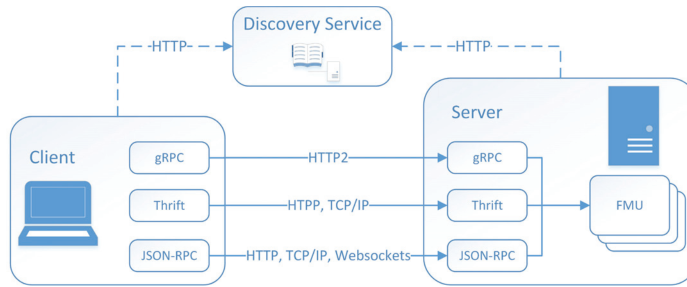


FIGURE 2. The high level software architecture of the proposed framework. The client-server architecture relies on RPCs for communication. The Discovery Service is optional, and serves as a centralized hub for locating available FMUs.

```
-> {"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}
```

Listing. 3. Example JSON RPC call.

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.

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.

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

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

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

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

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 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 sandboxed 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

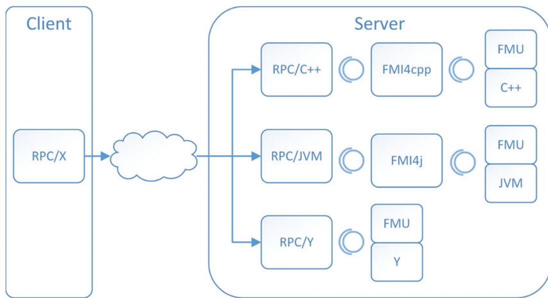


FIGURE 3. FMI4cpp and FMI4j's slave interface could hide slaves derived from either an in-memory implementation or an actual FMU. Slaves in any language supported by the chosen RPC could also be implemented directly behind the RPC layer.

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.

```
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()
```

Listing. 6. JVM Thrift example, written in Kotlin.

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.

```
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)
```

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

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.

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 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

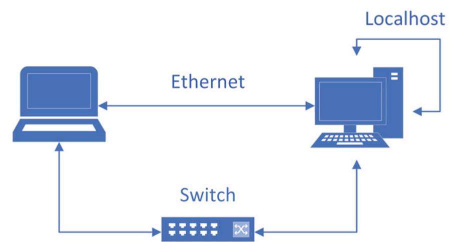


FIGURE 4. The different network topologies used in the case study.

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

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.

Running in parallel provides quite a significant performance gain, moving from a $\sim 9.5\times$ to a $\sim 3.6\times$ performance

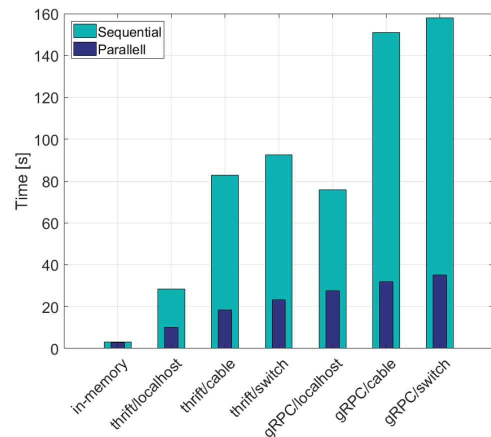


FIGURE 5. Bar plot of the results shown in Table. 3.

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 sand-boxed 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

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

Tool	Version	Name	Step size [s]	Stop time [s]	No. calls	*Time elapsed [ms]*						
						API	***Thrift***			***gRPC***		
						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**

network call becomes painfully obvious. Compare this to the 20Sim model, which only requires 126 invocations to simulate 12.56*s*. 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 Comput. Surv.*, 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," *Simul. Model. Pract. Theory*, vol. 43, pp. 16–33, Apr. 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*. Cham, Switzerland: Springer, 2016, pp. 3–28.
[4] Y. Chu, L. I. Hatledal, F. Sanfilippo, V. Æsøy, H. Zhang, and H. G. Schaathun, "Virtual prototyping system for maritime crane design and operation based on functional mock-up interface," in *Proc. OCEANS*, Genoa, Italy, May 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 *Proc. 35th Int. Conf. Ocean Offshore Arctic Eng.*, 2016, Art. no. 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," 2017, *arXiv:1701.00997*. [Online]. Available: https://arxiv.org/abs/1701.00997
[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. 20531–20544, 2018.
[8] M. Faruque, V. Dinavahi, M. Steurer, A. Monti, K. Strunz, J. A. Martinez, G. W. Chang, J. Jatskevich, R. Iravani, and A. Davoudi, "Interfacing issues in multi-domain simulation tools," *IEEE Trans. Power Del.*, vol. 27, no. 1, pp. 439–448, Jan. 2012.
[9] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, and D. Neumerkel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proc. 9th Int. MODELICA Conf.* Munich, Germany: Linköping Univ. Electronic Press, Sep. 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 *Proc. 13th Int. Modelica Conf.* Regensburg, Germany: Linköping Univ. Electronic Press, Mar. 2019, pp. 79–86.
[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 *Proc. 12th Int. Modelica Conf.* Prague, Czech Republic: Linköping Univ. Electronic Press, May 2017, pp. 329–335.
[12] JModelica. (2017). *Fmi Library*. Accessed: May 16, 2019 [Online]. Available: http://www.jmodelica.org/FMILibrary

[13] QTronic. (2014). *Fmu Sdk*. Accessed May 16, 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 *Proc. Workshop Modeling Simulation Cyber-Phys. Energy Syst. (MSCPES)*, May 2013, pp. 1–6.

[15] L. I. Hatledal. (2018). *FMI4cpp*. Accessed May 16, 2019. [Online]. Available: https://github.com/FMU-proxy/FMI4cpp

[16] D. Systems. (2017). *FMPy*. Accessed May 16, 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," Dept. Math. Sci., Lund Univ. Fac. Eng. Centre Math. Sci. Numer. Anal., Sölvegatan, Sweden, Tech. Rep. 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 *Proc. 59th Conf. Simulation Modelling (SIMS)*. Oslo, Norway: Linköping Univ. Electronic Press, Sep. 2018, pp. 37–42.

[19] J. S. C. Montenegro, "JavaFMI una librería Java para el estándar functional mockup interface," Univ. Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain, Tech. Rep., 2014. [Online]. Available: https://accedacris.ulpgc.es/handle/10553/12681

[20] D. Broman, C. Brooks, E. A. Lee, T. S. Nouidui, S. Tripakis, and M. Wetter. (2013). *JFMI—A Java Wrapper for the Functional Mock-Up Interface*. Accessed: May 16, 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 *Proc. 13th Int. Modelica Conf.* Regensburg, Germany: Linköping Univ. Electronic Press, Mar. 2019, pp. 785–794.

[22] C. Lacoursière and T. Härdin, "FMI Go! A simulation runtime environment with a client server architecture over multiple protocols," in *Proc. 12th Int. Modelica Conf.* Prague, Czech Republic: Linköping Univ. Electronic Press, May 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 *Proc. 56th Conf. Simulation Modelling (SIMS)*. Linköping, Sweden: Linköping Univ. Electronic Press, Oct. 2015, pp. 7–9,

[24] E. Pitt and K. McNiff, *Java Rmi The Remote Method Invocation Guide*. Reading, MA, USA: Addison-Wesley, 2001.

[25] M. Krammer, M. Benedikt, T. Blochwitz, K. Alekeish, N. Amringer, C. Kater, S. Materne, R. Ruvalcaba, K. Schuch, J. Zehetner, M. Damm-Norwig, V. Schreiber, N. Nagarajan, I. Corral, T. Sparber, S. Klein, and J. Andert, "The distributed co-simulation protocol for the integration of real-time systems and simulation environments," in *Proc. 50th Comput. Simulation Conf.*, 2018, p. 1.

[26] K. Varda. (Jul. 2008). Protocol Buffers: Google' S Data Interchange Format. Google Open Source Blog. Accessed: Nov. 8, 2019. Online: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html

[27] gRPC. *GRPC 018*. Accessed: May,16, 2019. [Online]. Available: https://grpc.io/

[28] Apache Software Foundation. (2019). *Apache Thrift*. Accessed: May, 16, 2019. [Online]. Available: https://thrift.apache.org/

[29] JSON-RPC Working Group and Others. (2012). *Json-Rpc 2.0 Specification*. Accessed: Mar. 27 2019. [Online]. Available: https://www.jsonrpc.org/specification

[30] Apache Foundation. (2019). *Apache Commons Math3*. Accessed: Mar. 27 2019. [Online]. Available: http://commons.apache.org/proper/commons-math/

[31] JNA Authors. (2018). *JNA FAQ*. Accessed: Mar. 27, 2019. [Online]. Available: https://github.com/java-native-access/jna/blob/5.2.0/www/FrequentlyAskedQuestions.md

[32] Boost Developers. (2019). *Apache Commons Math3*. Accessed Mar. 27, 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 *Proc. 10th Int. Modelica Conf.* Lund, Sweden: Linköping Univ. Electronic Press, Mar. 2014, pp. 27–33.

[34] OSP. (2019). *Open Simulator Platform*. Accessed: May 16, 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 *Proc. 1st Jpn. Modelica Conf.* Tokyo, Japan: Linköping Univ. Electronic Press, May 2016, pp. 35–42.

LARS IVAR HATLEDAL received the B.Sc. degree in automation and the M.Sc. degree in simulation and visualization from the Norwegian University of Science and Technology (NTNU), Ålesund, Norway, in 2013 and 2017, respectively, where he is currently pursuing the Ph.D. degree with the Department of Ocean Operations and Civil Engineering. After his graduation, he started working as a part-time Research Assistant with the Mechatronics Laboratory, Department of Ocean Operations and Civil Engineering, NTNU, Ålesund. His research interests include simulation, artificial intelligence, and 3D visualization.

ARNE STYVE received the B.E. degree (Hons.) in microelectronics and software engineering from the University of Newcastle upon Tyne, Newcastle upon Tyne, U.K., in 1991. In 2004, he joined the Offshore Simulator Centre AS (OSC), where he was a Research and Development Manager until his return to NTNU, Ålesund, Norway. He is currently an Assistant Professor with the Department of ICT and Natural Sciences, NTNU, Ålesund. 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).

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 with ABB Norway, Sweden, and Switzerland (Oslo, Västerås, and Baden, respectively), from 1997 to 2003, and took part in the development of ABBs control system for industrial robots. He was a Senior Lecturer in mechatronics with The University of Queensland, Brisbane, QLD, Australia, from 2004 to 2006, and he 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, Grimstad, and a Technical Manager of the Norwegian Motion Laboratory, Grimstad. He is the Chief Editor of the *MIC Journal*.

HOUXIANG ZHANG (M'04–SM'12) received the Ph.D. degree in mechanical and electronic engineering, in 2003, and the Habilitation degree in informatics from the University of Hamburg, in February 2011. Since 2004, he has been with the Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, Institute of Technical Aspects of Multimodal Systems (TAMS), University of Hamburg, Germany. He joined the NTNU, Norway, in April 2011, where he is currently a Professor of robotics and cybernetics. His research interests lies on two areas: one is on biological robots and modular robotics and the other is on virtual prototyping and maritime mechatronics. In these areas, he has published over 160 journals and conference papers as author or coauthor. He has applied for and coordinated more than 20 projects supported by the Norwegian Research Council (NFR), German Research Council (DFG), and industry. He has received four best paper awards and four finalist awards for the Best Conference Paper at International conference on Robotics and Automation.

Paper A4

ENABLING PYTHON DRIVEN CO-SIMULATION MODELS WITH PYTHONFMU

Hatledal, Lars Ivar*
Zhang, Houxiang
Department of Ocean Operations and Civil Engineering
Norwegian University of Science and Technology
Postbox 1517, 6025 Aalesund, Norway

Collonval, Frédéric
Modeling & Simulation
Safran Tech
CS80112 Chateaufort
78772 Magny Les Hameaux, France

KEYWORDS

Co-simulation; Modelling; FMI; FMU; Python

ABSTRACT

This paper introduces PythonFMU, an easy to use framework for exporting Python 3.x code as co-simulation compatible models compliant with version 2.0 of the Functional Mock-up Interface (FMI). The framework consists of a set of helper classes and a command line utility for transforming compliant python source into ready to use cross-platform FMUs. PythonFMU seamlessly takes care of a number of low-level FMI functions such as getting and setting variable values, and state handling, including serialization and deserialization. Furthermore it provides pre-built binaries for Windows and Linux 64-bits, generates the required *modelDescription.xml* containing meta-data about the model and packages all related files into a Functional Mock-up Unit (FMU) - ready to be imported into any FMI compatible simulation tool. The framework can be effortlessly installed using de-facto standard Python package managers pip and conda. While PythonFMU is more geared towards ease of use and enabling Python driven co-simulation models, it is shown to have adequate performance compared to much more low-level alternatives targeting other programming languages.

INTRODUCTION

The Functional Mock-up Interface (FMI) [Blochwitz et al., 2012] is a tool independent standard managed by the Modelica Association that supports both Model Exchange (ME) and Co-Simulation (CS) of dynamic models. A key goal of FMI is to improve the exchange of simulation models between suppliers and original equipment manufacturers (OEMs). The current major version of the standard is 2.0, which was released in 2014. A minor revision, 2.0.1, was released in 2019.

An FMU is a model that implements the FMI standard 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, 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 work presented in this paper, however, is only concerned about the co-simulation part of the standard.

Many tools support importing co-simulation FMUs, however, fewer tools supports exporting such FMUs. Many of whom are commercial and or domain specific. Furthermore, FMUs generated with these tools may not support the optional parts of the standard such as state handling, which are required by some more advanced co-simulation algorithms in order to achieve better numerical accuracy and stability during simulations[Broman et al., 2013, Cremona et al., 2016, Tavella et al., 2016].

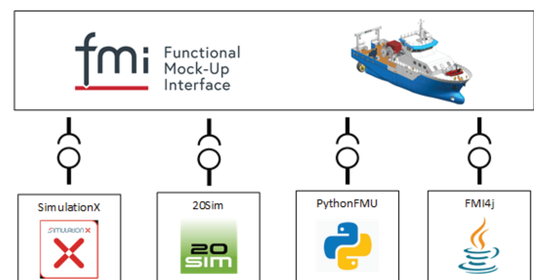


Fig. 1: Possible use of PythonFMU in realizing complex cyber-physical systems using FMI based co-simulation.

Python [Van Rossum et al., 2007] is one of the most popular programming language today [O'Grady, 2020]. The major reasons for that are the ease of learning the language, the huge spectra of libraries covering fields

*Corresponding author. E-mail: laht@ntnu.no

Communications of the ECMS, Volume 34, Issue 1,
Proceedings, ©ECMS Mike Steglich, Christian Mueller,
Gaby Neumann, Mathias Walther (Editors)
ISBN: 978-3-937436-68-5/978-3-937436-69-2(CD) ISSN 2522-2414

such as video game, machine learning, web server or scientific computing and the recent explosion of data science in which Python plays a central role. Of particular importance for scientific computing is the creation of the *Numpy*[Oliphant, 2007] library that bridge the gap between efficient code in C or Fortran languages and the ease of a scripting language. That library is now at the heart of all major scientific Python libraries from Pandas for data analysis to Scipy for classical algebra operators or Scikit-learn for machine learning analysis.

This paper introduces PythonFMU, an easy to use Python based framework that allows plain Python code to be exported as FMI compatible co-simulation FMUs. Figure. 1 shows how PythonFMU could potentially be used to implement complex cyber-physical systems that are aggregates of models from different simulation domains.

The paper is organized as follows. Firstly some related works are given. After which PythonFMU is introduced. Then some benchmark results are presented. Finally some concluding remarks and notes on future works are provided.

RELATED WORK

A number of open-source software frameworks for exporting FMUs from source-code have been developed in the recent years. While many more tools are capable of exporting FMUs like 20Sim, OpenModelica, MATLAB and SimulationX, this paper is more focused on frameworks that allows the generation of FMUs from plain source-code. Each of these are described in more detail below.

CPPFMU [SINTEF Ocean, 2017] is a set of interfaces and helper functions for writing FMI-compliant model/slave code in C++ using high-level features such as exceptions and automatic memory management, rather than having to implement the low-level C functions specified by FMI. However, while CPPFMU makes implementing and compiling the shared library required by an FMU, it does not handle generating the *modelDescription.xml* nor packaging of the FMU. CPPFMU was developed as part of the R&D project Virtual Prototyping of Maritime Systems and Operations (ViProMa) Hassani et al. [2016], and is currently maintained by SINTEF Ocean.

FMUSDK [QTronic, 2017] is a free, BSD licensed, software development kit (SDK) provided by QTronic to demonstrate basic use of FMUs for ME and for CS as defined by FMI version 1.0 and 2.0. The software is written in C++, but models are to be implemented in C. The first version of FMU SDK was released already in 2010, with the latest version, *2.0.6*, being released in 2018.

