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

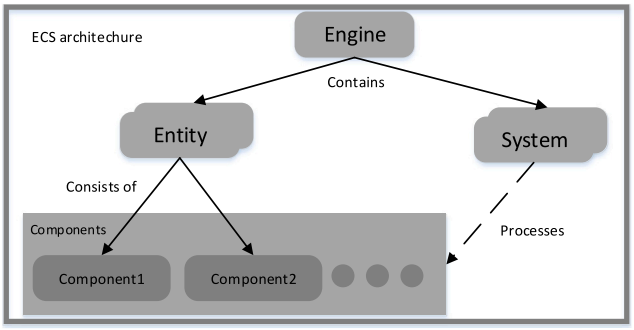

Fig. 1. High level overview of the ECS architecture.

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.

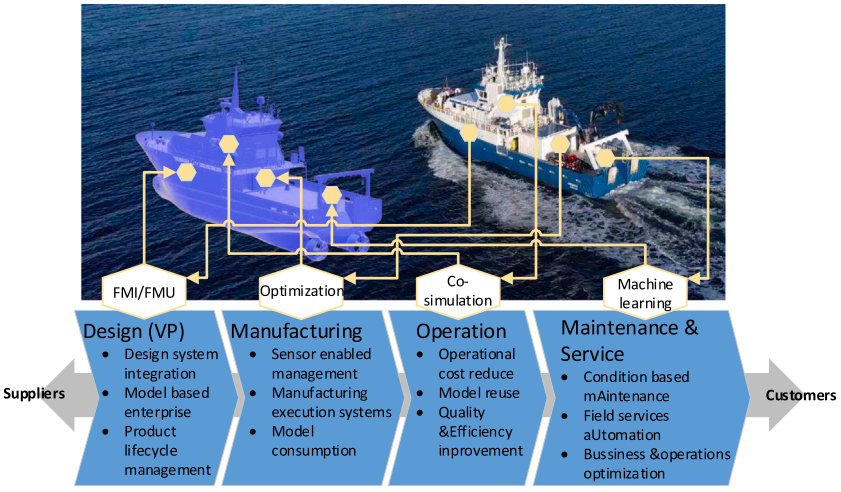

Fig. 2. A plausible development procedure of digital twins system for the marine industry.

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

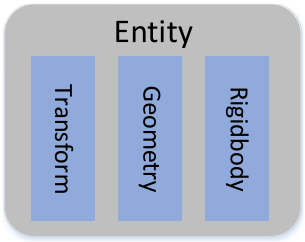

Fig. 3. Anatomy of an entity. An entity is a collection of components. Components can be seamlessly added and removed, which effectively changes what the entity represents.

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.

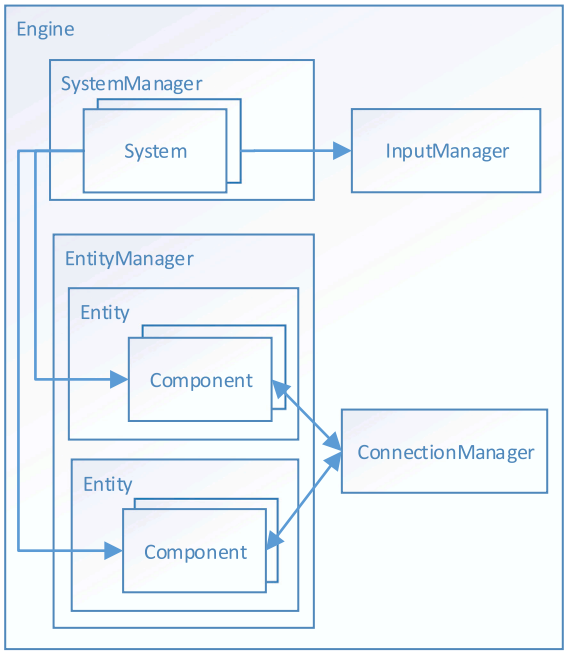

Fig. 4. Composition of the various elements found in the Vico ECS implementation.

