Integrating the Functional Mock-up Interface with ROS and Gazebo

Ralph Lange[1]**, Silvio Traversaro[2], Oliver Lenord[1], and Christian Bertsch[1]

[1] Robert Bosch GmbH, Robert-Bosch-Campus 1, 71272 Renningen, Germany,
firstname.lastname@de.bosch.com
[2] Italian Institute of Technology, Via Morego, 30 16163 Genova, Italy,
firstname.lastname@iit.it

Abstract. The *Functional Mock-up Interface* (FMI) is a widely used industry standard for exchange and co-simulation of dynamic models as *Functional Mock-up Units* (FMU). It is supported by more than 100 modeling and simulation tools. In this chapter, we present two implementations of FMI that bridge the gap between these tools and the ROS and Gazebo community: First, the *fmi_adapter* package for running/simulating FMUs in ROS nodes, from `https://github.com/boschresearch/fmi_adapter_ros2`. Second, the *gazebo-fmi* package for integrating FMUs with Gazebo, from `https://github.com/robotology/gazebo-fmi`.
After an introduction to the FMI standard, this chapter provides step-by-step, hands-on examples for both packages, followed by interface descriptions and selected implementation details. In addition to these tutorial-style sections, the chapter also provides comprehensive descriptions of two use-cases. First, it explains how the fmi_adapter enabled a convenient model-based control design workflow for a self-driving vehicle for industrial logistics. Second, it reports on the simulation of electrical actuators in Gazebo from a Modelica model.

Keywords: Functional Mock-up Interface, FMI, FMU, ROS, ROS 2, Gazebo, fmi_adapter, gazebo-fmi

1 Introduction

The *Functional Mock-up Interface* (FMI) is an industry standard for exchange and co-simulation of dynamic models, supported by more than 100 modeling and simulation tools [1], including Dymola, MATLAB/Simulink, OpenModelica, SimulationX, and Wolfram System Modeler. Such tools allow exporting dynamic models as *Functional Mock-up Units* (FMU). Technically, an FMU is a ZIP file containing the differential equations as a shared library and/or C source code together with an XML-based description of the variables, parameters and model structure, i.e. derivatives, initially unknown variables, etc. For co-simulation [2], a numerical solver may be integrated in the FMU.

** Ralph Lange is the corresponding author of this chapter.

A number of tools and libraries, including MATLAB/Simulink, Dymola, SimulationX, MapleSim, AMESim, GT-Suite, and Simpack, allow importing FMUs to compose larger models from existing ones and to simulate them together. This enables to integrate different models from (graphical and textual) modeling languages and tools. The use of a binary format even allows sharing of models while protecting intellectual property. The FMI standard is used in a number of industries and actively developed by an international consortium as explained in the history section of the FMI website [1].

In robotics, dynamic models are used for many aspects including advanced control of mobile platforms and manipulators, modern motion planning algorithms, and object tracking, to name a few. The *fmi_adapter* package [3,4] provides a straightforward integration of the FMI standard with ROS and ROS 2. It offers a generic ROS node as well as a C++ class to load an FMU at runtime and to map the major FMI concepts to ROS and vice-versa. This allows integrating an FMU with a complex dynamic model and solver into a ROS-based system with just a few lines of C++ code or launch code.

For example, for a self-driving intra-logistics vehicle, we created a path filter to avoid high bore friction at the vehicle's caster wheels. Both, the path filter and the underlying kinematic model to estimate the orientations of the wheels were implemented in the Modelica language and exported as an FMU. By the fmi_adapter package, this FMU could be integrated with the ROS-based motion planner and motion controller of the vehicle with only 25 lines of code.

Dynamic models also play a crucial role when testing robotic systems in simulation. Robotic simulators such as AirSim, Gazebo, MORSE, OpenRAVE, SynCity, V-REP, and Webots, are generally based on physics engines for rigid body dynamics. In advanced applications, there is often the necessity to integrate custom dynamic models, which cannot be expressed using the underlying physics engine. As an example, consider the simulation of the low-level actuator dynamics, compliance, and delays in legged robotics [5], that is often critical in reinforcement learning approaches to robot control [6,7]. The obvious solution is co-simulation, i.e. combining the generic robotics simulator and its physics engine with a custom dynamic model and solver. The FMI standard has been developed exactly for this use-case, and the *gazebo-fmi plugins* [8] provide an easy-to-use integration of FMUs into the Gazebo simulator.

For example, the gazebo-fmi plugins allow co-simulating an FMU that models an electrical actuator of the iCub humanoid robot [9], including the actuator's back electromotive force (EMF) and the rotational inertia, directly with Gazebo.