Like CPPFMU, FMUSDK does not auto-generate the *modelDescription.xml*. The main difference between these tools is that CPPFMU provides a more high level and structured API in C++, whereas FMUSDK requires source-code to be written in quite low-level C. On the other hand, FMUSDK supports ME and provides utilities for packaging the model as an FMU, whereas CPPFMU only provides helper functions to aid in development.

JavaFMI [Galtier et al., 2017] is a set of components for working with the FMI standard using Java, developed by SIANI institute (Las Palmas University) and funded by the European Institute for Energy Research (EIFER). It support both import and export of FMUs. For export, it support FMI 2.0 for Co-simulation. Generated FMUs runs both on Linux and Windows. JavaFMI has been actively maintained since its inception in 2013 and is licensed under the LGPLv3.

FMI4j [Hatledal et al., 2018] is a MIT licensed software package for dealing with Functional Mock-up Units (FMUs) on the JVM. It support both import and export of FMUs. For export, it support FMI 2.0 for Co-simulation. FMI4j is written in Kotlin, which is 100% interoperable with Java. On the native side, FMI4j makes use of CPPFMU to implement the FMI functions. FMUs generated using FMI4j can run on both Linux and Windows.

While both JavaFMI and FMI4j allows FMUs to be created using the Java language, they differ quite a bit in their implementation and usage. JavaFMI uses message-passing to bridge Java and the underlying C functions defined by FMI, while FMI4j relies on the Java Native Interface (JNI) for this. Consequentially, FMI4j generates much faster executables. Another key difference is how users define their model. JavaFMI is imperative, e.g meta-data is defined using API functions. FMI4j on the other hand is declarative, with meta-data defined using annotations.

Evidently, some open-source software for generating FMUs from source code already exists. See Table. I for a summary. However, only the ones targeting the JVM can be said to be easy to use as these manages everything related to the creation of an FMU. Still, the JVM may not be a natural choice for many for implementing models and the barrier for using these tools are high for non Java developers. CPPFMU and FMUSDK both eases the process within the realm of C/C++, but still requires significant know-how in order to produce a ready to use FMU. Furthermore, these tools only covers a small subset of available programming languages.

TABLE I: Open-source framework for exporting source-code as FMUs.

Tool	Target language	Target platform	FMI version
JavaFMI	JVM	Win, Linux	2.0
FMI4j	JVM	Win, Linux	2.0
CPPFMU	C++	Win[a], Linux[a]	1.0 & 2.0
FMUSDK	C	Win[a], Linux[a], OSX[a]	1.0 & 2.0

[a] Binaries are only built for the current platform.

PYTHONFMU

PythonFMU is a MIT licensed framework that enables the packaging of Python 3.x code as co-simulation FMUs, currently maintained in collaboration between NTNU and Safran Tech. The library required by users to implement their own FMI co-simulation slaves as well as the Command Line Interface (CLI) required to build the actual FMU is easily retrieved using either the *pip* or *conda* package managers. Unlike some FMU exporters, FMUs built with PythonFMU runs out of the box on both Windows and Linux 64-bit systems. PythonFMU has been implemented using the limited Python API, which makes it compatible with any Python 3.x version. However, PythonFMU does not bundle a Python distribution, which means that a compatible Python distribution must be present on the target system for the FMU to work. The same is true for any imported 3rd party libraries. Consequentially, if the slave makes use of e.g. the *numpy* package for scientific computing, this library must already be present on the target system. To remedy this, PythonFMU allows users to specify any dependency it should have on 3rd party libraries. This information is bundled with the FMU as a standard *requirements.txt* for use with one of Python's package managers. Thus allowing end-users to easily figure out what kind of libraries that must be installed for it to run on a particular machine.

Listing 1: Writing FMI 2.0 compatible slaves in Python using PythonFMU.

```
from pythonfmu import *

class PythonSlave(Fmi2Slave):

    author = "John Doe"
    description = "A simple description"

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.realOut = 0.1
        self.register_variable(Real("realOut",
            causality=Fmi2Causality.output))

    def do_step(current_time, step_size):
        return True
```

Listing. 1 shows the minimal required code to write FMI 2.0 compatible co-simulation models in Python using PythonFMU. Additional FMI functions like e.g. *setupExperiment*, *enterInitializationMode*, *exitInitializationMode* and *terminate* have default no-op implementations and may be overridden on demand. PythonFMU automatically handles getting and setting variables, logging, resetting, state handling, serialization and deserialization as well as generating the required *modelDescription.xml*. The fact that PythonFMU handles state handling makes it possible to use with advanced co-simulation master algorithms that depends on rollback capabilities, like variable step algorithms. This is important in order to achieve numerically stable and accurate simulation results. Listing. 2 shows how to build an FMU from Python source that implements the PythonFMU API using the accompanying CLI. Additional options may be specified, such as documentation and associated project files. The FMU built by PythonFMU contains pre-built binaries for Windows and Linux 64-bit. This lowers the threshold for using it tremendously compared to many exporting tools as a C++ compiler does not have to be installed and the user does not have to figure out how to cross-compile.

Like FMI4j, PythonFMU makes use of CPPFMU for implementing the C functions required by the FMI standard. This shows a clear utility for CPPFMU as an enabler for higher-level applications to support the export of FMI compatible co-simulation models.

Listing 2: Building an FMU from Python source using the PythonFMU CLI.

```
pythonfmu-builder -f PythonSlave.py
```

RESULTS

In the following some performance metrics for PythonFMU is given.

Table. II show the performance of PythonFMU compared to other similar tools. The FMUs used all implements the same model. The model does no computation during stepping, but defines a single real, integer, boolean and string variable. These variables are read by the importing tool after each iteration. 100.000 iterations were run. That makes for a total of 400.000 calls through the FMI API. The benchmark was performed on a computer running Windows 10 fitted with an Intel i7-8700k processor.

TABLE II: Time required to step a simple FMU with one integer, real, string and boolean variable 100.000 times. All variables are read after each step.

Tool	**Version**	**Time[s]**
FMUSDK	2.0.6	4.6
CPPFMU	-	4.6
FMI4j	0.30.0	6.1
JavaFMI	2.6.0	40
PythonFMU	0.6.0	7.9/7.3[a]

[a] Using lambdas for getters, as demonstrated in Listing. 3.

From the results we can see that FMUSDK and CPPFMU are equally fast, and as expected, faster than both FMI4j and PythonFMU. This is natural as both of these uses CPPFMU internally and have an additional overhead from having to cross the native bridge using JNI and the Python C API respectively. JavaFMI is by far the slowest contender, due to it's choice of using message passing over direct API calls through JNI. Note that PythonFMU provides two results. By supplying a lambda function to the optional *getter* and *setter* parameters of PythonFMUs *ScalarVariable* as

demonstrated in Listing. 3, users may increase performance of variable read/write. When not specifying lambda functions for the getter and setter, PythonFMU defaults to using the built in Python functions *getattr* and *setattr* respectively. Furthermore, the use of lambdas allows non Python fields to be used as variables.

Listing 3: Supplying a lambda for increased flexibility and performance at the cost of a slight increase in verbosity.

```
self.register_variable(Real("realOut",
                causality=Fmi2Causality.output
                getter=lambda: self.real))
```

Note that the results presented here does not necessarily translate to more complex models with more code evaluation, as the presented benchmark it is mainly interested in measuring the performance of raw FMI calls. As Python is an interpreted language it is naturally slower to run than e.g. C/C++.

CONCLUSIONS

This paper introduces PythonFMU, an easy to use framework for exporting Python code as FMI 2.0 for co-simulation compatible models. The framework is easy to install and requires very little boilerplate code, allowing users to focus on the problem at hand. This coupled with the fact that Python is an easy to use scripting language with a strong standard library and a rich set of 3rd party libraries makes it ideal for fast prototyping. Furthermore, the position Python has as a language for scientific computing should make PythonFMU a natural choice for data scientists that want to take advantage of or contribute to co-simulation technology. In fact, PythonFMU was specifically developed to allow data scientist with little or no background from co-simulation or software engineering at NTNU to contribute with models related to the development of digital twins, as Python and it's strong ecosystem of libraries allows easy integration with e.g. models that is connected to web services or utilizes neural networks. While the focus of PythonFMU is ease of use and being an enabler for Python driven co-simulation models, the performance is shown to be quite adequate compared to more low-level implementations.

Future works includes adding more features, bug-fixes and improving documentation. The source is available at `https://github.com/NTNU-IHB/PythonFMU`, and users are encourage to contribute.

ACKNOWLEDGMENT

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

REFERENCES

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*, number 076, pages 173–184. Linköping University Electronic Press, 2012.

D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of fmus for co-simulation. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–12. IEEE, 2013.

F. Cremona, M. Lohstroh, D. Broman, M. Di Natale, E. A. Lee, and S. Tripakis. Step revision in hybrid co-simulation with fmi. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 173–183. IEEE, 2016.

V. Galtier, M. Ianotto, M. Caujolle, R. Corniglion, J.-P. Tavella, J. É. Gómez, J. J. H. Cabrera, V. Reinbold, and E. Kremers. Building parallel fmus (or matryoshka co-simulations). In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 663–671. Linköping University Electronic Press, 2017.

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 Digital Collection, 2016.

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 *The 59th Conference on Simulation and Modelling (SIMS 59)*. Linköping University Electronic Press, Linköpings universitet, 2018.

S. O'Grady. The redmonk programming language rankings: January 2020, 2020. URL `https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/`. (Date accessed 08-March-2020).

T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

QTronic. Fmusdk, 2017. URL `http://www.qtronic.de/de/fmusdk.html`. (Date accessed 06-March-2020).

SINTEF Ocean. Cppfmu, 2017. URL `https://github.com/viproma/cppfmu`. (Date accessed 06-March-2020).

J.-P. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. Toward an accurate and fast hybrid multi-simulation with the fmi-cs standard. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–5. IEEE, 2016.

G. Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007.

AUTHOR BIOGRAPHIES

LARS IVAR HATLEDAL received the B.Sc. degree in automation and the M.Sc. degree in simulation and visualization from the Norwegian University of Science

and Technology (NTNU), Ålesund, Norway, in 2013 and 2017, respectively, where he is currently pursuing the Ph.D. degree with the Department of Ocean Operations and Civil Engineering. Email: laht@ntnu.no

DR. FRÉDÉRIC COLLONVAL is the lead developer of a collaborative multi-systems and multi-physics simulation tool, of the Collaborative System Design team at Safran R&D center. He obtained his PhD in numerical simulation and modeling in gas turbine combustion chamber at TU Munich. He then joined Safran Group to work on airplane engine performance modeling and associated simulation tools. In 2018, he co-founded a new kind of collaborative simulation tool to target multi-physics simulation in pre-design phase for rapid design evaluation at Safran. He is interested in enhancing physical simulation tools by leveraging the features of innovative open-source projects.

PROF. HOUXIANG ZHANG received the Ph.D. degree in mechanical and electronic engineering, in 2003, and the Habilitation degree in informatics from the University of Hamburg, in February 2011. Since 2004, he has been with the Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, Institute of Technical Aspects of Multimodal Systems (TAMS), University of Hamburg, Germany. He joined the NTNU, Norway, in April 2011, where he is currently a Professor of robotics and cybernetics. His research interests lies on two areas: one is on biological robots and modular robotics and the other is on virtual prototyping and maritime mechatronics.

E

Paper A5

Modeling, Identification and Control, Vol. 41, No. 4, 2020, pp. 297–311, ISSN 1890–1328

Co-simulation as a Fundamental Technology for Twin Ships

L.I. Hatledal [1] R. Skulstad [1] G. Li [1] A. Styve [2] H. Zhang [1]

[1] *Department of Ocean Operations and Civil Engineering, Norwegian University of Science and Technology, Larsgrdsvegen 2, 6009 Ålesund, Norway. Norway. E-mail:* {laht,robert.skulstad,guoyuan.li,hozh}@ntnu.no

[2] *Department of ICT and Natural Sciences, Norwegian University of Science and Technology, Larsgrdsvegen 2, 6009 Ålesund, Norway. E-mail:* asty@ntnu.no

Abstract

The concept of digital twins, characterized by the high fidelity with which they mimic their physical counterpart, provide potential benefits for the next generation of advanced ships. It allows analysis of data and monitoring of marine systems to avoid problems before they occur, and plan for the future by using simulations. However, issues related to integration of heterogeneous systems and hardware, memory, and CPU utilization makes implementing such a digital twin in a monolithic or centralized manner undesirable. Co-simulation addresses this problem, allowing different sub-systems to be modelled independently, but simulated together. This paper presents the ongoing work towards realizing a digital twin of the Gunnerus research vessel by applying co-simulation and related standards. The paper does not present a complete, ready-to-use digital twin. Rather it presents the preliminary results, procedure, and enabling technologies used towards realizing one. In order to accommodate this goal, a novel co-simulation solution, developed in cooperation by members of the Norwegian maritime industry, is presented. Furthermore, a maneuvering case-study is carried out, utilizing pre-recorded sensor data obtained from the Gunnerus. Through a comparative study with the real maneuver in terms of speed, course, and power consumption, the proposed approach is verified in simulation.

Keywords: Co-simulation, Digital twin, FMI, SSP, R/V Gunnerus

1 Introduction

There is a strong demand for innovation and efficiency within operations, life cycle services, and design of marine systems. Modern marine vessels operate increasingly autonomously through strongly interacting subsystems. These systems are dedicated to a specific, primary objective of the vessel or may be part of the general essential ship operations. The sub-systems exchange data and make coordinated operational decisions, ideally without any user interaction. The task of designing, operating, and integrating life cycle services for such vessels is a complex engineering task that requires an efficient development approach, which must consider the mutual interaction between the inherent multi-disciplinary on-board sub-systems. Digitalization thus has become a key aspect of making the maritime industry more innovative, efficient, and fit for future operations Sanchez-Gonzalez et al. (2019); Sullivan et al. (2020).

A digital twin can be defined as a virtual representation of a physical asset enabled through data and simulators for real-time prediction, optimization, monitoring, controlling, and improved decision making Rasheed et al. (2020). The digital twin should be able to take advantage of all digital information available for an asset, such as the system and data information models, 3D models, mathematical models, de-

doi:10.4173/mic.2020.4.2 © 2020 Norwegian Society of Automatic Control

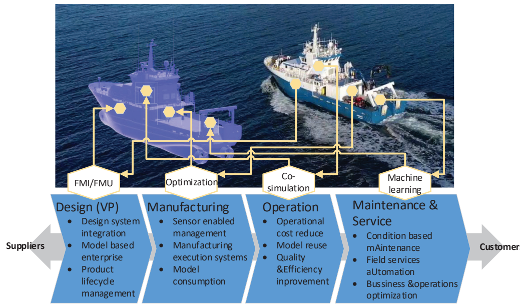


Figure 1: A plausible development procedure of digital twins system for marine industry.

pendability models, condition and performance indicators, and data analytics.

The maritime industry will benefit from digital twin technology Perabo et al. (2020). These proxies of the physical world will help maritime companies in developing enhancements to existing products, operations, and services, and can even help drive the innovation of new businesses. Additional benefits for the maritime industry as a whole is highlighted in Bekker (2018). The eventual goal of this research is to develop digital twins of maritime systems and operations, not only allowing configuration of systems and verification of operational performance, but also to provide early warning, life cycle service support, and system behaviour prediction. As illustrated in Fig. 1, the use of co-simulation together with data related optimization, like data purification, and machine learning methods will be seamlessly combined from the design phase to maintenance phase to achieve heterogeneous simulation, data analytics and behavioural prediction of maritime systems.

As stated in Schleich et al. (2017), the scientific literature has reported that challenges persist in the vision of the implementation of the digital twin, such as insufficient synchronization between the physical and the digital world to establish closed loops, a lack of high-fidelity models for simulation and virtual testing at multiple scales, lacking uncertainty quantification for such models, difficulties related to the prediction of complex systems, and challenges related to the gathering and processing of large data sets. Overcoming these limitations will require a sound conceptual framework and comprehensive reference models. An open platform would ensure that all companies in the surrounding maritime cluster could potentially benefit from and contribute to it. The platform should allow companies to benefit from each other's models and data without necessarily exposing their intellectual property Durling et al. (2017).

In this paper we seek to promote an open-source framework that can leverage the possibilities provided by a digital twin in order to support ongoing work in the Knowledge-building Project for Industry (KPN) *Digital Twins for Vessel Life Cycle Service (TwinShip)* [1]. In order to establish such an open framework for digital twins that enables users to easily develop, integrate, and combine their own components into a complete system, e.g. for the purpose of maritime industry design, operation, service, and maintenance, it is essential to realize effective co-simulation mechanisms and related auxiliary tools. To support the KPN project, DNV GL, Kongsberg Maritime (formerly Rolls-Royce Marine), SINTEF Ocean, and NTNU initiated a Joint Industrial Initiative (JIP) nicknamed the Open Simulation Platform (OSP) Open Simulation Platform (2020) in 2019. The purpose of the OSP is to lay the foundation for an ecosystem where the maritime industry can perform co-simulation and share simulation models in an efficient and secure way. The ultimate goal of the OSP is to facilitate building of digital twin systems and vessels, making it easier to solve challenges related to designing, building, integrating, commissioning, and operating complex integrated systems. Thus it will enable the realization of complex cyber-physical-systems (CPS) like the vessel model illustrated in Figure 2, where the complete vessel model is an aggregation of several independent sub-models that connect through a standardized co-simulation interface.