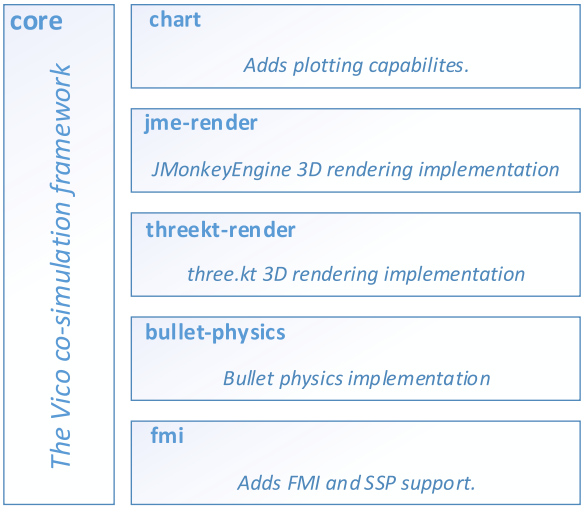

Fig. 5. Overview of available software modules.

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.

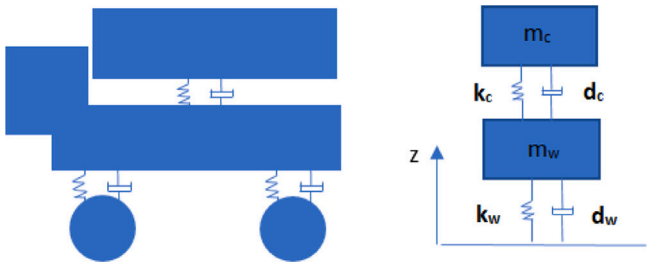

Fig. 6. Illustration of the quarter-truck system.

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).

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

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

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

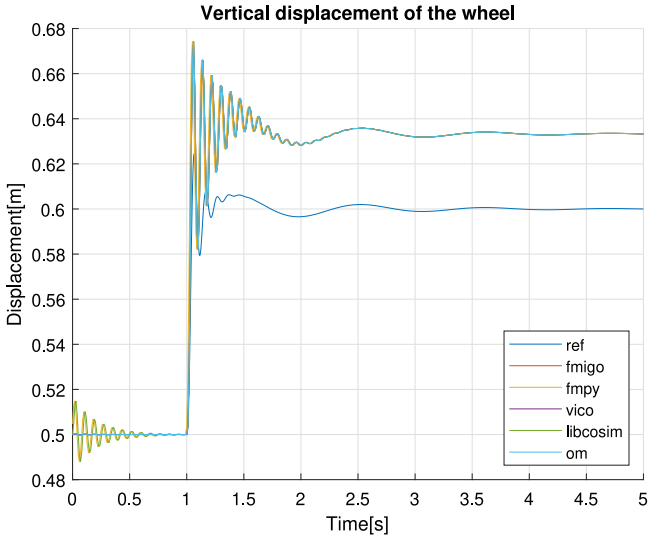

Fig. 7. Wheel response when simulated at 100 hz. The simulation starts from equilibrium state where $z_w = 0$. The ground profile is defined as a step function excited by a jump of 0.1 m in vertical direction at 1 s.

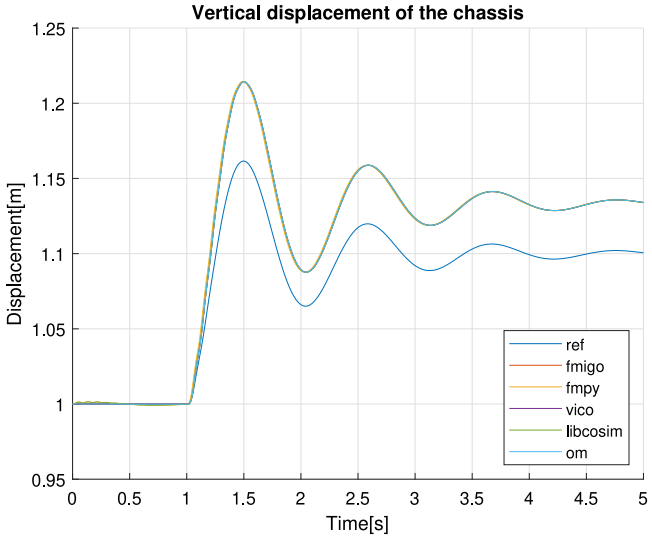

Fig. 8. Chassis response when simulated at 100 hz.

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.