The goals of this chapter are three-fold:

1. to teach the basics of the FMI by giving a comprehensive introduction to the FMI standard and corresponding tools,
2. to overcome any entry hurdle in the use of FMI technology with ROS by providing hands-on tutorials and documentations on the fmi_adapter package and the gazebo-fmi plugins, and

3. to inspire for practical use of the FMI in ROS-based systems and Gazebo-based simulations by reporting on advanced applications from our labs.

The remainder of this chapter is structured along those goals. Section 2 gives an overview to the goals and history of the FMI standard, before describing the contents of an FMU and related tools. Section 3 explains all steps to setup the fmi_adapter package on ROS or ROS 2 and to run the examples provided by the fmi_adapter_examples package. Then, details on the interface and the implementation are given. Analogously, Section 4 describes all necessary steps to build or install the gazebo-fmi plugins and to run a first demo application before explaining the interface and selected implementation details. Thereafter, Section 5 reports on the use of the fmi_adapter in the mentioned autonomous intra-logistics vehicle and on the implementation of the electrical actuator model based on gazebo-fmi. The chapter concludes in Section 6 with final remarks.

2 Overview to FMI

A crucial task in the development of robotic systems and software intensive systems in general, is the validation of the overall system behavior including software and hardware. This implies to shift the scope of simulation from components to systems and from single technical domains to multi-physical simulations with interfaces to software components.

As different simulation tools may be best suited for a particular task and varying preferences between departments, companies, and organizations exist, it is crucial to be able to exchange and to co-simulate models from different simulation tools. In this context, model exchange refers to the case of integrating an encapsulated model from one into another simulation environment, while co-simulation refers to the coupling of simulation models together with their individual numerical solver using a master algorithm and accepting a certain numerical error due to the coupling. This demand led to the development of the FMI standard.

In this section, we first give an introduction to the history of FMI, followed by a technical overview. Then, we present important FMI tools, before we discuss limitations. The section concludes with an outlook on further developments.

2.1 History of FMI

The FMI standard was developed in the publicly funded ITEA3 project MODELISAR from 2008 to 2011, which had been initiated by Daimler and others to create a tool-independent standard for model exchange and co-simulation [10]. Before FMI, one had to use or develop proprietary, bilateral tool interfaces, with a rapidly growing effort due to the large number of possible combinations. Alternatively, one had to use a proprietary de-facto standard like S-Functions from MATLAB/Simulink [11] within its limitations.

The name “Functional Mock-up Interface” is motivated from the “Digital Mock-up”, which is commonly used for the integration of CAD data from the

different parts and subsystems to an overall CAD assembly of a vehicle. This idea is transferred to the domain of system simulation in the sense of a complementary "functional" representation of the system.

The FMI 1.0 was released in 2010, see overview paper [12]. This first version of the standard was rapidly adopted by many simulation tools and was revised in 2014 by the improved version FMI 2.0 [13], as described in [14].

Since the MODELISAR project has ended, FMI is further developed as a project within the Modelica Association [15], a non-profit organization under Swedish law dedicated to the coordinated standardization in the field of systems engineering. FMI has been adopted very fast by industry, academia, and tool vendors [16].

Current developments within the FMI project focus on FMI 3.0, while in the publicly funded ITEA3 project EMPHYSIS [17] the extension of FMI for the deployment to embedded systems is being developed. More details are given in Subsection 2.5.

2.2 Functional Mock-up Units: Technical overview

A *Functional Mock-up Unit* (FMU) is the artifact exchanged between different simulation tools. Technically, an FMU is a ZIP file [18] containing the following elements:

- A model and interface description in form of an XML file named `modelDescription.xml`.
- A model representation with either
  - one or multiple platform specific binary representation(s) (e.g., Windows 64 bit or Linux 32 bit), or
  - a source code representation in C.
- An optional documentation folder.
- An optional resources folder containing, for instance, data files.

As FMI defines a model interface, there is always an *exporting tool* that generates the FMU and an *importing tool* that executes the FMU during a simulation run.

FMU kinds. In FMI 1.0 and FMI 2.0, two kinds of FMUs are distinguished as illustrated in Figure 1:

- *Model Exchange (ME) FMUs* describe a model, typically consisting of ordinary differential equations, where the numerical solver is not included in the model but must be provided by the importing tool.
- *Co-Simulation (CS) FMUs* contain both the model and a suitable solver. The communication with the importing tool takes place at discrete *communication time instances* only. The orchestration of multiple FMUs during the simulation process by passing the signals between the FMUs (possibly with filtering and extrapolation) and calling the individual FMUs in the right order at a certain time step, is the responsibility of a *master algorithm* provided by the importing tool. Please note that the FMI standard does not specify the master algorithm itself.