In this work, we make use of the NTNU owned research vessel (R/V) Gunnerus, as shown in Fig. 1, as the test-bed to demonstrate a sound conceptual framework that uses co-simulation as a fundamental technology towards realizing a digital twin for ship maneuvering. The contributions of the paper include:

1. Employment of a novel co-simulation library as a platform for digital twins.

2. Utilization of a freely available tool-box of marine black-box models, provided by the OSP, in order to accelerate modeling of the Gunnerus.

3. Real-life application of FMU-proxy — enabling co-simulation of otherwise incompatible simulation models.

4. Demonstration of the System Structure and Parameterization (SSP) standard for defining the structure, connections, and the parameterization of the full system to be simulated. Additionally, we demonstrate that components other than

[1] https://org.ntnu.no/intelligentsystemslab/project/twinship.html

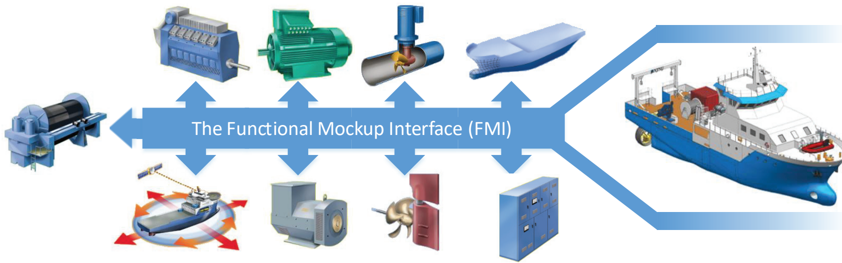


Figure 2: The vessel model depicted in the figure is an aggregate of several different sub-components. *Vessel sub-component figures courtesy of the Virtual Prototyping of Maritime Systems and Operations project (Research Council of Norway, grant nr. 225322).*

Table 1: Open source co-simulation master tools supporting FMI.

	FMI support										
	CS		*ME*								
Name	*v1.0*	*v2.0*	*v1.0*	*v2.0*	**SSP**	**Distributed**	**API**	**CLI**	**GUI**	**Version**	**License**
Coral	✓	✓				✓	✓	✓		0.10.0 (Dec. 2018)	MPLv2
DACCOSIM		✓				✓			✓	2.4.0 (Feb. 2020)	AGPL
FMI Go!	✓	✓	✓	✓	✓[a]	✓		✓		0.5.0 (Nov. 2019)	MIT
Maestro		✓				✓	✓[b]		✓	1.0.10 (Apr. 2020)	GPLv3
MasterSim	✓	✓					✓	✓	✓	0.8.2 (Dec. 2019)	LGPLv3
Ptolemy II	✓	✓	✓	✓			✓		✓	11.0.1 (Jun. 2018)	MIT
FMPy	✓	✓	✓	✓	✓[a]		✓	✓	✓	0.2.17 (Feb. 2020)	BSD
OMSimulator		✓		✓	✓		✓	✓	✓	2.0.1 (Jan. 2019)	GPLv3

[a] Draft version
[b] HTTP API

Functional Mock-up Units (FMUs), such as FMU-proxy, may be used within the context of SSP.

5. Implementation of a preliminary digital-twin model of the Gunnerus, together with subsequent simulation results comparing the power consumption of the model and its real-life counterpart.

The rest of the paper is organized as follows. Section 2 introduces some related work for co-simulation and digital twin platforms. An introduction to the employed co-simulation framework is given in Section 3. The following section provide some implementation notes on the work towards realizing a digital twin of the Gunnerus. Section 5 introduces the case-study, with results and discussions following in Section 6. Finally, some concluding remarks are provided in Section 7.

2 Related work

This section presents an overview of co-simulation technology and related tools, as well as a brief overview of digital twin platforms. Co-simulation as a technology was born out of the idea that 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. In a co-simulation, different sub-systems are modeled separately and composed into a global simulation where each model is being executed independently, sharing information only at discrete time-points. A comprehensive state-of-the-art survey on this topic is given in Gomes et al. (2018). Compared to more traditional monolithic simulations, co-simulation encourages re-usability, model sharing, and fusion of simulation domains. Thus it is in line with the OSP’s vision of establishing an eco-system for model sharing within the maritime industry.

Two noteworthy standards for co-simulation exist. The High Level Architecture (HLA) Dahmann et al. (1997) mainly for discrete event co-simulation and the Functional Mock-up Interface (FMI) Blochwitz et al. (2012) for continuous time co-simulation. This work primarily addresses the latter, due to the high number

of supporting tools and the ease with which models can be created and shared. Moreover, a recent survey showed that experts consider the FMI standard as the most promising standard for continuous time, discrete event and hybrid co-simulation Schweiger et al. (2019). Some efforts have also been devoted towards combining the two standards as demonstrated in Yilmaz et al. (2014); Falcone and Garro (2019).

The FMI, currently at version 2.x, is a tool-independent standard that aims to improve the exchange of simulation models between suppliers and original equipment manufacturers. The standard supports both model exchange (ME) and co-simulation (CS) of dynamic models. The key difference between these two variants is that CS models embed a solver, making it easier to deploy at the cost of flexibility. An FMU is a model which implements the FMI standard. It 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.

Since the inception of the FMI standard, a myriad of libraries and software tools have been created or adapted to support it. At the time of writing, the official FMI web page lists over 140 tools, which clearly shows that the standard is being adopted in force. Examples of FMI based co-simulation applied within the maritime domain can be found in Bulian and Cercos-Pita (2018); Hassani et al. (2016); Chu et al. (2018, 2019). Although this standard has reached acceptance in industry, it provides only limited support for simulating systems that mix continuous and discrete behavior, which are typical for CPS Cremona et al. (2018). A future version of the standard (FMI 3.0) will introduce clocks for synchronization of variables changes across FMUs, allowing co-simulation with events.

The Distributed co-simulation protocol (DCP) Krammer et al. (2018) is a standard for real-time and non-real-time system integration and simulation, which the Modelica Association has adopted as a Modelica Association Project. The DCP is compatible with FMI, and just like FMI, it leaves the design of the master out of scope from the specification.

FMU-proxy Hatledal et al. (2019a,b) is an open-source framework that enables language and platform independent access to FMUs. In short, FMU-proxy provides remote procedure call (RPC) mapping to the FMI 2.0 for co-simulation interface. This is achieved by wrapping one or more FMU in a server program supporting multiple schema-based and language-independent RPC systems over several network protocols. The use of schema-based RPCs allows users to easily auto-generate client/server code for a wide range of common programming languages. The framework is independent of the master algorithm, and can therefore be re-used in different software projects.

The System Structure and Parameterization (SSP) Köhler et al. (2016) is a tool-independent standard to define complete systems consisting of one or more components (such as FMUs) including their parameterization, which can be transferred between simulation tools. Version 1.0 of the standard was released in March 2019. The SSP standard is closely aligned with the FMI standard, using the same definition of units and variable types. While FMI is the only model format explicitly mentioned in the standard, a component, which is a blueprint for a model in this context, does not necessarily need to be an FMU. This allows other model formats to be referenced within a SSP archive, such as FMU-proxy or DCP.

Table. 1 provides an overview of open-source tools that are able to orchestrate and run systems of FMUs Gómez et al. (2019); Liu et al. (2001); Nicolai (2017); Lacoursière and Härdin (2017); Ochel et al. (2019); Thule et al. (2019); Catia-Systems (2019); Sadjina et al. (2019). This excludes low-level libraries like the FMI Library JModelica (2017), JavaFMI Galtier et al. (2017), and similar, which only handle loading of individual FMUs. Although a number of existing co-simulation master tools exist and usage of co-simulation to facilitate the digital twin has been presented in, e.g., Yun et al. (2017); Jung et al. (2018); Scheifele et al. (2019); Negri et al. (2019), the OSP partners decided to develop their own alternative, introduced in the following section, due to requirements related to the licensing model, performance, implementation language, maritime ontology, distributed model execution and support for key technologies like FMI 1.0 & 2.0, DCP, and SSP. None of the tools listed support DCP and only FMPy, FMIGo! and OMSimulator support SSP. However, the SSP draft version used by FMPy and FMIGo! is outdated and incompatible with the 1.0 version. Due to the inner workings of some of the models involved, not all models can co-exist within the same process. To overcome this, distributed model execution is required. Neither, FMPy nor OMSimulator supports this. In this way there is sufficient reasoning behind developing an alternate solution that among other things supports SSP 1.0, enables optional distributed execution of FMUs, and which plans to sup-

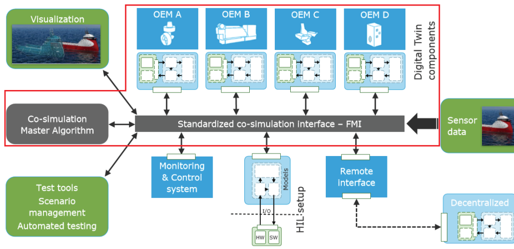


Figure 3: The OSP architecture, with the scope *libcosim* highlighted. *Figure courtesy of the Open Simulation platform.*

port the DCP standard in the future. However, a significant reason for developing yet another co-simulation platform is to maintain control over the software, which allows the collaborators to decide on issues regarding licensing, which features to support and so on.

3 Co-simulation environment

This section introduces the co-simulation environment employed for this research, termed the OSP. The OSP is a collection of software packages developed in collaboration by DNV-GL, SINTEF Ocean, Kongsberg Maritime, and NTNU to facilitate co-simulations and building of digital twin systems and vessels. One of the fundamental deliverables of the OSP is a software library for orchestrating and performing co-simulation named *libcosim*, which is described further below.

3.1 libcosim

libcosim is a cross-platform C/C++ library enabling co-simulations to be orchestrated and run. The scope of *libcosim* within the context of the overall vision of the OSP is illustrated by the framed region in Fig. 3. The OSP will create the foundation for an ecosystem where the maritime industry can perform co-simulation and share simulation models in an efficient and secure way to facilitate building of digital twin systems and vessels. *libcosim* is the cornerstone of the system, enabling users to easily integrate and combine their own components into a complete system, e.g. for the purpose of maritime industry design, operation, service, and maintenance. The co-simulation interface is based on the FMI, with both FMI 1.0 and 2.0 for CS being supported. ME models are not directly supported, however such models may be converted to CS using some appropriate stand-alone tool. Additionally, distributed execution of FMUs is supported through integration with FMU-proxy. Support for the DCP is also planned, which will enable hard real-time integration of hardware devices.

The *libcosim* is written in modern C++, making heavy use of features found in C++11 and above. In order to more easily support integration with other tools, a separate C library is maintained that provides access to most of the functions found in the C++ library. The library is loosely based on Coral with some elements added from CyberSea developed by DNV-GL. Both of whom were developed by collaborating partners. Compared to similar co-simulation libraries and frameworks, *libcosim* is mostly concerned with establishing a solid API that can be embedded in higher-level applications developed by end-users. For convenience, a CLI, which makes the software accessible to non-developers and that simplifies the realization of a number of use-cases, has been developed.

Some of the features of *libcosim* are:

- Integration with Conan dependency manager—making building and distributing the software easier.
- A separate C-API for easier integration with other applications.
- Support for both version 1.0 & 2.0 of the FMI standard for CS.
- Basic support for version 1.0 of the SSP standard, which allows complete simulation systems to be represented in a standardized way.
- Bulk read/write and caching of variable data for efficient access.
- FMU-proxy integration, enabling (optional) distributed execution of FMUs. This in turn enables models to be run regardless of platform, license and software dependencies.
- An extensible design, where master algorithms, slaves, observers, and manipulators are pluggable–allowing library users more control over the simulation.
- The ability to specify events, inline or through configurations files, to occur at specified trigger points, through so-called *scenarios*.

The design of *libcosim* is centralized, with all data flowing through the master. This makes for a less complicated, easier to maintain, easier to debug, and more flexible design compared to similar co-simulation engines such as Coral, where data flows directly between slaves. For instance, entities that want to observe or manipulate the simulation can do so directly as all data

Listing 1: Specifying FMU-proxy sources using libcosim & SSP. Components can be loaded from either an URL, the file system, or using the guide of an already-loaded FMU.

```
<ssd:Component name="model1" source="fmu-proxy://localhost:9090?file=Component.fmu">
<ssd:Component name="model2" source="fmu-proxy://localhost:9090?guid=85bb6608-13d0-46b8-9b8e">
<ssd:Component name="model3" source="fmu-proxy://localhost:9090?url=http://example.com/Component.fmu">
```

is obtainable from a single source. Pure distributed co-simulation masters such as Coral and FMI Go! dictate that all slaves are to be run distributed, whereas *libcosim* makes this entirely optional. Support for this is currently implemented through integration with FMU-proxy, which communicates with remote FMUs using Thrift over TCP/IP. Listing. 1 shows how FMU-proxy components are configured using SSP. A plug-in based system is used to resolve component URIs, allowing custom component sources like FMU-proxy to be added with ease. The support for SSP is not feature complete, but includes the ability to apply linear transformations to connections and multiple parameter sets, both defined inline and as external files.

A crucial part of any co-simulation tool is the available master algorithms. Currently, the library only ships with a single algorithm. A fixed-step algorithm that supports individual FMUs to run at separate step-sizes. However, the API facilitates the creation of additional master algorithms, and as time passes, hopefully more algorithms will be added.

C++ can be a challenging language to learn. Especially compared to higher-level languages like Python or Java. For instance Java has fewer features to learn, is garbage-collected and comes with a richer standard library. Additionally, the tooling, in the form of integrated development environments (IDEs), build systems, and package managers, is state-of-the-art. Therefore, and in order to aid developers that would rater develop in Java, NTNU has developed *cosim4j*, a Java wrapper for *libcosim* introduced in more detail below.

3.2 cosim4j

cosim4j is a Java wrapper for *libcosim*. The goal of the Java API is to be generally easier to use and provide more high-level features than its native counterpart. It uses the Java Native Interface to efficiently interact with the native library. To make the library accessible, it is made available as a Maven artifact at https://bintray.com/open-simulation-platform/maven/cosim4j. Furthermore, the artifact include pre-built native binaries for Linux and Windows, which means that no prior installation of *libcosim* is required.

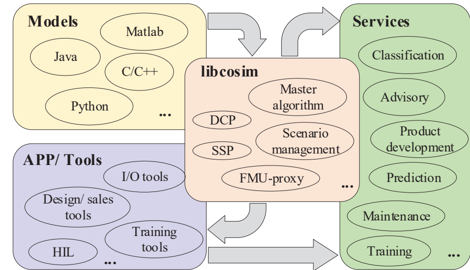


Figure 4: Proposed structure for digital twin implementation.

4 Implementation

Inspired by the overall vision of the OSP in Fig.3, a plausible digital twin framework based on *libcosim* is proposed, as shown in Fig. 4. In the following, implementation details for the proposed framework are provided.

The workflow used towards realizing a digital twin model of the Gunnerus, or digital twins in general, is as follows.

1. Establish the purpose of the model. What should it communicate?
2. Any existing models related to the vessel are collected.
3. Any missing pieces of the puzzle are mapped and consequently implemented using the appropriate software, e.g. FMI4j, PythonFMU or some domain-specific tool, and exported as FMUs.
4. Define the structure of the simulation using the standardized SSP format.
5. Run the simulation using an appropriate tool.

Currently, *cosim4j* is used to run the simulation. The configuration of the system to be simulated is done using SSP. Much as FMI allows us to decouple from the modeling tools, SSP allows us to decouple from the co-simulation master. However, as some of the models currently in use by the digital twin dictate that the full simulation may not run within a single process, the SSP

implementation should support components that can run in separate processes such as FMU-proxy or DCP. An alternative approach to solve this issue, is to run all models in a distributed fashion like e.g. FMIGo! does. However, selective distributed execution, as found in the proposed implementation, has some benefits like easier debugging and less communication overhead in the general case.

As it stands, a complete public overview of available tools that supports SSP is lacking, and implementations, as they become available, will most likely only support FMUs loaded from the file-system—the basic requirement of such an implementation. Using SSP, the structure of a simulation is defined in an XML configuration file. At least one configuration file named *SystemStructure.ssd* must be present. However, additional configurations may optionally be defined, allowing a single SSP archive to contain multiple simulation configurations. Simply explained, an *.ssd* defines which models make up a simulation (components), which variables are exposed (connectors), how they are connected (connections), and how they are parameterized (parameter-sets). Annotations are used to define tool-specific features. The *.ssd* files are packed, together with any resources required, like FMUs, in a zip archive with an *.ssp* extension. While SSP makes it easier to configure systems that can be simulated in a standardized way, it may still be challenging to manually create valid SSP archives due to the shear amount of XML that might have to be written and the packaging of files that goes into the archive. To ease this process, NTNU has developed *SSPgen* Hatledal (2020)—a domain specific language for generating self validating SSP archives. Aside from getting the SSP archive validated prior to simulation, *SSPgen* drastically reduces the amount of code required.