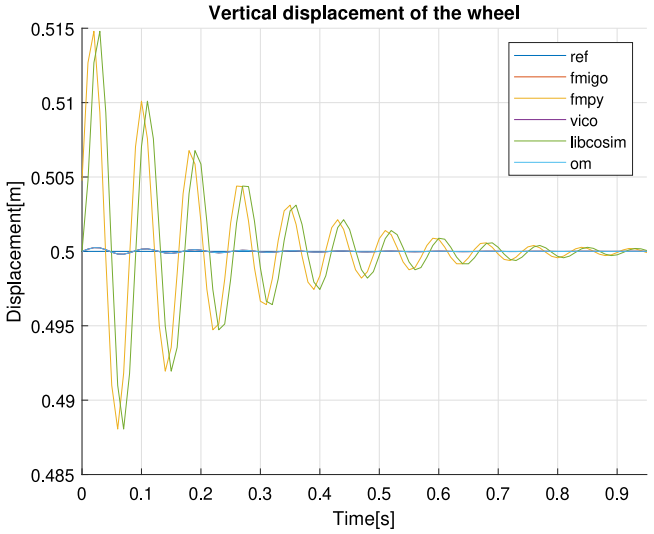

Fig. 9. Detailed view of the first second of simulation presented in Fig. 7.

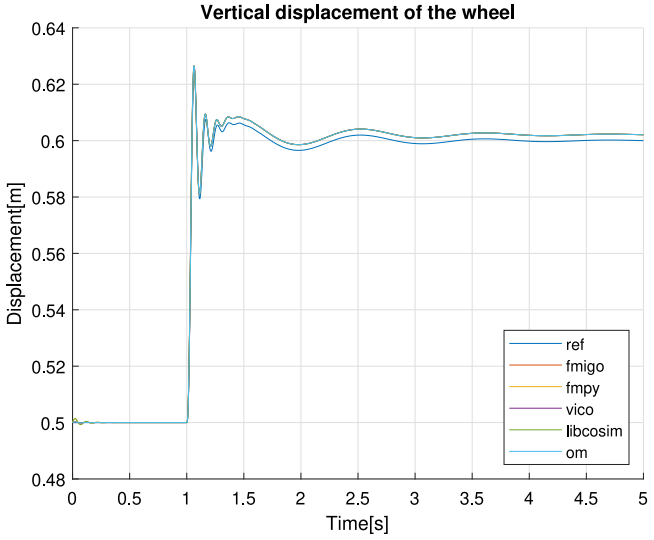

Fig. 10. Wheel response when simulated at 1000 hz.

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.

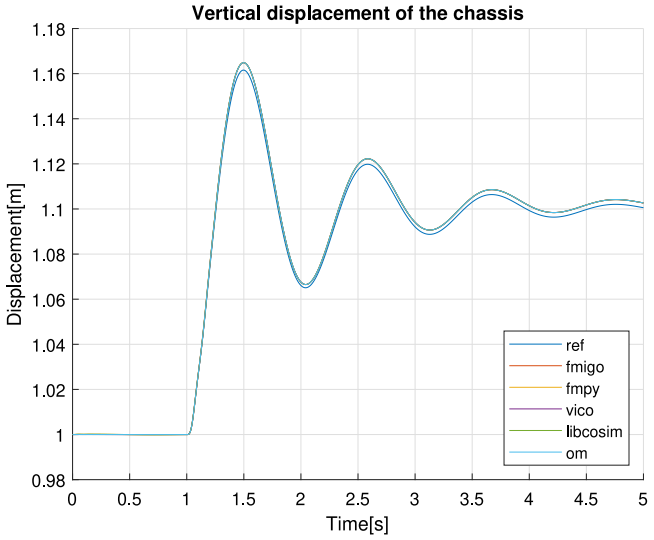

Fig. 11. Chassis response when simulated at 1000 hz.