Since FMI 2.0, the standardized description of both kinds of FMUs have been combined. Hence, an FMU may contain both kinds of FMUs at once.

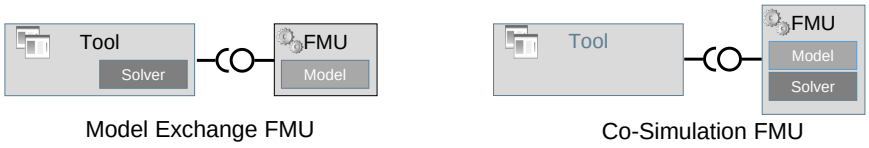

Fig. 1. Model Exchange vs. Co-Simulations FMUs.

Self-contained versus tool-wrapper FMUs. An FMU is from the basic idea a self-contained simulation model that does not have external dependencies, but is shipped with everything that is needed for execution (including necessary libraries). However, external dependencies are allowed and can be signaled by a flag in the XML file. This can in the extreme case lead to the situation that an FMU is just a wrapper that communicates with the instance of an installed simulation tool to simulate the model as depicted in Figure 2. Such an FMU is referred to as *tool wrapper FMU*. Especially for the deployment to real-time targets or software environments like ROS, self-contained FMUs are beneficial.

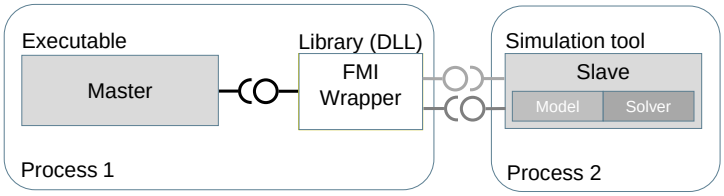

Fig. 2. Tool wrapper FMU.

C-API and calling sequence. The FMI standard defines an application programming interface (API) in the C programming language. In FMI 2.0 [13] it consists of 25 + 9 (ME) resp. 10 (CS) function prototypes. Examples for such function prototypes are

```
fmi2Component fmi2Instantiate(fmi2String instanceName, fmi2Type fmuType,
       fmi2String fmuGUID, fmi2String fmuResourceLocation,
       const fmi2CallbackFunctions* functions, fmi2Boolean visible,
       fmi2Boolean loggingOn);
```

for instantiation of an FMU,

```
fmi2Status fmi2DoStep(fmi2Component c,fmi2Real currentCommunicationPoint,
       fmi2Real communicationStepSize,
       fmi2Boolean noSetFMUStatePriorToCurrentPoint);
```

for performing a simulation step for a co-simulation FMU, and

```
fmi2Status fmi2GetReal(fmi2Component c, const fmi2ValueReference vr[],
       size_t nvr, fmi2Real value[]);
```

for getting the value of a variable form the FMU.

For each FMU kind, a state machine defines the allowed calling sequence of the functions. For example, for a Co-Simulation FMU this could look like the following pseudocode:

```
fmi2Component c = fmi2Instantiate(...)
fmi2SetupExperiment(c, ..., relTol, ...)
fmi2EnterInitializationMode(c)
fmi2SetReal(c, ...)
fmi2ExitInitializationMode(c)
...
(loop)
  fmi2SetReal(c, ...)
  fmi2DoStep(c, t=0.1, dt=0.1, ...)
  fmi2GetReal(c, ...)
(end loop)
...
fmi2Terminate(c)
fmi2FreeInstance(c)
```

Model description file. An FMU's model description file specifies all necessary interface information about the model required by the importing tool to properly integrate the model in terms of variables, parameters, structural dependencies and settings for model exchange and/or co-simulation. As an example, we consider the `modelDescription.xml` file from the `DampedPendulum.fmu`, which is provided by the fmi_adapter_examples package explained in the next section. This FMU implements a simple simulation model of a damped pendulum with configurable length, damping ratio and gravity constant:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription fmiVersion="2.0" modelName="dampedPendulum"
       description="..." guid="{192455ca-e123-595a-b9df-f95fe079b4ef}">

<CoSimulation modelIdentifier="dampedPendulum"
       canHandleVariableCommunicationStepSize="true">
  ... optional list of source files ...
</CoSimulation>

... some logging settings ...