Also embedded in the workflow for realizing the digitalization of the Gunnerus is the use of a set of in-house developed open-source tools for creating FMI 2.0-compatible models in Java (FMI4j) and Python (PythonFMU). Their ease of use makes them ideal for rapid prototyping. FMI4j is an open-source cross-platform Java framework for importing and exporting FMUs. Initially created with FMI import in mind, it has been updated to enable Java code to be exported as FMUs in order to support the work addressed in this paper. Compared to the similar JavaFMI package, FMI4j relies on the Java Native Interface rather than a message passing system making it significantly faster. A Gradle plugin and an easy-to-use CLI that exports conforming Java code as cross-platform FMUs is provided. Listing. 2 shows the minimal required code to write FMI 2.0 compatible models in Java using FMI4j. PythonFMU Hatledal et al. (2020) is a lightweight, open-source, and cross-platform Python 3.x framework for building FMUs readily available through the pip package manager. It has been specifically designed to enable data scientists in the team to contribute with models as the work progresses. Listing. 3 shows the minimal required code to write FMI 2.0 compatible models in Python using PythonFMU.

Listing 2: Writing slaves in Java using FMI4j.

```
public class JavaSlave extends Fmi2Slave {

    @ScalarVariable(causality=output)
    private double realOut;

    public JavaSlave(Map<String, Object> args) {
        super(args);
    }

    @Override
    public void doStep(double t, double dt) {
        realOut = ...
    }
}
```

Listing 3: Writing slaves in Python using PythonFMU.

```
class PythonSlave(Fmi2Slave):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.realOut = 0.0
        self.register_variable(Real("realOut",
            causality=output))

    def do_step(self, t, dt):
        self.realOut = ...
        return True
```

When designing co-simulations, there might be issues related to causality when two models, which theoretically would be a good match for coupling, have inputs and outputs flowing in the wrong direction compared to each other. Thus, declaring causalities when writing models like shown in Listing. 2 and 3 must be done with great care and in collaboration with other model developers.

5 Case study

Here, the configuration of a case study utilizing the Gunnerus is presented. Its purpose is to test possible applications of digital twin for ship maneuvering and on-board decision support.

The Gunnerus, as seen in Fig. 5, is equipped with the latest technology for a variety of research activities within biology, technology, geology, archaeology, oceanography, and fisheries research. In addition to research, the ship is used for educational purposes and is an important platform for marine courses at all levels and disciplines. Some main dimensions of the vessel are given in Table 2.

Figure 5: Starboard view of the R/V Gunnerus.

Table 2: Main dimensions of the Gunnerus.

Parameter	Value
Length overall (Loa)	36.25 m
Length between pp (Lpp)	33.90 m
Waterline length (Lwl)	29.90 m
Breadth middle (Bm)	9.60 m
Breadth extreme (B)	9.90 m
Depth mld. Main deck (Dm)	4.20 m
Draught, mld (dm)	2.70 m
Deadweight	165 t

In this preliminary work, pre-recorded data from the Gunnerus in the form of comma-separated values files are used, as neither the infrastructure for establishing a live link to the vessel nor the means to access recorded data from the cloud are ready. The pre-recorded data from the Gunnerus is wrapped in an FMU, hiding this particular implementation detail and making it possible to add a cloud-connected drop-in-replacement in the future. For this, the plan is to leverage the Cognite[2] cloud platform for data cleaning, analytics, and contextualisation.

One of the deliverables of the OSP is a set of free-of-charge reference models, including models of the most common marine systems and ship dynamics components. The case-study makes use of a number of these models to realize the digital twin. The point of this case study is not to go into detail about how these models are implemented, which in general are black-boxes that could hide proprietary information. Rather, the point is to showcase how co-simulation technology, open-source software, open standards and a library of readily available marine models can be used to develop a digital twin scenario.

The following list briefly describes each of the FMUs used to create the digital Gunnerus.

[2] https://www.cognite.com/

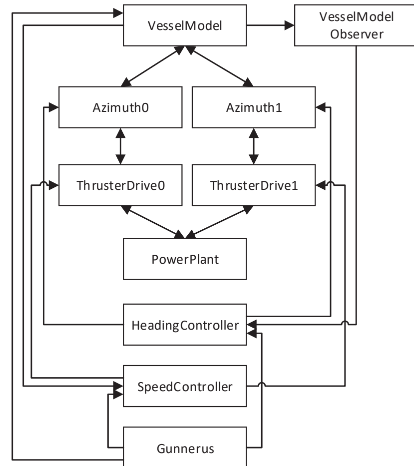


Figure 6: Diagram showing the logical relationship of the involved components.

1. **Gunnerus -** This model contains previously recorded sensor data measured during operation of the Gunnerus. The time-series data is sampled at 1Hz and includes information such as:

    - Heading angle and percent-wise commanded RPM of the tunnel-thruster in the bow as well as the two azimuth thrusters in the aft.
    - Longitude and latitude.
    - Surge, sway, and heave.
    - Yaw, pitch, and roll.
    - Wind direction and speed.
    - Positional and rotational velocities.

    The FMU implements linear interpolation of the recorded data, which is convenient given the low sample rate of the sensor data relative to the simulation, which runs at 20 Hz. In this work, the model acts as a stand-in for what eventually should become a stream of data originating from the real asset.

2. **VesselModel -** This model computes the vessel hydrodynamics such as the radiation forces, mass, and restoring forces as well as manoeuvring forces (resistance and cross flow drag as well as semi-empirical corrections). The equations of motions are solved by this model, summing up all the external forces acting on the vessel. SINTEF Ocean originally implemented the *VesselModel* to model

Table 3: FMUs involved in the case study.

Component	Tool	Vendor	FMI version	canBeInstantiated-OnlyOncePerProcess
HeadingController	FMI4j	NTNU	2.0	False
SpeedController	FMI4j	NTNU	2.0	False
Gunnerus	FMI4j	NTNU	2.0	False
VesselModelObserver	FMI4j	NTNU	2.0	False
ThrusterDrive[a]	20sim	SINTEF Ocean	1.0	True
PowerPlant[a]	20sim	SINTEF Ocean	1.0	True
VesselModel[a]	VeSim	SINTEF Ocean	1.0	True[b]
PMAzimuth[a]	VeSim	Kongsberg Maritime	1.0	True[b]

[a] OSP reference model.
[b] Additionally, only one instance of any model generated by this tool may be instantiated within the same process.

the Gunnerus as part of the SimVal Hassani et al. (2015) project. It was later updated to better approximate an elongated Gunnerus vessel as part of the MAROFF KPN: Digital Twins for Vessel Life Cycle Service (TwinShip). While the model was validated during the SimVal project, it has yet to be validated against the elongated version of the vessel.

3. **VesselModelObserver -** A simple model that computes the direction of travel and speed over ground of the *VesselModel* based on its current and previous position.

4. **SpeedController -** A general-purpose proportional-integral-derivative (PID) controller. It is used to regulate the force required by the *ThrusterDrives* so that the speed of the *VesselModel* and the *Gunnerus* are aligned.

5. **HeadingController -** A special-purpose PID controller where the input data used to compute the controller error is treated as angles in the range $[-180^{\circ}, 180^{\circ}]$. This unwinds any input angles that lie outside of the specified range.

6. **PMAzimuth -** The hydrodynamic model of the azimuth thrusters without actuator/motor, implemented by Kongsberg Maritime using VeSim as part of the ViProMa project. Given a certain RPM command (issued by the *ThrusterDrive* FMU), location on the hull, azimuth angle, vessel speed, and the loss factor, the model will output the 3DOF (surge, sway, heave) force generated.

7. **ThrusterDrive -** A drive that converts force commands from the *SpeedController* into RPMs for the *PMAzimuth*.

8. **PowerPlant -** A marine power plant with two equally large gensets, including auxiliary load and circuit breakers.

Fig. 6 shows the logical relationship of the different FMUs, with additional information about the FMUs being provided in Table 3. As illustrated by Fig. 7, the system is far from trivial with a total of 48 variable connections between the models involved. Note that, instances of the *ThrusterDrive* and *PowerPlant* models generated by 20Sim, using an early version of their FMI exporter, cannot co-exist within the same process. This is also true for VeSim Fathi (2013) generated FMUs like the *VesselModel* and *PMAzimuth*. Moreover, these models cannot co-exist within the same process as any other models generated by this tool due to shared library symbol conflicts. To overcome this challenge, execution of the various model instances that cannot co-exist within the same process are split across multiple processes. This is easily solvable using FMU-proxy. Running two instances of FMU-proxy provides two additional processes, which is sufficient for this scenario to run. The distribution of FMUs across the available processes can be seen in Table 4.

To realize the simulation, *cosim4j* is used. The case study presented in this paper is challenging to execute due to the fact that some of the FMUs cannot co-exist within the same process. This makes it impossible to run in non-distributed co-simulation software. This

Table 4: Distribution of FMUs across processes.

Process	FMUs
libcosim	PowerPlant, Gunnerus, VesselModel, VesselModelObserver, SppedController, HeadingController
fmu-proxy1	PMAzimuth, ThrusterDrive
fmu-proxy2	PMAzimuth, ThrusterDrive

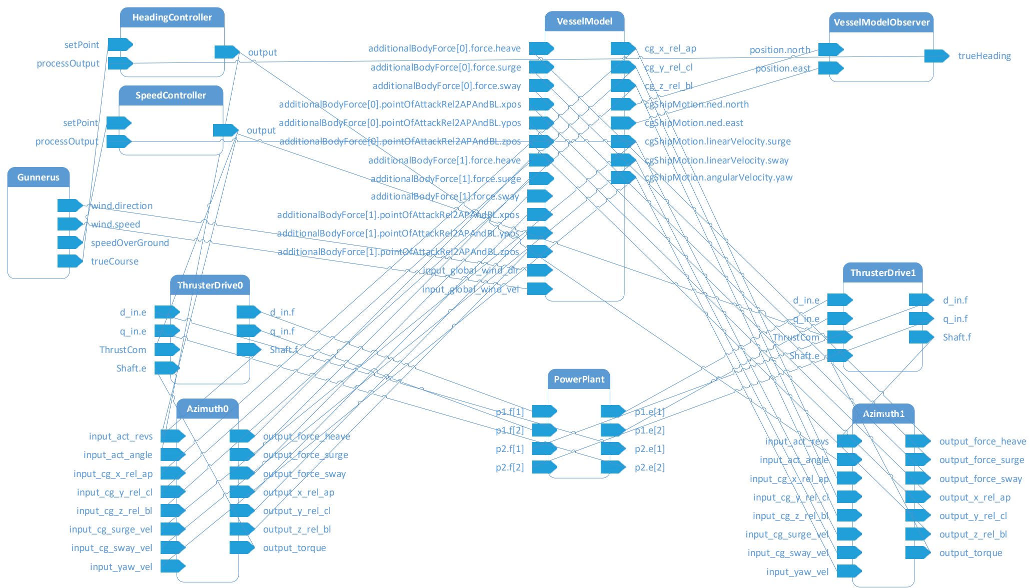


Figure 7: FMU connection graph.

challenge can be solved by means of FMU-proxy, which allows FMUs to be selectively chosen for distributed execution. This enables other parts of the simulation to run within the main application process, which provides the benefit of faster execution times and easier debugging. As noted, when taking the requirements for distributed model execution and the use of SSP 1.0 for creating a standardized system representation, *libcosim* is currently the only library that is applicable of the libraries presented in Section 2.

The idea of the case study is to compare the power consumption of the real vessel and the preliminary digital twin, using data collected from the Gunnerus while performing experiments in the open sea near the city of Trondheim. This is done by feeding the speed and true heading of the real vessel into a set of controllers used to regulate the motion of the twin. To simplify the case study, the equipped thunnel-thruster is not utilized and the command signals to both azimuth thrusters in the aft are equal. Ideally, the power consumption should be comparable, which would indicate a good model fit. However, environmental effects such as current, which are very difficult to measure, could introduce discrepancies between the real and simulated vessel. Yet, the Gunnerus is able to measure and record both wind direction and speed, which are being fed into the model. These measurements are illustrated in Fig. 8. The case study can be run both with and without 3D visualisation enabled. When it is enabled, the simulation is interactive and can be paused/resumed and real-time simulation can be toggled on/off. With real-time on, the execution will try to run in real-time. When successful, the real-time-index (RTI) of the simulation will stay close to 1.0. The other option is to run the simulation as fast as possible. In this preliminary work, it is not necessary to run in real-time as pre-recorded data is used. The case study runs with an RTI of about 30 using a 7th generation Intel i7-8700 CPU on Windows 10. This means that the current models should not be a potential bottleneck once online data becomes available.

6 Results and Discussion

In the following, the simulation results from the case-study are shown. The simulation lasts for approx. 33 minutes, in which the Gunnerus is performing maneuvers in the open sea outside the city of Trondheim. Fig. 9 shows the position and heading of the real and simulation vessel during the case study. Furthermore, the wind direction and normalized magnitude are also shown. To see the actual magnitude of the measured wind speeds, refer to Fig. 8. A comparison of the course of the two vessels is shown in Fig. 10. As can be seen, they are aligning quite well during the entire simula-

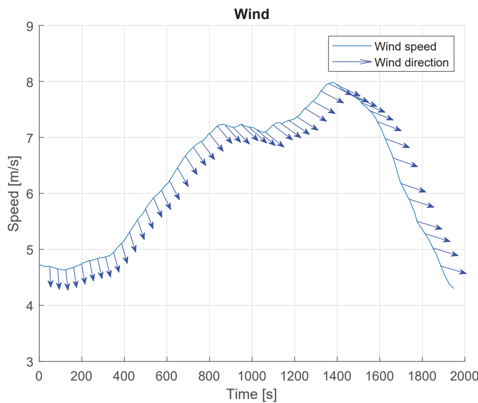


Figure 8: Wind speed and direction measurements obtained from the Gunnerus. The arrow indicates the wind direction according to north.

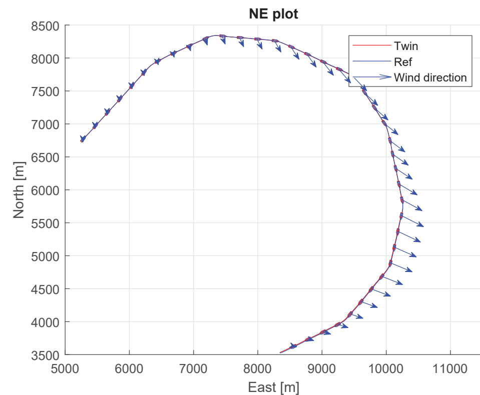


Figure 9: North-east plot showing the trajectory and heading of the vessels during the experiment. The blue arrow indicates the wind direction according to north and normalized magnitude of the speed.

tion. However, the heading controller is a bit aggressive, leading to some oscillations around the set-point, which the authors have not been able to eliminate without sacrificing accuracy over time. The effect of this would be that the power consumption of the twin vessel is increased to some extent. Furthermore, a comparison of the surge speed is shown in Fig. 11. As seen in Fig. 12, speed transients for the twin relates to changes in course made by the Gunnerus. The power consumption is shown in Fig. 13. Interestingly, the power consumption calculated from the twin is showing higher correlation with the speed than that of the real vessel. After approximately 1100s, the power measurement for the real vessel is actually reduced as the speed increases. This could indicate that the vessel is affected by external forces that the model is not aware of, such as current. Therefore, this discrepancy does not necessarily indicate a weakness in the model, but actually provides potentially valuable information regarding external environmental forces acting on the real hull.

From these results, it is clear that some of the underlying models could be more accurately tuned to better reflect the current vessel design. As noted, the employed hull model used has not been thoroughly validated after the Gunnerus underwent an elongation. Doing so might improve the observed difference in terms of overall power consumption.

In order to improve the usability of the digital twin, the offline approach of using pre-recorded operational data should be discarded in favour of a cloud-connected solution with live access to the real asset. This will enable stakeholders and crew members to benefit from the insights provided by the model. This is perhaps the most challenging part, as it requires significant upgrading of the vessel's infrastructure. Today, data from the vessel is buffered on board and transmitted in bulk at intervals measured in minutes. One pragmatic solution to this could be to run the model on board. Implementation-wise, going from pre-recorded data to live data is only a matter of performing a drop-in replacement of the FMU that emits sensor data from the vessel. The simulation structure would not have to be updated as the replacement would share the same model interface. However, using a model connected to a real asset would imply that the simulation would have to be performed in real-time. This mode is supported by *libcosim* and the models used in the simulation are all lightweight enough for the simulation to achieve real-time execution speeds.