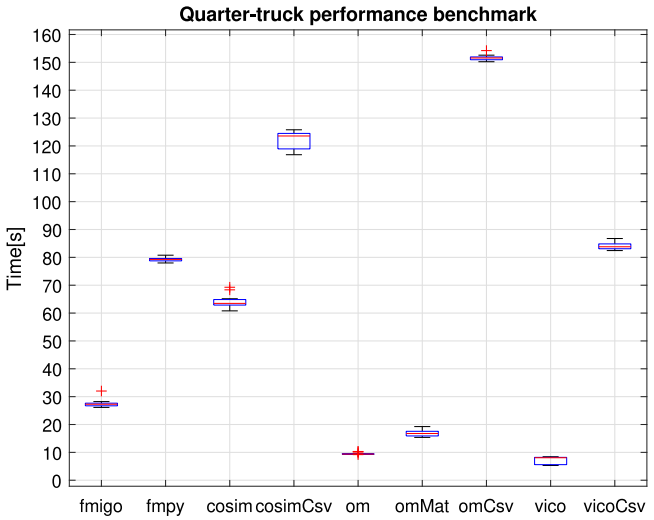

Fig. 12. Performance of the various tools when considering the presented quarter-truck system. Simulation time = 1000 s, step-size = 0.001 s, number of runs = 15.

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.

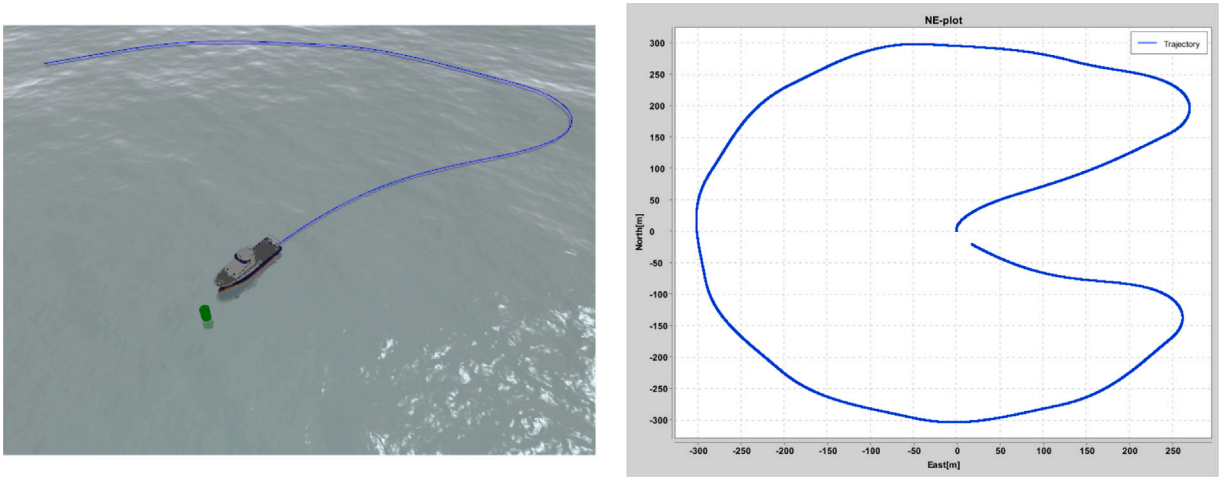

Fig. 13. Demonstration of a vessel path following simulation running in Vico with 3D visualisation and plotting enabled. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

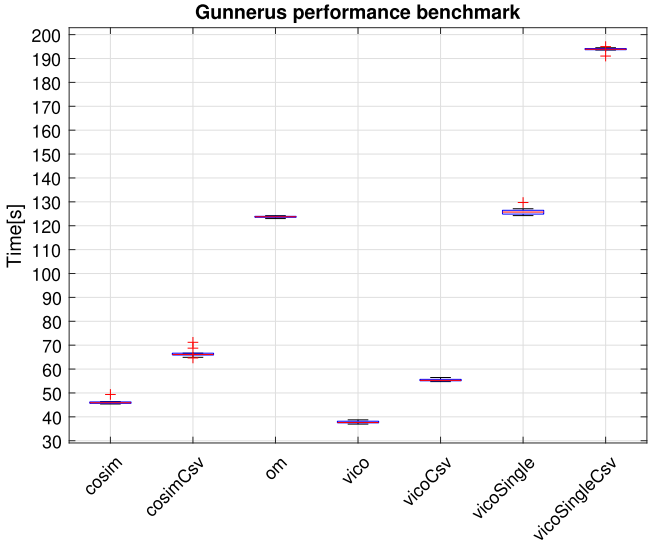

Fig. 14. Performance of libcosim and Vico when considering the Gunnerus system. Simulation time = 1000, step-size = 0.05 s, number of runs = 10.

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.