<DefaultExperiment stepSize="0.002"/>
```

This first part of the XML refers to the FMI standard 2.0, provides the name of the FMU and a GUID (Globally Unique Identifier) string that is used for a unique instantiation and consistency check between the model description (XML) and the model representation (source code and/or binary). It also states that this FMU is a Co-Simulation FMU that can not only handle fixed but also variable communication time steps, as indicated by the capability flag `canHandleVariableCommunicationStepSize`. A crucial parameter for the simulation is the default step size, which is set to 2 ms in this example.

The next section of the xml file describes the model variables by specifying their names, attributes, and initial values:

```
<ModelVariables>
  <ScalarVariable name="a" valueReference="0" causality="output"
         variability="continuous" description="Angular␣displacement␣(in␣
             ↪ radians)">
    <Real/> <!-- index="1" -->
  </ScalarVariable>
  <ScalarVariable name="der(a)" valueReference="1"
         description="Angular␣velocity">
    <Real derivative="1"/> <!-- index="2" -->
  </ScalarVariable>
  <ScalarVariable name="der(a,2)" valueReference="2"
         description="Angular␣acceleration">
    <Real derivative="2"/> <!-- index="3" -->
  </ScalarVariable>
  <ScalarVariable name="l" valueReference="3" causality="parameter"
         variability="fixed" initial="exact"
         description="Length␣of␣the␣pendulum␣(in␣meter)">
    <Real start="1.0"/> <!-- index="4" -->
  </ScalarVariable>
  ... similar tags for damping ratio and gravity constant ...
</ModelVariables>
```

Finally, the structural dependencies between the known and unknown variables of the model are defined. The `Derivatives` refer those variables required to compute the corresponding state variables during a simulation step. The `InitialUnknowns` define the variables to be computed during the initialization. The variables are referred to by their `index`, which is a unique number within the XML file applied in increasing order starting from 1.

```
<ModelStructure>
  <Derivatives>
    <Unknown index="2"/>
    <Unknown index="3"/>
  </Derivatives>
  <InitialUnknowns>
    <Unknown index="2"/>
    <Unknown index="3"/>
  </InitialUnknowns>
</ModelStructure>
```