7 Conclusion

This paper presents the preliminary results, procedure, and enabling technologies related to our ongoing work to establish a fully operational digital twin of R/V Gunnerus. Co-simulation allows the CPS that the vessel represents to be simulated using models from different vendors and tools. This is absolutely crucial for an aggregate model in the maritime domain, as many different vendors and domain-specific tools are usually involved. Not only does the use of co-simulation allow building of aggregate systems from different vendors, it also allows the simulation to be performed in freely available open-source tools. Furthermore, it makes it possible to decorate the system with models implemented in the tools that best fit the objective.

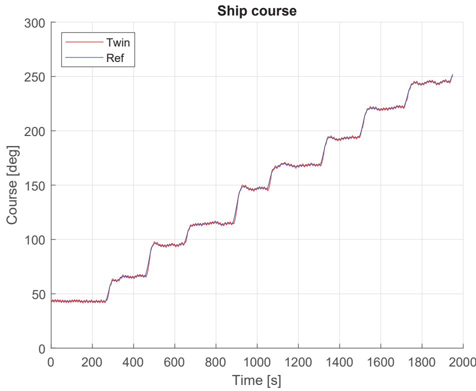


Figure 10: Course of the vessels. Revisit Fig. 9 for an alternative representation.

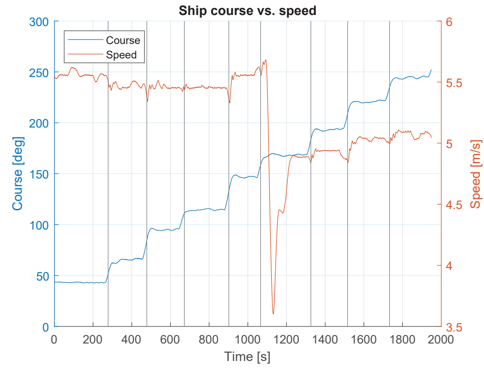


Figure 12: Twin surge speed with respect to course changes by the Gunnerus.

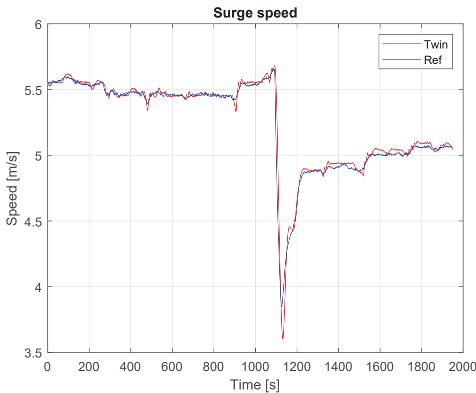


Figure 11: Speed of the vessels.

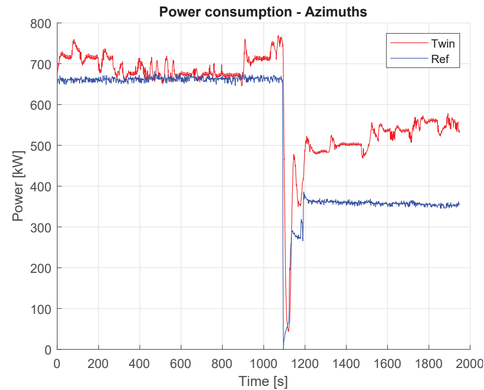


Figure 13: Power consumption comparison. The power output shown is the sum of the two azimuths.

Using the presented simulation framework, model library and tools presented in this paper, NTNU will continue its work towards realizing a digital twin of the Gunnerus, gradually improving its accuracy. Continued development of use-cases will provide meaningful on-board decision support for the crew on-board the Gunnerus. A plausible next step would be to expand on the presented case study by applying a force correction to the hull model in order to offset any differences in position and/or yaw. The amount of force required for this correction could be used as an estimation of environmental forces being applied to the real hull. Being able to quantity these forces would provide substantial support for the crew.

Acknowledgement

This work was supported in part by the Project "Digital Twins for Vessel Life Cycle Service", under Grant 280703 from Research Council of Norway, and in part by the Project "SFI Offshore Mechatronics", under Grant 237896 from Research Council of Norway.

The authors would like to thank the members of the Open Simulation Platform for their work related to the employed co-simulation library. A special thanks goes to Stian Skjong and Martin Rindary at SINTEF Ocean for their contribution in terms of models and invaluable insights regarding their usage.

References

Bekker, A. Exploring the blue skies potential of digital twin technology for a polar supply and research vessel. In *Proceedings of the 13th International Marine Design Conference Marine Design XIII (IMDC 2018)*, volume 1. pages 135–146, 2018. doi:10.1201/9780429440533-11.

Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., 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*, 076. Linköping University Electronic Press, pages 173–184, 2012. doi:10.3384/ecp12076173.

Bulian, G. and Cercos-Pita, J. L. Co-simulation of ship motions and sloshing in tanks. *Ocean Engineering*, 2018. 152:353–376. doi:10.1016/j.oceaneng.2018.01.028.

Catia-Systems. Fmpy. 2019. URL `https://github.com/CATIA-Systems/FMPy`. (Date accessed 10-December-2020).

Chu, Y., Hatledal, L. I., Æsøy, V., Ehlers, S., and Zhang, H. 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*, 2018. 140(2). doi:10.1115/1.4038346.

Chu, Y., Pedersen, B. S., and Zhang, H. Virtual prototyping for maritime winch design and operations based on functional mock-up interface co-simulation. *Ships and Offshore Structures*, 2019. 14(sup1):261–269. doi:10.1080/17445302.2019.1577597.

Cremona, F., Lee, E., Lohstroh, M., Masin, M., Broman, D., and Tripakis, S. Hybrid co-simulation: It’s about time. In *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, 14 October 2018 through 19 October 2018*. Association for Computing Machinery, Inc, 2018. doi:10.1145/3239372.3242896.

Dahmann, J. S., Fujimoto, R. M., and Weatherly, R. M. The department of defense high level architecture. In *Proceedings of the 29th conference on Winter simulation*. pages 142–149, 1997. doi:10.1145/268437.268465.

Durling, E., Palmkvist, E., and Henningsson, M. 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*, 132. Linköping University Electronic Press, pages 329–335, 2017. doi:10.3384/ecp17132329.

Falcone, A. and Garro, A. Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface. *Simulation Modelling Practice and Theory*, 2019. 97:101967. doi:10.1016/j.simpat.2019.101967.

Fathi, D. Marintek vessel simulator (vesim), user manual. *MARINTEK. Report*, 2013.

Galtier, V., Ianotto, M., Caujolle, M., Tavella, J.-P., Gómez, J. É., Cabrera, J. J. H., Reinbold, V., and Kremers, E. Experimenting with matryoshka co-simulation: Building parallel and hierarchical fmus. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*. pages 663–671, 2017. doi:10.3384/ecp17132663.

Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. Co-simulation: a survey. *ACM Computing Surveys (CSUR)*, 2018. 51(3):1–33. doi:10.1145/3179993.

Gómez, J. É., Cabrera, J. J. H., Tavella, J.-P., Vialle, S., Kremers, E., and Frayssinet, L. Daccosim ng: co-simulation made simpler and faster. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, 157. Linköping University Electronic Press, 2019. doi:10.3384/ecp19157785.

Hassani, V., Rindarøy, M., Kyllingstad, L. T., Nielsen, J. B., Sadjina, S. S., Skjong, S., Fathi, D., Johnsen, T., Æsøy, V., and Pedersen, E. Virtual prototyping of maritime systems and operations. In *ASME 2016 35th International Conference on Ocean, Offshore and Arctic Engineering*. American Society of Mechanical Engineers, pages V007T06A018–V007T06A018, 2016. doi:10.1007/s00773-017-0514-2.

Hassani, V., Ross, A., Selvik, Ø., Fathi, D., Sprenger, F., and Berg, T. E. Time domain simulation model for research vessel gunnerus. In *ASME 2015 34th International Conference on Ocean, Offshore and Arctic Engineering*. American Society of Mechanical Engineers Digital Collection, 2015. doi:10.1115/OMAE2015-41786.

Hatledal, L. I. sspgen. 2020. URL `https://github.com/NTNU-IHB/sspgen`. (Date accessed 10-December-2020).

Hatledal, L. I., Collonval, F., and Zhang, H. Enabling python driven co-simulation models with pythonfmu. In *ECMS*. pages 235–239, 2020. doi:10.7148/2020-0235.

Hatledal, L. I., Styve, A., Hovland, G., and Zhang, H. A language and platform independent co-simulation framework based on the functional mock-up interface. *IEEE Access*, 2019a. 7:109328–109339. doi:10.1109/ACCESS.2019.2933275.

Hatledal, L. I., Zhang, H., Styve, A., and Hovland, G. 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*, 157. Linköping University Electronic Press, 2019b. doi:10.3384/ecp1915779.

JModelica. Fmi library. 2017. URL `http://www.jmodelica.org/FMILibrary`. (Date accessed 10-December-2020).

Jung, T., Shah, P., and Weyrich, M. Dynamic co-simulation of internet-of-things-components using a multi-agent-system. *Procedia CIRP*, 2018. 72:874–879. doi:10.1016/j.procir.2018.03.084.

Köhler, J., Heinkel, H.-M., Mai, P., Krasser, J., Deppe, M., and Nagasawa, M. Modelica-association-project system structure and parameterization–early insights. In *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, 124. Linköping University Electronic Press, pages 35–42, 2016. doi:10.3384/ecp1612435.

Krammer, M., Benedikt, M., Blochwitz, T., Alekeish, K., Amringer, N., Kater, C., Materne, S., Ruvalcaba, R., Schuch, K., Zehetner, J., 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, page 1, 2018. doi:10.22360/summersim.2018.scsc.001.

Lacoursière, C. and Härdin, T. 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*, 132. Linköping University Electronic Press, pages 653–662, 2017. doi:10.3384/ecp17132653.

Liu, H., Liu, X., and Lee, E. A. Modeling distributed hybrid systems in ptolemy ii. In *Proceedings of the 2001 American Control Conference.(Cat. No. 01CH37148)*, volume 6. IEEE, pages 4984–4985, 2001. doi:10.1109/ACC.2001.945773.

Negri, E., Fumagalli, L., Cimino, C., and Macchi, M. Fmu-supported simulation for cps digital twin. In *International Conference on Changeable, Agile, Reconfigurable and Virtual Production*, volume 28. pages 201–206, 2019. doi:10.1016/j.promfg.2018.12.033.

Nicolai, A. Mastersim - a simulation master for functional mockup units. 2017. URL `https://bauklimatik-dresden.de/mastersim/index.php?aLa=en`. (Date accessed 10-December-2020).

Ochel, L., Braun, R., Thiele, B., Asghar, A., Buffoni, L., Eek, M., Fritzson, P., Fritzson, D., Horkeby, S., Hällquist, R., et al. Omsimulator–integrated fmi and tlm-based co-simulation with composite model editing and ssp. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, 157. Linköping University Electronic Press, 2019. doi:10.3384/ecp1915769.

Open Simulation Platform. Open simulation platform - joint industry project for the maritime industry. 2020. URL `https://opensimulationplatform.com/`. (Date accessed 10-December-2020).

Perabo, F., Park, D., Zadeh, M. K., Smogeli, Ø., and Jamt, L. Digital twin modelling of ship power and propulsion systems: Application of the open simulation platform (osp). In *2020 IEEE 29th International Symposium on Industrial Electronics (ISIE)*. IEEE, pages 1265–1270, 2020. doi:10.1109/ISIE45063.2020.9152218.

Rasheed, A., San, O., and Kvamsdal, T. Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access*, 2020. 8:21980–22012. doi:10.1109/ACCESS.2020.2970143.

Sadjina, S., Kyllingstad, L. T., Rindarøy, M., Skjong, S., Æsøy, V., and Pedersen, E. Distributed co-simulation of maritime systems and operations. *Journal of Offshore Mechanics and Arctic Engineering*, 2019. 141(1). doi:10.1115/1.4040473.

Sanchez-Gonzalez, P.-L., Díaz-Gutiérrez, D., Leo, T. J., and Núñez-Rivas, L. R. Toward digitalization of maritime transport? *Sensors*, 2019. 19(4):926. doi:10.3390/s19040926.

Scheifele, C., Verl, A., and Riedel, O. Real-time co-simulation for the virtual commissioning of production systems. *Procedia CIRP*, 2019. 79:397–402. doi:10.1016/j.procir.2019.02.104.

Schleich, B., Anwer, N., Mathieu, L., and Wartzack, S. Shaping the digital twin for design and production engineering. *CIRP Annals - Manufacturing Technology*, 2017. 66(1):141–144. doi:10.1016/J.CIRP.2017.04.040.

Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schoeggl, J., Posch, A., and Nouidui, T. An empirical survey on co-simulation: Promising standards, challenges and research needs. *Simulation modelling practice and theory*, 2019. 95:148–163. doi:10.1016/j.simpat.2019.05.001.

Sullivan, B. P., Desai, S., Sole, J., Rossi, M., Ramundo, L., and Terzi, S. Maritime 4.0–opportunities in digitalization and advanced manufacturing for vessel development. *Procedia Manufacturing*, 2020. 42:246–253. doi:10.1016/j.promfg.2020.02.078.

Thule, C., Lausdahl, K., Gomes, C., Meisl, G., and Larsen, P. G. Maestro: The into-cps co-simulation framework. *Simulation Modelling Practice and Theory*, 2019. 92:45–61. doi:10.1016/j.simpat.2018.12.005.

Yilmaz, F., Durak, U., Taylan, K., and Oğuztüzün, H. Adapting functional mockup units for hla-compliant distributed simulation. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, 096. Linköping University Electronic Press, pages 247–257, 2014. doi:10.3384/ecp14096247.

Yun, S., Park, J.-H., and Kim, W.-T. Data-centric middleware based digital twin platform for dependable cyber-physical systems. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, pages 922–926, 2017. doi:10.1109/ICUFN.2017.7993933.

CC BY

F

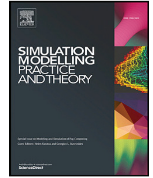
Paper A6

Simulation Modelling Practice and Theory 108 (2021) 102243

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



Vico: An entity-component-system based co-simulation framework

Lars I. Hatledal [a,*], Yingguang Chu [b], Arne Styve [c], Houxiang Zhang [a]

[a] Department of Ocean Operations and Civil Engineering, Norwegian University of Science and Technology, Larsgårdsvegen 2, 6009 Ålesund, Norway

[b] SINTEF Ålesund, Borgundvegen 340, 6009 Ålesund, Norway

[c] Department of ICT and Natural Sciences, Norwegian University of Science and Technology, Larsgårdsvegen 2, 6009 Ålesund, Norway

ARTICLE INFO

Keywords:
Co-simulation
Continuous-time simulation
Entity-component-system
Functional mock-up interface
System structure and parameterisation

ABSTRACT

This paper introduces a novel co-simulation framework running on the Java Virtual Machine built on a software architecture known as the Entity-Component-System. Popularised by games, this architecture favours composition over inheritance, allowing for greater flexibility. Rather than using a fixed inheritance tree, an entity is defined by its traits, which can be seamlessly changed during simulation. The framework supports the Functional Mock-up Interface standard for co-simulation, as well as the System Structure and Parameterisation standard for defining the system structure. Furthermore, the employed architecture allows users to seamlessly integrate physics engines, plotting, 3D visualisation, co-simulation masters and other types of systems into the framework in a modular way. To show its effectiveness, this paper compares the framework to four similar open-source co-simulation frameworks by simulating a quarter-truck system defined using the System Structure and Parameterisation standard.

1. Introduction

This paper introduces *Vico*, a novel high-level co-simulation framework, which is founded on a software architecture based on the Entity-Component-System (ECS) architecture [1–4]. The ECS, and variations of it, has roots from the gaming world [5] and follows the composition over inheritance principle, which allows for greater flexibility in terms of defining simulation objects than traditional alternatives afford. Rather than having objects inheriting data and functionality from a parent object (object-oriented programming), the object (entity) is composed of data (components). Every entity consists of one or more components which contains data. Therefore, the behaviour of an entity can be changed during run-time by systems that add, remove, or mutate components. This eliminates the ambiguity problems of deep and wide inheritance hierarchies that are difficult to understand, maintain and/or extend. In an inheritance-based architecture, for example, an instance of class Breakable will always be of type Breakable, while within an ECS the Breakable component in an entity can be removed or replaced with other components, seamlessly changing the entity's characterisation. The ECS architecture should not be confused with the entity-component (EC) architecture employed by mainstream game engines like Unreal Engine and Unity3D. While similar, the EC architecture does not split behaviour and data between systems and components. Rather, the component takes the role of both. In the employed ECS architecture, illustrated by Fig. 1, every object taking part in the simulation is known as an *entity*. An entity is basically just a container for *components*. A component is just state, with no behaviour. Behaviour is added to the simulation through *systems* that acts on entities within a certain *family*. A family is a set of entities with a certain set of components attached. These systems are responsible for acting upon and/or mutating the state of these components, which then drives the simulation forward. Entities, components, and systems

* Corresponding author.
E-mail address: laht@ntnu.no (L.I. Hatledal).

https://doi.org/10.1016/j.simpat.2020.102243
Received 16 September 2020; Received in revised form 23 November 2020; Accepted 5 December 2020
Available online 23 December 2020
1569-190X/© 2020 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

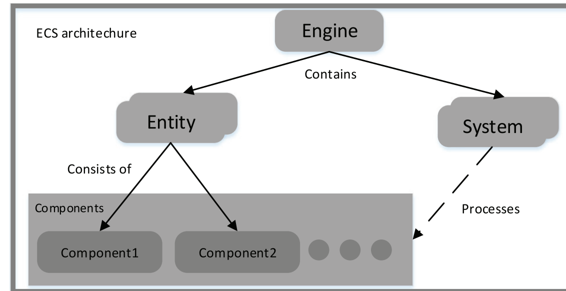


Fig. 1. High level overview of the ECS architecture.

may be added or removed from the engine at any time; thus family relationships, what an entity represents, and which entities a given system acts on are all highly dynamic. Achieving flexibility in terms of how objects in a simulation behaves and what they represent has always been a key driver for Vico, which was originally developed to support research activities related to virtual prototyping at NTNU Ålesund. Being able to change the fidelity of a running simulation is beneficial here, for example to intuitively enable the transformation of a virtual prototype purposed for a real-time training scenario into a more accurate engineering oriented simulation. In order to accommodate changing the fidelity of a running simulation like this, it is necessary to retain state. The ECS architecture solves this in a natural way by logically keeping state and behaviour separate. While the related EC architecture allows flexibility in terms of what an object represents, through adding/removing components just like with ECS, it does not accommodate state preservation.

Vico focuses on co-simulation and naturally supports the Functional Mock-up Interface (FMI) standard [6], which aims to improve the exchange of simulation models between suppliers and original equipment manufacturers. Currently at version 2.x, the FMI is a tool-independent standard that supports both model exchange (ME) and co-simulation (CS) of dynamic models. The key difference between these two variants is that CS models embed a solver, making them easier to deploy at the cost of flexibility. A model implementing the standard is called a 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.

Since the introduction of the FMI standard, a number of libraries and software tools have been created or adapted to support it. At the time of this writing, the official FMI web page lists over 140 tools, which clearly shows that the standard has been well received. A recent survey showed that experts consider the FMI standard to be the most promising standards for continuous time, discrete event, and hybrid co-simulation [7]. Vico supports both version 1.0 & 2.0 of the FMI for CS. ME models are not directly supported and should be converted to a CS model a priori in some appropriate tool. Distributed execution is possible using FMU-proxy [8], which makes it possible to run otherwise incompatible FMUs due to limitations in the FMU or incompatible system requirements. The System Structure and Parameterisation (SSP) [9] standard is also supported, which enables a tool-independent way of defining complete systems consisting of one or more components (such as FMUs), including their parameterisation.

Vico has in various forms been developed internally at the Intelligent Systems Lab with NTNU Ålesund for several years, serving as a test bed for testing software architectures to support simulation & visualisation of cyber-physical systems, virtual prototyping, and digital twin systems [10,11]. The current focal point is to act as an enabling technology for the MAROFF KPN Project *Digital Twins for Vessel Life Cycle Service (TwinShip)*,[1] with the purpose of developing digital twins of maritime systems and operations, which allows for not only configuration of systems and verification of operational performance, but also the provision of early warning, life cycle service support, and system behaviour prediction. As illustrated in Fig. 2, the use of co-simulation together with data-related optimisation, like data purification, and machine learning methods will be seamlessly combined from the design phase to the maintenance phase to achieve heterogeneous simulation, data analytics and behavioural prediction of maritime systems.

The rest of the paper is organised as follows. Firstly, some related work is presented in Section 2, followed by a description of the software architecture in Section 3. Case-studies are presented in Section 4, and some concluding remarks and future works appear in Section 5.

[1] https://org.ntnu.no/intelligentsystemslab/project/twinship.html.

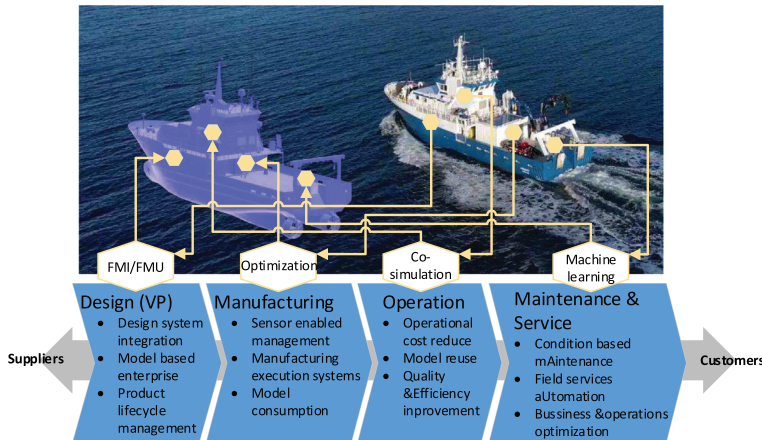


Fig. 2. A plausible development procedure of digital twins system for the marine industry.

2. Related work

The following presents existing open-source FMI based co-simulation frameworks, which also support the SSP standard. While the FMI standard enables the same *model* to be simulated in different tools, the SSP standard enables the same *system* to be simulated in different tools. This seems attractive, but in practice there are only a few tools that actually support the SSP standard. These are as follows:

FMIGo! [12] is a software infrastructure to perform distributed simulations with FMI-compatible components that run on all major platforms. Both CS and ME FMUs are supported, where ME FMUs are wrapped into CS FMUs. FMI Go! uses a client–server architecture, where a server hosts an individual FMU. The server and clients components are implemented using C++. The software supports a draft version of the SSP standard. Unfortunately, the development of FMIGo! is currently stagnant and pre-built binaries are not available. On the plus side, FMIGo! provides some quite advanced co-simulation algorithms that could provide better accuracy and/or performance than other frameworks.

FMPy [13] 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 graphical user interface for running and presenting simulation results. Like FMIGo! the software support the SSP standard, but only a draft version.

libcosim [14] is a cross-platform C++library for performing co-simulation. The library was open-sourced in 2020 and ships with support for FMI 1.0 & 2.0 for CS as well as basic SSP 1.0 support. Additionally, libcosim provides a reference implementation of the OSP-IS [15], a newly introduced standard for defining the co-simulation structure. Furthermore, libcosim provides a C interface for easier integration with other languages, as well as a Java wrapper (cosim4j), command line interface (CLI) tool (cosim), and a client/server demo application (cosim-demo-app) provides a basic web interface and plotting capabilities.

OMSimulator [16] is an FMI-based co-simulation tool that supports ordinary (i.e., non-delayed) and Transmission Line Modelling connections. It provides a C-API and language wrappers for this API in Lua and Python. The OMSimulator is available both as a standalone and through OpenModelica [17], which also provides it with a user interface. Additionally a CLI is available.

Other open-source co-simulation tools worth mentioning here are DACCOSIM [18], Maestro [19], Coral [20] and MasterSim [21]. However, these tools does not provide a standardised way of defining the system to be simulated as the SSP standard provides.

It should be noted that neither FMPy nor FMIGo! support version 1.0 of the SSP standard. Rather, they support an older draft version of the standard, which is no longer publicly available and that is not compatible with the released version. This makes the SSP feature quite complicated to use and defeats some of the purpose of the SSP as no other tool can load the system.

The frameworks mentioned above use traditional software architectures centred around a master algorithm and FMI-compatible models. The ECS architecture applied to military simulators are considered in [5,22]. What differentiates the framework introduced in this paper from any of the systems mentioned above is how it integrates co-simulation with an ECS architecture. It allows integration of components handled by different systems to be connected in a co-simulation fashion, with data transfer occurring at discrete communication steps. A system could be generic or represented by more tangible concepts like an FMI master or a physics engine. By adding or removing systems and components the nature of simulation can be changed seamlessly during execution. As behaviour and state are logically separated between systems and components, state is retained even if the behaviour changes.

However, with great flexibility comes great responsibility. Vico does not define any sort of ontology [23]. Thus, there are no pre-defined set of rules related to how a simulation is designed or what a certain set of components represent. In [2] the authors applied the concept of semantic traits to their ECS, in order to perform compile time checks and to detect an entity's class affiliation

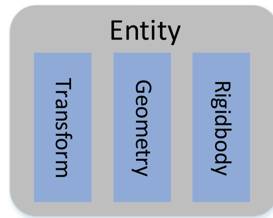


Fig. 3. Anatomy of an entity. An entity is a collection of components. Components can be seamlessly added and removed, which effectively changes what the entity represents.

and allow functionality changes during run-time. Similarly, using the family concept described later, Vico handles run-time detection of entity types without the need of pre-declaring an ontology. As Vico puts no restraint on the type of objects created, it is the users responsibility to avoid ill-formed simulations. However, much of this responsibility can be delegated using a standardised format like the SSP. Another related standard is the ontology based OSP-IS, which provides the means of adding semantic meaning to model interface variables. Ontology is also applied in [24] to describe simulation model parameters in a simulation system independent way.

3. Software architecture

This section introduces Vico, a high-level co-simulation framework based on the ECS software architecture. Implementing Vico around an ECS architecture provides a number of benefits, such as a clear separation between state and behaviour, flexibility, and extensibility. The framework is designed so that physics engines and other types of systems that are not FMUs can be integrated into a co-simulation setting. Many students at NTNU Ålesund are also exposed to the related EC architecture from using Unity3D, which should make the concept of ECS easier to reason with. Vico is written in Kotlin/JVM, a strongly typed language 100% interoperable with Java, which in turn allows it to be used as a library by any JVM language. The fact that Vico runs on the JVM makes it very accessible and easy to extend with the vast amount of high quality libraries covering most needs imaginable. It also makes the system more approachable to students at NTNU Ålesund, which has a long history of teaching Java in their courses. Building and developing software is generally easier on the JVM, especially for many students, compared to a native tool-chain, which is often employed by simulators. Not only that, but with the recent developments of GraalVM [25], a JVM run-time with support for polyglot programming, it is possible to extend or embed Vico using JavaScript, R, or Python code without any additional run-time overhead.

Some of the main features of Vico are as follows:

1. ECS-based software architecture that allows discrete connections between components.
2. Support for FMI 1.0 & 2.0 for co-simulation.
3. Support for SSP 1.0.
4. A CLI for simulating single FMUs and systems of FMUs described using SSP.
5. 3D visualisation and 2D plotting capabilities.
6. Modular, easy to extend framework.
7. Implemented in Kotlin, 100% interoperable with other JVM languages like Java.

A description of some of the core elements used within the context of Vico is given below.

3.1. Entity

An entity is basically just a collection of components as illustrated by Fig. 3. By adding the correct components to an entity, any type of simulation object can be created. In a pure ECS, entities may be represented simply by an integer. In Vico, however, an entity is an object with a (unique) name and an optional tag. This makes it possible to look up an entity once it has been added to the simulation. An entity is a concrete class and cannot be extended.

3.2. Component

A component contains data. Additionally, a Vico component can define so-called properties, which can be used in connections between components. While the data within a component can be of any type, properties can only be of type integer, double, boolean or strings. This ensures compatibility with the FMI standard. Only data that are meant to be plotted, exported to file, or used in connections need to be mapped to a property.

3.3. Family

In naive ECS implementations, every system iterates through the complete list of all entities, and selects only those entities that should be processed. This work is repetitive and cumbersome for the user. Furthermore, it makes systems difficult to reason about as their ontology is not explicit. The concept of families found in ECS implementations such as Ahsley [26] are a way to mitigate this. A family is a list of entities that all contain or exclude a specific set of component types. As components are added or removed from an entity, its family changes. Subsequently, this triggers an add/remove event that is pushed to subscribers, e.g. systems, which act accordingly. This process ensures that a system only iterates through the relevant entities. This might increase performance, especially when component changes are infrequent. However, the main reason for incorporating this feature is to improve usability. Families provides an ontology to systems that ensures that the entities available are limited to those it has explicitly asked for. This helps reducing code-bloat as certain assertions are made superfluous and enables self-documenting code.

3.4. System

A system subscribes to a given family of entities, and is responsible for acting upon or mutating the state of the relevant components belonging to the entities in those families. For example, a *PhysicsSystem* may subscribe to a family of entities that hold a *Transform*, *Geometry*, and a *Rigidbody*. Adhering to the laws of physics, this system will then update the position and rotation of the component during each simulation step. As behaviour and state is separated between systems and components, this allows use-cases where the physics implementation can be changed on the fly simply by replacing the system. Some ECS architectures let each system run in a separate thread, continuously updating components. In Vico, however, systems are stepped forward in time explicitly by the engine to ensure determinism. As systems might potentially act on the same set of components, systems are assigned a priority, which ensures that changes are performed in a user determined order.

3.5. Engine

The engine is the heartbeat that controls and connects every part of the architecture together. As illustrated by Fig. 4, the engine consists of an *EntityManager*, a *SystemManager*, a *ConnectionManager* and a *InputManager*, which, as the naming suggest, handles aspects related to entities, systems, connections, and peripheral input respectively. The *EntityManager* also plays a role as the *ComponentManager* found in some ECS implementations. Unlike common game engines with an ECS architecture, Vico's rate of simulation is not dependent on the variable rendering speed of the graphics processing unit. Rather it may only be stepped using a user provided step-size. In order to achieve real-time execution of the simulation, the engine provides access to a wrapper class called *EngineRunner* that allows the user to control the real-time factor (RTF) of the simulation. By setting the RTF to 1.0, the system will try to synchronise the wall-clock and simulation-clock—slowing down the simulation if necessary.

3.6. Connections

Component *properties* can be connected, allowing data transfer between components during discrete communication intervals. This allows FMI components to be connected with other types of components that are not FMUs, such as rigid bodies. It is possible to apply modifier functions to connections that will modify the output value before it is applied to the output, for example to convert a unit or to apply a filter.

3.7. Scenarios

Scenarios in the context of Vico are pre-configured actions to be executed at specific time points or events during the simulation. Scenarios can be specified to last for a limited time period only, after which any variables that may have changed will be reset to its original value, e.g. to simulate a fault. Scenarios are written in Kotlin, even when provided as standalone input files, which are interpreted as scripts. Unlike typical configuration file formats like JSON, XML or YAML, Kotlin allows users to use logical expressions and otherwise use the full potential of the JVM when writing scenario logic.

3.8. Add-on modules

An overview of the available software modules for Vico are shown in Fig. 5. Much like a game engine, the core Vico module does not provide much functionality other than providing the infrastructure to develop generic co-simulations. However, a number of complementary components and systems are provided. The *Transform* for instance, holds a position and rotation in 3D space. These components can be parented to another so that when the parent transform changes, the child will move with it. In order to add 3D representation to an entity, a *Geometry* is available. Both of these components are required for rendering. A *GeometryRenderer* is also available, which transform the data provided by the components to actual objects rendered on the screen. 3D visualisation can be configured in code or through an XML configuration file, which is especially useful as this allows users to enable 3D visuals when invoking Vico through the provided CLI, described in more detail later. As the 3D graphics window allows for capturing mouse and keyboard events, these inputs could potentially be used, for example to interact with the simulation dynamically in order to more intuitively understand how a system behaves.

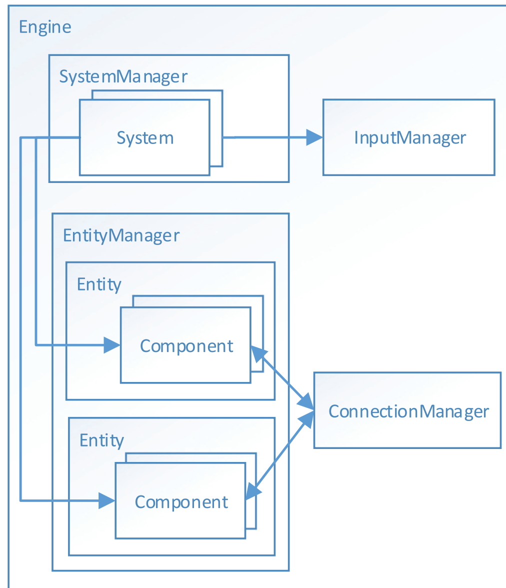


Fig. 4. Composition of the various elements found in the Vico ECS implementation.