```
43
44 </fmiModelDescription>
```

In this example the two variables with index 2 and index 3, respectively `der(a)` and `der(a,2)`, are both declared as unknown derivative and as well as unknown initial variable.

2.3 FMI tools

There is a rapidly growing number of (simulation) tools that support the FMI standard. The official list can be found in the tools section of the FMI website [1]. In May 2019, 132 tools were listed. Many other tools (e.g., in-house simulation tools of companies) exist beyond the list.

Introspection tools and FMU cross check. Even though there is no tool available that can 100% guarantee the compliance with the FMI standard, there are very good tools available to support the users in performing compliance checks. One must distinguish between checking exported FMUs (i.e., whether the exported FMUs comply with the FMI standard) and checking the import and simulation of FMUs (i.e., whether the importing tools comply with the standard). In the first case, checking the export of FMUs, one can use the following two tools:

- *FMI Compliance Checker* [19]
- *FMPy* Python toolbox [20]

In addition, the *FMI Cross-Check* [21] measures the FMI maturity level of the importing and exporting tools (also using the FMPy Python toolbox) by validating uploaded FMUs. The results of such checks are displayed in the tools section of the FMI website [1]. Exporting tools provide FMUs together with reference solutions, while importing tools report on the results of the simulation. If a minimum set of FMUs is simulated successfully, the corresponding tool gets a green badge as displayed in Figure 3.
In the repository of the FMI Cross-Check, one can find hundreds of example FMUs. The *Test FMUs* by Dassault Systèmes [22] are of particular interest as they provide checks of importing tools inspired by the concept of *reference FMUs* developed in [23]. Furthermore, a long list of different FMI supporting tools is provided with:

- Commercial tools: Modelica-based simulation tools (e.g., Dymola, SimulationX, and MapleSim) as well as other simulation tools (e.g., MATLAB/ Simulink, AMESim and GT-Suite).
- Open-source tools: Simulation tools (e.g., OpenModelica), SDKs (e.g., FMU-SDK), test FMUs, and supporting tools for FMU generation and simulation for many programming languages like C, Java, and Python.

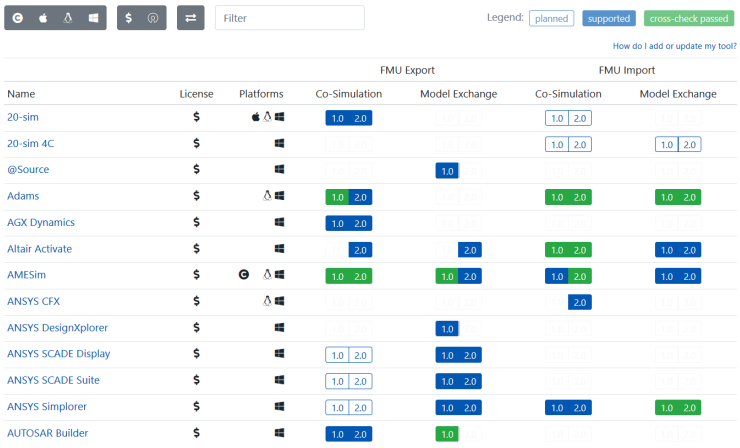

Fig. 3. FMI Cross-Check results displayed in the Tools section of the FMI website [1].

2.4 Limitations of FMI

Due to its design with the possibility for black-box model exchange, FMUs hide internal information of the model. When splitting overall systems into multiple FMUs, it is very important to carefully partition the system and to carefully select a proper co-simulation master algorithm to avoid numerically difficulties in case of tight couplings.

The currently released versions FMI do not support structured signal interfaces. Hence, interfaces consist of scalars only and there is no semantics of physical connectors, which can easily lead to a poor graphical readability of models with a large number of inputs and outputs. Also does FMI addresses "only" the interface problems, not the underlying numerical problems. Again, note that master algorithms are not part of the FMI standard.

Some of these issues will be improved by the next FMI release as described in the next subsection.

2.5 Future trends in the development and usage of FMI

The version 3.0 of the FMI standard is currently under development. It will provide the following new features, cf. FAQ section of the FMI website [1]:

- *Support for ports and icons:* Help the user to build consistent systems from FMUs and render the systems more intuitively with better representation of structured ports (e.g., buses and physical connectors) in the model description file.
- *Array variables:* Allow FMUs to communicate multi-dimensional variables of variable size using structural parameters.
- *Clocks and hybrid co-simulation:* Introduce clocks for synchronization of variable changes across FMUs. Allows co-simulation with event handling.

- *Binary data type:* Adds an opaque binary data type to FMU variables to allow, for instance, the efficient exchange of complex sensor data.
- *Intermediate variable access:* Allow access to input and output values between communication time instances to enable an enhanced numerical stability by advanced co-simulation master algorithms or to disclose relevant subsystem behavior for analysis purposes. (See [24] for more information on master algorithms. A master algorithm that could be implemented using this new FMI 3.0 feature is described in [25].)
- *Better support for source code FMUs:* Adding more information to the model description file to improve automatic import of source code FMUs.
- *Numeric variable types:* Adds 8, 16, 32 and 64-bit signed and unsigned integer and single-precision floating-point variable types to improve efficiency and type safety when importing or exporting models from the embedded, control and automotive domains.

While FMI was originally particularly developed for modeling and simulation on non-real-time systems, FMI is also successfully used on real-time systems like Hardware-in-the-Loop (HiL) systems. FMUs have been ported prototypically to embedded Electronic Control Units (ECUs), as demonstrated in [26]. This has inspired the standardization effort *FMI for Embedded Systems* (eFMI) within the ITEA 3 EMPHYSIS Project, see [27]. An additional approach of using FMI within control software is the usage of FMI within ROS-based applications, which is described in the next section.

3 The fmi_adapter package

The fmi_adapter is a ROS package for wrapping co-simulation FMUs into ROS nodes. It has been implemented for ROS 1 as well as for ROS 2 in the C++ programming language. Binary distributions are currently available for ROS Melodic and for ROS 2 Crystal, Dashing, and Eloquent.

The fmi_adapter package aims at providing a mapping between the most important functions of the FMI 2.0 Co-Simulation interface and corresponding ROS concepts and types as depicted in the following table:

FMI	**ROS**
input variable	subscriber
output variable	publisher
state variable	*no explicit counterpart*
parameter initialization	parameter server
simulation time	ROS clock
communication step-size	timer

By this mapping, FMUs can be integrated with ROS without the necessity to use FMI types in the ROS C++ code. Fmi_adapter is intended neither to implement the whole FMI 2.0 API nor to provide the rich set of introspection functions as,

for instance, FMI Library [28] or FMI4cpp [29]. For advanced use-cases with FMUs, ROS developers are referred to such libraries.

The fmi_adapter package allows for two types of usage named *node-based use* and *library-based use* in the following. Accordingly, it provides an executable ROS node as well as a shared library. In node-based use, the fmi_adapter node is called directly with a path to an FMU. The node creates a subscriber for each input variable of the FMU and a publisher for each output variable. Then, it runs – i.e., simulates – the FMU according to ROS time.

The library-based use gives much more control about the integration of this FMU in a ROS node. For this type of usage, the fmi_adapter package provides a C++ class `fmi_adapter::FMIAdapter`, which is instantiated with the path to an FMU and wraps this FMU. This allows translating complex ROS message types to the input types of the FMU and vice-versa for the output types. In addition, the class provides member functions to introspect the FMU. Finally, this type of usage allows for multiple FMUs inside a single node.

In this section, we first describe how to install or build the fmi_adapter package for ROS 1 and ROS 2, followed by instructions on how to setup a free modeling tool for creating own FMUs. Then, we explain how to run the sample applications provided by the fmi_adapter_examples package and how to create and run an own FMU in node-based use. Thereafter, we give details on library-based use and the interface provided by the `FMIAdapter` class. The section is concluded with a description of the architecture of the fmi_adapter package and selected implementation details.

3.1 ROS environment configuration

The fmi_adapter package and the associated fmi_adapter_examples package are shipped with all dependencies. Therefore, they can be installed or built directly on current vanilla ROS 1 or ROS 2 installations. There is one important limitation for the fmi_adapter_examples package: The two provided sample FMUs (`DampedPendulum.fmu` and `TransportDelay.fmu`) have been built for AMD64 architecture only. We tested them with Ubuntu 16.04 and Ubuntu 18.04 successfully.

Binary installation. On Debian, Ubuntu, and related Linux distributions, the fmi_adapter package for ROS Melodic and the associated example package can be installed using apt with the command:

```
sudo apt install ros-melodic-fmi-adapter
sudo apt install ros-melodic-fmi-adapter-examples
```

Similarly, the binary packages for ROS 2 Dashing can be installed by:

```
sudo apt install ros-dashing-fmi-adapter
sudo apt install ros-dashing-fmi-adapter-examples
```

Building from source. To build the fmi_adapter package with its examples for ROS 1 from source, we recommend using the usual Catkin Command Line Tools [30]. This also allows building the package for older ROS versions such as Kinetic and Lunar. Detailed instructions are given in the `README.md` of the corresponding repository [3]. Please note that during the build, the *FMILibrary* from JModelica.org [28] is being downloaded and built as a third-party library using CMake's `externalproject_add` command [31].

Similarly, the ROS 2 version of the fmi_adapter package can be built from source using the usual colcon tool [32]. Further instructions are given in the `README.md` of the repository [4]. Unlike in the ROS 1 version, the FMILibrary is not downloaded and built by the build file of the fmi_adapter package itself, but by the build file of a separate vendor package named *fmilibrary_vendor*.

Installation of modeling tools. The fmi_adapter_examples package provides two FMUs (`DampedPendulum.fmu` and `TransportDelay.fmu`) as well as a Modelica file (`DampedPendulum.mo`), from which a third one can be created.

Several tools may be used for the latter, as explained in Section 2. In the next subsection, we explain this procedure by the example of OpenModelica. Please navigate to the Download page of the OpenModelica website [33] for detailed instructions on how to install it from pre-built Debian packages. The website also provides instructions for installations under Windows and macOS as well as for building from source.

The two prebuilt FMUs `DampedPendulum.fmu` and `TransportDelay.fmu` distributed in the fmi_adapter_examples package have been implemented using the FMU SDK by QTronic. To follow the next subsections, there is no need to install this SDK. Nevertheless, if you are interested, this SDK can be downloaded from the QTronic website as a ZIP file [34]. Once unzipped, several further sample FMU projects can be found at `fmu20srcmodels`. They can be build using the `build_fmu` script in the same folder.

3.2 First applications by node-based use

As a first example, we run the `DampedPendulum.fmu` from the fmi_adapter_examples package located in the `share` folder. Thereafter, we run a more advanced example with the two provided FMUs `DampedPendulum.fmu` and `TransportDelay.fmu` before we simulate a damped pendulum FMU created from the `DampedPendulum.mo` file.

Simple damped pendulum example on ROS 1. The examples package provides a launch file to run the prebuilt `DampedPendulum.fmu`. Simply call

```
roslaunch fmi_adapter_examples simple_damped_pendulum.launch
```

from the command line. This launches two nodes: First, the fmi_adapter node with the `DampedPendulum.fmu` as value for the `fmu_path` parameter. Second, rqt_plot with argument `/fmi_adapter_node/a`. This last argument makes rqt_plot to

visualize the data published by the fmi_adapter node on the topic `/fmi_adapter_node/a`, which is the angle of the simulated pendulum. Figure 4 shows the output by rqt_plot.

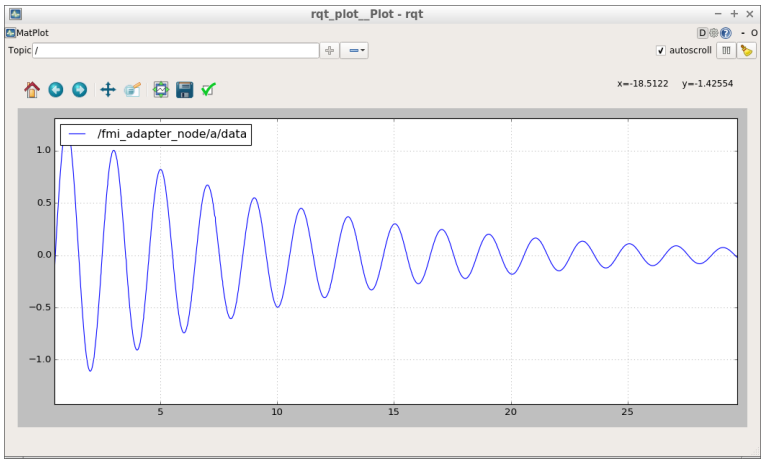

Fig. 4. Output by rqt_plot for the angle of the simple damped pendulum.

The launch file `simple_damped_pendulum.launch` consists of the following six lines:

```
1 <launch>
2   <include file="$(find␣fmi_adapter)/launch/fmi_adapter_node.launch">
3     <arg name="fmu_path" value="$(find␣fmi_adapter_examples)/
        ↪ DampedPendulum.fmu" />
4   </include>
5   <node name="rqt_plot" pkg="rqt_plot" type="rqt_plot" args="/
      ↪ fmi_adapter_node/a" />
6 </launch>
```

The lines 2 to 4 start the fmi_adapter node by calling a tiny, generic launch file from the fmi_adapter package with the path to the `DampedPendulum.fmu`. It is also possible to refer directly to the fmi_adapter node by replacing the lines 2 to 4 with:

```
2   <node name="fmi_adapter_node" pkg="fmi_adapter" type="node">
3     <param name="fmu_path" value="$(find␣fmi_adapter_examples)/
        ↪ DampedPendulum.fmu" />
4   </node>
```

The name of the topic for the pendulum's angle is composed from the name of the fmi_adapter node (which is set to `fmi_adapter_node` in line 2) and the name `a` of the output variable for the pendulum's amplitude (the angle in radians) specified in the `modelDescription.xml` file of `DampedPendulum.fmu`.

Of course, the fmi_adapter node may be also invoked directly using `rosrun`. For example, assuming a standard installation of the fmi_adapter_examples package, the damped pendulum simulation can be started by:

```
rosrun fmi_adapter node _fmu_path:=/opt/ros/melodic/share/
    ↪ fmi_adapter_examples/DampedPendulum.fmu
```

Simple damped pendulum example on ROS 2. On ROS 2, the simple damped pendulum example can be invoked by

```
ros2 launch fmi_adapter_examples simple_damped_pendulum.launch.py
```

analogously to ROS 1. The output may be retrieved on a second command prompt by

```
ros2 topic echo /a
```

which should give an output like

```
data: -1.0744451949191
data: -1.067165037971358
data: -1.0571404911146913
```

and so on. Note that the node name is not prepended to the topic name as in ROS 1.

As in the ROS 1 version, the fmi_adapter package provides a generic launch file, which is called from `simple_damped_pendulum.launch.py` with the `DampedPendulum.fmu` as value for the `fmu_path` parameter. The following listing shows this Python-based launch code.

```
1  ... some import statements ...
2
3  def generate_launch_description():
4      fmu_path = (ament_index_python.packages.get_package_share_directory('
           ↪ fmi_adapter_examples') + '/share/DampedPendulum.fmu')
5
6      fmi_adapter_description = launch.actions.IncludeLaunchDescription(
7          launch.launch_description_sources.PythonLaunchDescriptionSource(
8              ament_index_python.packages.get_package_share_directory(
9                  'fmi_adapter') + '/launch/fmi_adapter_node.launch.py'),
10         launch_arguments={'fmu_path': fmu_path}.items())
11
12     description = launch.LaunchDescription()
13     description.add_action(fmi_adapter_description)
14
15 return description
```

The path to the FMU file is determined in line 4. The lines 7 to 9 load the launch file from the fmi_adapter package. For ROS 2, the fmi_adapter node has been implemented as a *ManagedNode* [35], which provides a standardized runtime lifecycle. The generic `fmi_adapter_node.launch.py` file automatically switches the node from the initial state *unconfigured* to *inactive* and then to *active*.

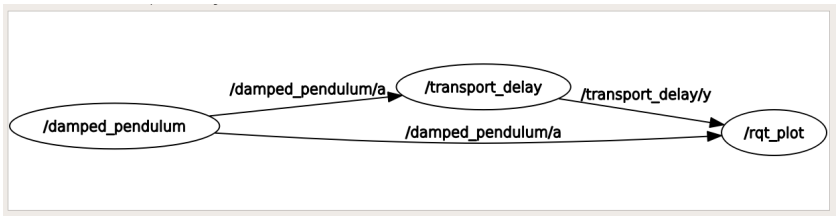

Fig. 5. Screenshot of rqt_graph for damped pendulum with transport delay example.

Damped pendulum with transport delay example. The examples package provides a second launch file `damped_pendulum_with_transport_delay.launch`, which creates two instances of the fmi_adapter node with the damped pendulum FMU and a simple transport delay FMU. The latter gets the output of the pendulum FMU as an input and delays it by 2.33 s as depicted in Figure 5. On ROS 1, call