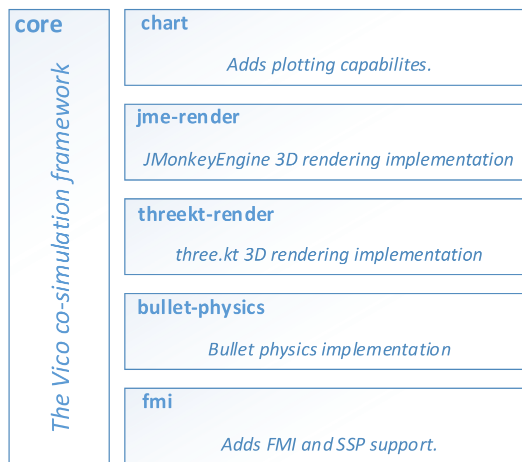


Fig. 5. Overview of available software modules.

Vico also provides a set of generic physics components, such as rigid bodies and constraints through the *physics-api* module. For example, the *Rigidbody* makes an entity subject to the laws of physics. However, as components have no logic, an entity with a *Rigidbody* will not fall to the ground unless some sort of *PhysicsSystem* is added to the simulation. However, in order for the rigid body to move, it needs a position (Transform) and in order for it to collide it needs a 3D representation (Collider). A system that makes use of these physics components, adding behaviour to the entities holding them, have been implemented using the Bullet [27] physics engine, which is available through a module named *bullet-physics*.

A module named *fmi* adds support for FMI 1.0 & 2.0-based co-simulation, and relies on FMI4j [28] for interacting with FMUs. Since FMI4j was initially released, it has changed the way it interacts with native code, making it the fastest open-source JVM library for simulating FMUs. The library also supports export of FMUs compatible with FMI 2.0 for co-simulation and provides a Gradle plugin to simplify the usage of this feature. This allows for a workflow where slaves can be automatically exported to FMUs during the build process and loaded by Vico within the same project. The *fmi* module adds a system named *SlaveSystem* that takes an instance of *MasterAlgorithm*, which is an interface, as a constructor parameter. The idea is that users should be free to develop

their own master algorithm. However, the module also provides a ready-to-use implementation of a fixed step-size master algorithm, which allows users to configure slaves to run at different rates. Now, due to the fact that FMUs comprise both behaviour and state, they are difficult to fit into an ECS architecture, as they fit into neither a component nor a system. This is solved in Vico by creating a component that represents the location of an FMU. This component also contains a buffer for variable writes and a cache for variable reads. The system then loads the FMU from the path specified and continuously updates the cache/buffer. This enables read and write operations to be performed in bulk and access to variable values are cached, which help maintain performance as simulations become more complex. This is especially true if the underlying FMU operations are slow due to internal implementation details such as networking. SSP support is also provided by the *fmi* module. Currently, there is no up-to-date list of tools that support the SSP standard, and the authors are only aware of two other non-commercial tools that support version 1.0, namely OMSimulator and libcosim. None of these support the entire standard, but they support enough features to support common use-cases. FMPy and FMIGo!, mentioned in the previous section, only supports an out-of date draft version, from which the documentation is no longer publicly available. Furthermore the draft version is different between the tools, both of which are incompatible with each other. Like OMSimulator and libcosim, Vico supports a limited set of the SSP 1.0 standard, where additional features might be implemented as use-cases appear.

Being able to make sense of a simulation while it unfolds or immediately afterwards is quite valuable, which is why Vico offers support for plotting time-series and XY charts. The properties of these plots can be defined using an XML input file or configured in code. The plots can be configured to be shown and updated live or at the end of a simulation run.

3.9. Command line interface

To allow non-programmers and to enable easier access to the software in general, Vico ships with a pre-built and cross-platform CLI. The top-level commands are presented in Listing 1. In turn, these takes additional parameters, which may be investigated by invoking the command with no arguments provided.

Listing 1: Vico command line interface

```
Usage: <main class> [-h] [COMMAND]
  -h, --help     Display a help message.
  -v, --version  Prints the version of this application.
Commands:
  simulate-fmu   Simulate a single FMU.
  simulate-ssp   Simulate an SSP co-simulation system.
```

The *simulate-fmu* command takes an FMU as input and simulates it. This is mostly useful for testing an FMU that would normally be used as a building block in a larger system, whereas the *simulate-ssp* command takes an SSP archive as input and simulates it. In both cases, the simulation can be decorated with 2D plots, 3D visualisations, and scenarios by specifying additional input files.

3.10. Scripting

Vico itself does not provide scripting, but the implementation language Kotlin does. This makes it natural to use Vico in a scripting context. A scripting example is provided in Listing 2. This example shows the modularity of Vico, as modules are included as required. The script file can be executed within the context of IntelliJ IDE or in a shell on any system with a stand-alone Kotlin compiler. This could be an easier way to develop and distribute use-cases than creating Maven or Gradle projects, as is common when developing on the JVM. A custom Domain Specific Language (DSL) is also available, aimed at easing the creation of Vico simulations.

Listing 2: Using Vico with Kotlin scripting.

```
@file:Repository("https://dl.bintray.com/ntnu-ihb/mvn")

@file:DependsOn("no.ntnu.ihb.vico:core:0.x.x")
@file:DependsOn("no.ntnu.ihb.vico:chart:0.x.x")
@file:DependsOn("no.ntnu.ihb.vico:jme-render:0.x.x")

import no.ntnu.ihb.vico.core.*

Engine().use { engine ->
  ...
}
```

4. Case studies

This section describes two case-studies to show the effectiveness of the Vico framework. The first case-study is used to compare the accuracy and performance of Vico against other SSP compatible co-simulation frameworks using a simple quarter-truck system. The second case-study shows a more complex co-simulation of the NTNU owned research vessel Gunnerus, demonstrating parallel performance as well as the 3D and plotting capabilities of Vico.

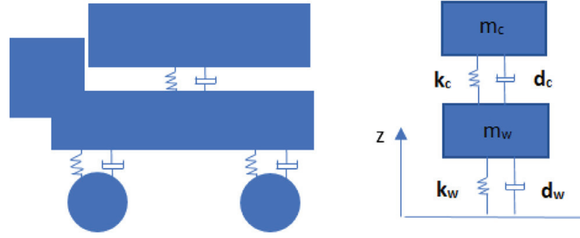


Fig. 6. Illustration of the quarter-truck system.

Table 1
Input and output variables of the quarter-truck models used for connections.

FMU	Variable	Input/output	Description
Chassis	F_{susp}	Output	Chassis suspension force applied to the wheel.
	$\dot{z}_w$	Input	Velocity of the wheel from the wheel model.
Wheel	F_{susp}	Input	Chassis suspension force from the chassis part.
	$\dot{z}_w$	Output	Velocity of the wheel sent to the chassis part.
	F_{tyre}	Output	Tyre force applied to the ground.
	$\dot{z}_g$	Input	Ground profile, given as vertical velocity variation from the ground model.
Ground	F_{tyre}	Input	Tyre force from the truck wheel.
	$\dot{z}_g$	Output	Ground profile, given as vertical velocity variation sent to the wheel.

Table 2
Summary of tools included in the case study.

Tool	Main implementation language	Platform support	FMI version	SSP version
FMPy	Python	Win, Linux, Mac	1.0 & 2.0 for CS	Draft20171219
FMIGo!	C++	Win, Linux, Mac	2.0 for CS & ME	Draft20170606
libcosim	C/C++	Win, Linux	1 .0 & 2.0 for CS	1.0
OMSimulator	C/C++	Win, Linux, Mac	1 .0 & 2.0 for ME & CS	1.0
Vico	Kotlin/JVM	Win, Linux	1 .0 & 2.0 for CS	1.0

4.1. Quarter-truck case-study

In the following case study, the tools listed in Table 2 are used to load and simulate the same models representing a simplified quarter-truck system, also known in the literature as a quarter-car system [29–31]. The system for simulation is defined using the FMI and SSP standards in order to test performance in terms of accuracy and efficiency [32]. The model for the quarter-truck system is illustrated in Fig. 6 with m_w and m_c representing the mass of wheel and chassis respectively. Both masses have a single vertical degree of freedom coupled by a linear spring–damper system representing the chassis suspension and wheel tyres. The ground profile is given as external input. The co-simulation system representing the quarter truck is comprised of three models: the *chassis* including the suspension, the *wheel* including the tyre and the *ground*. The input and output variables used to connect these models are given in Table 1.

As a benchmark for the simulation accuracy, the analytical model for the system is derived. The suspension force and the tyre force are given by Eq. (1), while the equations of motion for the chassis and wheel are given by Eq. (2).

$$\begin{aligned} F_{susp} &= k_c(z_w - z_c) + d_c(\dot{z}_w - \dot{z}_c) \\ F_{tyre} &= k_w(z_g - z_w) + d_w(\dot{z}_g - \dot{z}_w) \end{aligned} \quad (1)$$

$$\begin{aligned} m_c\ddot{z}_c &= F_{susp} - m_c g \\ m_w\ddot{z}_w &= F_{susp} - F_{tyre} - m_w g \end{aligned} \quad (2)$$

Vico, OMSimulator, and libcosim load the same .ssp, while FMPy and FMIGo! both require a slightly modified version that is compatible with the draft version they use. In practice, however, there is no practical difference. The system is simulated using the default master algorithm for each tool, which in all cases is some form of a fixed-step algorithm. Each tool comes with a CLI, which is used to run the simulation. A reference solution has been computed by means of Euler method, with the integration time step set to 0.001 s. Co-simulation results are shown using both a 100 hz and 1000 hz fixed-step-size for the master algorithms. Fig. 7 and Fig. 8 shows the vertical displacement of the wheel and chassis respectively when simulated at 100 hz. In this case, none of the tools are very accurate and they highlight one of the inherent weaknesses of co-simulation compared to monolithic simulations. FMPy also appears to constantly provide output timestamped one time-step earlier than the other tools. libcosim and FMPy both appear to generate stronger oscillations during the first second of simulation. This response can be seen more in detail through Fig. 9. The authors of libcosim have been made aware of this issue, and it should be fixed in a later release if it turns out to be some kind of

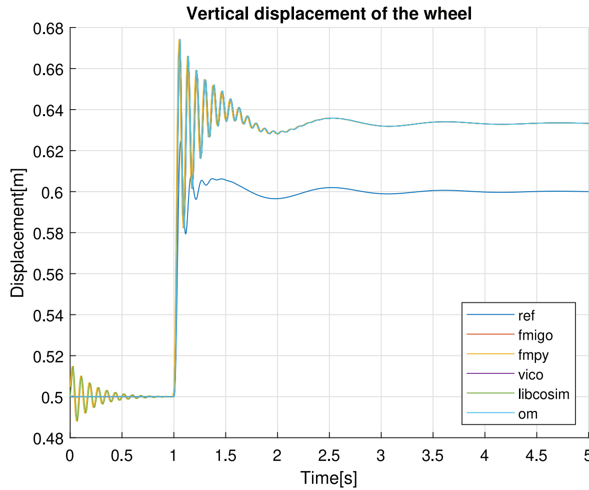


Fig. 7. Wheel response when simulated at 100 hz. The simulation starts from equilibrium state where $z_w = 0$. The ground profile is defined as a step function excited by a jump of 0.1 m in vertical direction at 1 s.

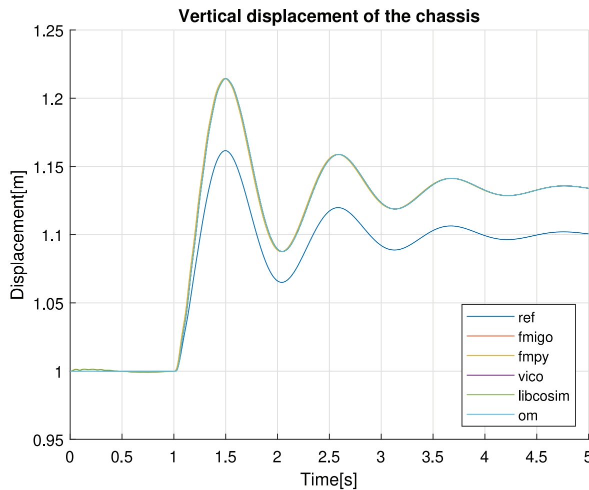


Fig. 8. Chassis response when simulated at 100 hz.

Table 3
Root mean square error of the computed vertical displacement of the wheel.

Tool	RMSE	
	100 hz	*1000* hz
FMPy	0.0300358	0.0019367
FMIGo!	0.030062	0.0018814
libcosim	0.030109	0.0018815
OMSimulator	0.030062	0.0018814
Vico	0.030062	0.0018814

initialisation issue. Naturally, simulating the system at 1000 hz shows a clear improvement in accuracy as can be seen in Figs. 10 and 11. In this case there are barely any differences regarding simulation results between the tools and the reference solution. The improvement with respect to root mean square error (RMSE) can be seen in Table 3. The increase in accuracy comes, however, with a run-time cost.

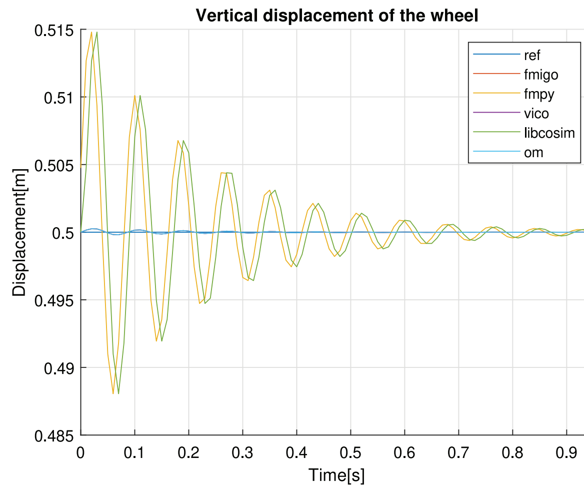


Fig. 9. Detailed view of the first second of simulation presented in Fig. 7.

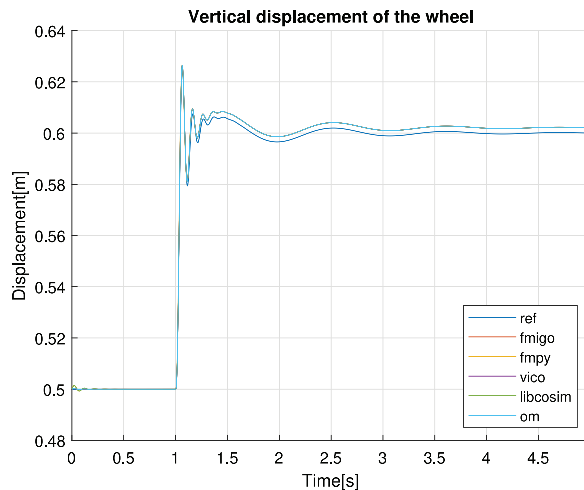


Fig. 10. Wheel response when simulated at 1000 hz.

The results of a performance benchmark appear in Fig. 12 in the form of box-plots. The benchmark is performed on a 64-bit Windows 10 system equipped with a Intel Core i5-3570 CPU with four logical processors. Each tool has been run 15 times, simulating the system for 1000 s with a step-size of 0.001 s. FMI Go! and FMPy both exports a handful of variables to CSV. libcosim, OMSimulator and Vico is run both with and without exporting all 121 available variables to CSV. Additionally, OMSimulator also exports in MATLAB format.

Although Vico is implemented on the JVM, which involves some inherent overhead due to the fact that it must cross the native bridge when it communicates with FMUs, it is the fastest of the tools participating in the benchmark. OMsimulator is the second fastest, ahead of FMIGo!. The results of FMIGo! are quite impressive, considering that it is the only one of the tools to run distributed. Next is libcosim, followed by FMPy. It is not surprising that FMPy is the slowest tool, as Python is not known to be a particularly fast language. OMSimulator and Vico are configured to run this particular system single-threaded, which libcosim has no option to do, which may explain its poor performance. As the individual models in the system are computationally inexpensive, it would seem that the inherent overhead of handling threads/fibers/co-routines is actually degrading performance. Both OMSimulator and Vico were tested with multiple threads, and Vico in particular showed over a 2x performance increase when running single-threaded. A couple of things should be noted about OMSimulator. When exporting simulations results to *.csv* rather than *.mat*, the performance deteriorates significantly—going from a mean 19.2 s to about 150 s. Note, however, that the performance indicators presented here are only valid for this particular system and should not be used as a general pointer to how well the various tools perform.

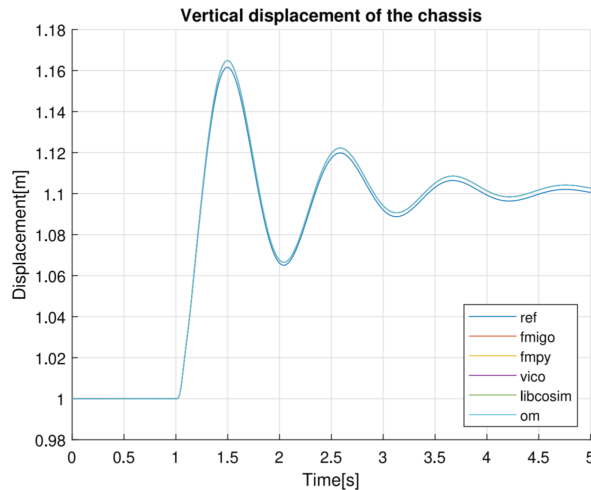


Fig. 11. Chassis response when simulated at 1000 hz.

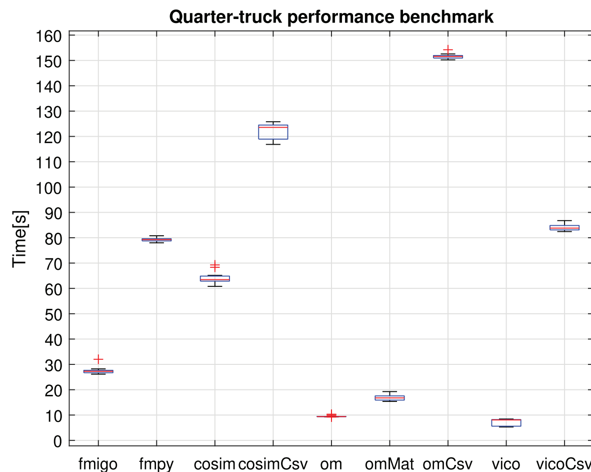


Fig. 12. Performance of the various tools when considering the presented quarter-truck system. Simulation time = 1000 s, step-size = 0.001 s, number of runs = 15.

4.2. Gunnerus case-study

Fig. 13 demonstrates how Vico's built in 3D graphics and plotting capabilities are used to support ongoing research projects at NTNU Ålesund, such as the TwinShip KPN project. Here, a model of the research vessel Gunnerus is performing a path following simulation. The blue line in the 3D visualisation shows the most recent trajectory of the vessel, while the green cylinder shows the current way-point that the vessel should navigate towards. As the vessel comes within reach of the target way-point, a new one appears and the process continue. The modelled Gunnerus vessel is an aggregation of eight FMUs, including a hull model, thrusters, controllers, and power utilities, the structure of which are defined using the standardised SSP format. The properties of the visualisation and file logging are specified through separate XML configuration files. The SSP along with the run-time configurations can then be supplied as arguments to the Vico CLI. This example makes use of several Vico features, including FMI, SSP, 3D visuals, and distributed execution of FMUs. Distributed execution is facilitated using FMU-proxy, which is compatible with any FMI 2.0 based tools and works by wrapping an existing co-simulation FMU into a new one that internally employs a client/server architecture. Some FMUs, like the thruster used in this example can only be instantiated once per process. This is clearly an issue as the hull requires two thrusters. However, FMU-proxy overcomes this by running model instances in separate processes.

Fig. 14 shows the performance of Vico compared to libcosim and OMSimulator when simulating the Gunnerus system. FMU-proxy is used in order to make the system, which originally consisted of both FMI 1.0 & 2.0 FMUs, compatible with OMSimulator.

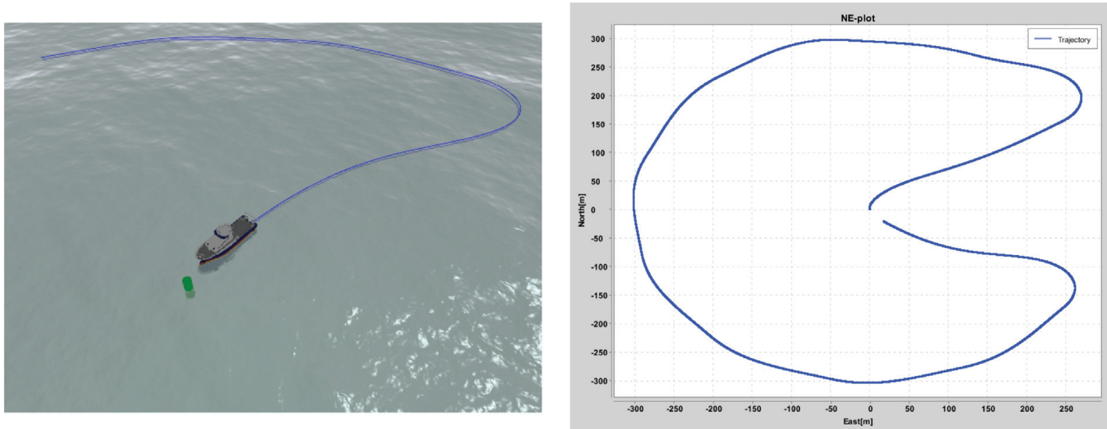


Fig. 13. Demonstration of a vessel path following simulation running in Vico with 3D visualisation and plotting enabled. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

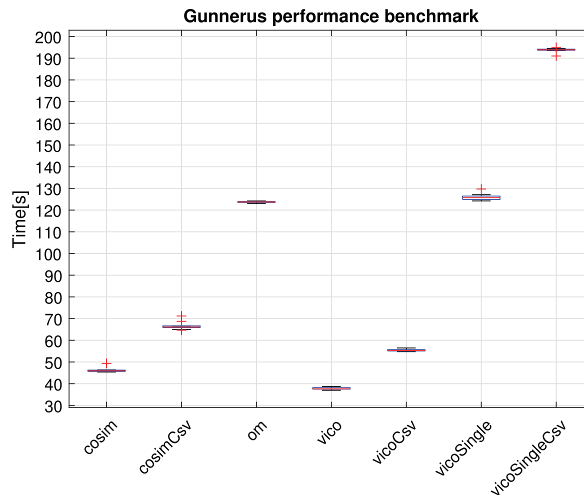


Fig. 14. Performance of libcosim and Vico when considering the Gunnerus system. Simulation time = 1000, step-size = 0.05 s, number of runs = 10.

An attempt were made in order to run the system in the same set of tools as for the quarter-truck, but adopting the SSP file to the obsolete versions used by FMPy and FMIGo! proved difficult and attempts to simulate the system in those frameworks were unsuccessful. The benchmark is performed on a 64-bit Windows 10 system equipped with a Intel Core i7-8700 CPU with twelve logical processors. The simulation is run 10 times, simulating the system for 1000 s with a step-size of 0.05 s. Vico and libcosim performs the simulation both with and without exporting available time-series data, while OMSimulator is configured to not record time-series data. The system contains a total of 3006 variable values that must be retrieved from the various model instances at each time-step and later written to disk. Furthermore, the use of FMU-proxy means that networking is involved. Both Vico and libcosim implements a strategy to optimise variable reads and writes, however, it seems OMSimulator does not. Because of this, OMSimulator is not able to simulate the system in a timely manner when also set to export time-series data. For example, it took OMSimulator approx. 250 s to simulate 40 s. To compare, Vico used approx. 58 s to simulate 1000 s. Furthermore, Vico runs the simulation both single- and multi-threaded. Compared to the quarter-truck system, this simulation benefits from parallel execution in terms of performance. The difference between Vico and libcosim is less in this case, but Vico still performs better when utilising multiple threads. Even with the additional overhead of exporting time-series data, both Vico and libcosim perform better than OMSimulator. This is related to how variable reads and writes are handled by the frameworks. Basically, OMSimulator seems to perform variable reads and writes on individual variables, while libcosim and Vico execute these operations in bulk. This puts the performance of OMsimulator, which runs in parallel, in the vicinity of Vico in single-threaded mode.

5. Conclusions and future work

This paper introduced Vico, a novel co-simulation framework based on the ECS software architecture most commonly found in games. The proposed architecture provides a number of benefits, such as flexibility, extensibility, and a clear separation between state and behaviour. Furthermore, the framework has been designed so that physics engines and other types of systems that are not FMUs can be integrated into a co-simulation setting. Choosing to implement Vico using a JVM language also brings benefits, such as strong tooling, a simple build process, and a vast number of available libraries. Additionally, NTNU Ålesund has a long history of teaching Java in their courses, which should make the framework more approachable to students here. Furthermore, many of these students are exposed to the related EC architecture from game engines like Unity3D, which should make the concept of ECS easier to relate to.

The presented case-studies showed that Vico is effective compared to other open-source co-simulation tools and demonstrated support for the well established FMI standard for co-simulation, as well as the newer, less established SSP standard for defining the simulation structure. Moreover, a number of built-in features like support for 3D visualisation, 2D plotting, export of time-series data as CSV files and distributed execution of FMUs was shown. In the presented quarter-truck case-study Vico was shown to be the fastest tool and provided no-less of an accuracy than the other co-simulation frameworks using their default solver. The Gunnerus case-study showed the visual capabilities and parallel performance of Vico, and also demonstrated the importance of efficient variable handling in larger, more complex co-simulations.

Vico is under continuous development and further work includes:

1. Implementation of additional state-of-the-art co-simulation masters.
2. Adding a web-server plugin that allow web-clients to monitor and modify the simulation.
3. Implementation of additional SSP features as use-cases appear.
4. Integration with additional physics engines available on the JVM.

Furthermore, it would be interesting to explore the multi-platform capabilities of the Kotlin language in order to allow the core Vico engine to support not only the JVM, but also JavaScript and native targets. While plausible, this would require some effort and should therefore be motivated by a sound use-case, which has not emerged to date.

The source code of Vico is hosted at https://github.com/NTNU-IHB/Vico under a permissive MIT license.

Acknowledgements

This work was supported in part by the Project "Digital Twins for Vessel Life Cycle Service", under Grant 280703 from Research Council of Norway, and in part by the Project "SFI Offshore Mechatronics", under Grant 237896 from Research Council of Norway.

References

[1] D. Adam, Entity Systems are the future of MMOG development, 2007, URL: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. (Date Accessed 31 August 2020).
[2] D. Wiebusch, M.E. Latoschik, Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems, in: 2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS, IEEE, 2015, pp. 25–32.
[3] T. Raffaillac, S. Huot, Polyphony: Programming interfaces and interactions with the entity-component-system model, Proc. ACM Human-Comput. Interact. 3 (EICS) (2019) 1–22.
[4] P. Lange, R. Weller, G. Zachmann, Wait-free hash maps in the entity-component-system pattern for realtime interactive systems, in: 2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS, IEEE, 2016, pp. 1–8.
[5] D.D. Hodson, J. Millar, Application of ECS game patterns in military simulators, in: Proceedings of the International Conference on Scientific Computing, CSC, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2018, pp. 14–17.
[6] T. Blockwitz, 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, 2012.
[7] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, T. Nouidui, An empirical survey on co-simulation: Promising standards, challenges and research needs, Simulat. Model. Pract. Theory 95 (2019) 148–163.
[8] L.I. Hatledal, A. Styve, G. Hovland, H. Zhang, A language and platform independent co-simulation framework based on the functional mock-up interface, IEEE Access 7 (2019) 109328–109339.
[9] J. Köhler, H.-M. Heinkel, P. Mai, J. Krasser, M. Deppe, M. Nagasawa, Modelica-association-project "system structure and parameterization"–early insights, in: The First Japanese Modelica Conferences, May 23–24, Tokyo, Japan, (124) Linköping University Electronic Press, 2016, pp. 35–42.
[10] L.I. Hatledal, H.G. Schaathun, 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, 2015.
[11] L.I. Hatledal, A Flexible Network Interface for a Real-Time Simulation Framework (Master's thesis), NTNU, 2017.
[12] C. Lacoursière, 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, (132) Linköping University Electronic Press, 2017, pp. 653–662.
[13] Catia-Systems, FMPy, 2019, URL: https://github.com/CATIA-Systems/FMPy. (Date Accessed 31 March 2020).
[14] Open Simulation Platform, Libcosim, 2020, URL: https://github.com/open-simulation-platform/libcosim. (Date Accessed 25 August 2020).
[15] Open Simulation Platform, OSP-IS, 2020, URL: https://opensimulationplatform.com/specification/. (Date Accessed 25 August 2020).
[16] L. Ochel, R. Braun, B. Thiele, A. Asghar, L. Buffoni, M. Eek, P. Fritzson, D. Fritzson, S. Horkeby, R. Hällquist, et al., OMSimulator–Integrated FMI and TLM-based co-simulation with composite model editing and SSP, in: Proceedings of the 13th International Modelica Conference, No. 157, Regensburg, Germany, March 4–6, 2019, Linköping University Electronic Press, 2019.
[17] P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli, D. Broman, The OpenModelica modeling, simulation, and development environment, in: 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society, SIMS2005, Trondheim, Norway, October 13–14, 2005.

[18] J.É. Gómez, J.J.H. Cabrera, J.-P. Tavella, S. Vialle, E. Kremers, L. Frayssinet, Daccosim NG: co-simulation made simpler and faster, 2019.
[19] C. Thule, K. Lausdahl, C. Gomes, G. Meisl, P.G. Larsen, Maestro: The INTO-CPS co-simulation framework, Simulat. Model. Pract. Theory 92 (2019) 45–61.
[20] S. Sadjina, L.T. Kyllingstad, M. Rindarøy, S. Skjong, V. Æsøy, E. Pedersen, Distributed co-simulation of maritime systems and operations, J. Offshore Mech. Arctic Eng. 141 (1) (2019).
[21] A. Nicolai, Co-simulations-masteralgorithmen-analyse und details der implementierung am beispiel des masterprogramms MASTERSIM, (Master's thesis), Technische Universität, 2018.
[22] J.A. Vagedes, D.D. Hodson, S.L. Nykl, J.R. Millar, ECS Architecture for modern military simulators, in: Proceedings of the International Conference on Scientific Computing, CSC, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2019, pp. 118–122.
[23] P. Benjamin, M. Patki, R. Mayer, Using ontologies for simulation modeling, in: Proceedings of the 2006 Winter Simulation Conference, IEEE, 2006, pp. 1151–1159.
[24] F. van Wermeskerken, G. Ferdinandus, T. van den Berg, K. Bosch, R. Smelik, H. Henderson, Simulation independent model configuration, in: Proceedings of the 2018 Winter Simulation Innovation Workshop, SIW, Orlando, FL, 2018.
[25] F. Niephaus, T. Felgentreff, R. Hirschfeld, Towards polyglot adapters for the graalvm, in: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, 2019, pp. 1–3.
[26] libgdx authors, ashley, 2020, URL: https://github.com/libgdx/ashley. (Date Accessed 16 November 2020).
[27] E. Coumans, et al., Bullet Physics Library, Vol. 15, No. 5, Open Source: Bulletphysics. org, 2013, p. 5.
[28] L.I. Hatledal, H. Zhang, A. Styve, G. Hovland, Fmi4j: A software package for working with functional mock-up units on the java virtual machine, in: The 59th Conference on Simulation and Modelling, SIMS 59, Linköping University Electronic Press, 2018.
[29] M. Gull, O.F. Ergin, S. Savas, Control of quarter car model by co-simulation with adams and matlab, Int. J. Res. Appl. Sci. Eng. Technol. 6 (2018).
[30] T. Lundberg, Analysis of simplified dynamic truck models for parameter evaluation, 2013.
[31] P. Li, Q. Yuan, Influence of coupling approximation on the numerical stability of explicit co-simulation, J. Mech. Sci. Technol. (2020) 1–10.
[32] M. Arnold, C. Clauß, T. Schierz, Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation V2. 0, in: Progress in Differential-Algebraic Equations, Springer, 2014, pp. 107–125.

Lars Ivar Hatledal received the B.Sc. degree in automation from NTNU, Ålesund, Norway, in 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 an M.Sc. in Simulation and Visualisation also from NTNU, where he is currently working towards a Ph.D. degree with the Department of Ocean Operations and Civil Engineering.

Dr. Yingguang Chu received his Ph.D. degree in Marine Technology in 2018 from Norwegian University of Science and Technology, Norway. Beijing Technology and Business University, China, awarded his B.Sc. degree in Mechanical Engineering and Automation in 2007 and Ålesund University College, Norway, awarded his a M.Sc. degree in Product and System Design in 2013. From 2010 to 2011, he studied with the M.Sc. program in Hydraulic Engineering at Ocean University of China. Since 2018, Dr. Chu has been working as a Research Scientist with Sintef Ålesund AS. His research interests include mechanical design, hydraulics, robotics, modelling and simulation of dynamic operation systems, and virtual prototyping. He has published several papers and articles in international conferences and journals within these areas.

Arne Styve received a B.E. degree (honours) in microelectronics and software engineering from the University of Newcastle upon Tyne, in the 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 software industry, having worked in areas including 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.

Prof. Houxiang Zhang received both M.Sc. and Ph.D. degrees in Mechanical and Electronic Engineering from Robotics Institute, Beihang University, in 2000 and 2003 respectively. From 2004, he worked as a postdoctoral fellow, senior researcher 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 of Mechatronics. Dr. Zhang has engaged into two main research areas, including biological locomotion control and digitalisation and virtual prototyping in demanding marine operation. He has applied for and coordinated more than 30 projects supported by the Norwegian Research Council, German Research Council, EU, and industry. He has published over 200 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. He has been elected to the Norwegian Academy of Technological Sciences in 2019.

ISBN 978-82-326-5777-3 (printed ver.)
ISBN 978-82-326-5231-0 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

NTNU
Norwegian University of
Science and Technology