```
roslaunch fmi_adapter_examples damped_pendulum_with_transport_delay.
    ↪ launch
```

to launch this example. As with the simple damped pendulum, rqt_plot is opened and visualizes both, the output of the damped pendulum FMU and the output after the transport delay.

On ROS 2, the corresponding command is

```
ros2 launch fmi_adapter_examples damped_pendulum_with_transport_delay.
    ↪ launch.py
```

but does not provide a visualization. Here, the launch file does not call the generic `fmi_adapter_node.launch.py` launch file from the fmi_adapter package but directly starts two instances of the fmi_adapter node. In doing so, the lifecycles of the two instances are switched synchronously from unconfigured to inactive and then to active.

Running your own damped pendulum FMU. The fmi_adapter_examples package contains a Modelica file of a damped pendulum model created with OpenModelica. If you performed a binary installation of the package, you may open the file by

```
OMEdit /opt/ros/melodic/share/fmi_adapter_examples/DampedPendulum.mo
```

or rather

```
OMEdit /opt/ros/dashing/share/fmi_adapter_examples/share/DampedPendulum.
    ↪ mo
```

in the OpenModelica Connection Editor. To export an FMU from this file, perform the following steps (cf. also Figure 6):

– Click on the DampedPendulum model in the project tree on the left.

– Navigate to Tools → Options → FMI and ensure that Version=2.0, Type=Co-Simulation and Platforms=Dynamic is selected.
– Then click File → Export → FMU.[3]
– The path of the resulting FMU file is shown in the message browser at the bottom of the window, typically `/tmp/OpenModelica_[user]/OMEdit/DampedPendulum.fmu`, where `[user]` is your user name.

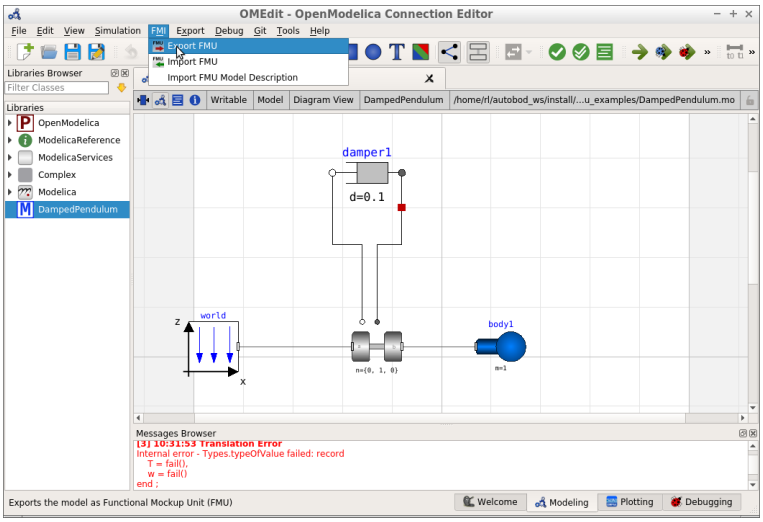

Fig. 6. Screenshot of OpenModelica Connection Editor with DampedPendulum.mo model.

Once the FMU has been exported, you may use the generic launch file of the fmi_adapter package to launch it by

```
roslaunch fmi_adapter fmi_adapter_node.launch fmu_path:=/tmp/OpenModelica
    ↪ \_[user]/OMEdit/DampedPendulum/DampedPendulum.fmu
```

or, alternatively, using rosrun, by

```
rosrun fmi_adapter node _fmu_path:=/tmp/OpenModelica\_[user]/OMEdit/
    ↪ DampedPendulum/DampedPendulum.fmu
```

The angle of the pendulum is published on the topic `/fmi_adapter_node/revolute1_angle`.

ROS 2 does not support to pass arguments directly to launch files. For `ros2 run`, the parameters have to be passed in a YAML file. Therefore, create a file `my_params.yaml` as follows:

[3] In versions before 1.14, navigate to the menu FMI and click Export FMU.

```
fmi_adapter_node:
    ros__parameters:
        fmu_path: '/tmp/OpenModelica\_[user]/OMEdit/DampedPendulum/
            ↪ DampedPendulum.fmu'
```

Then, start the fmi_adapter node by

```
ros2 run fmi_adapter fmi_adapter_node __params:=my_params.yaml
```

Nothing will happen at this point yet, since the node will start in the lifecycle state *unconfigured*. To change the state to *active* call

```
ros2 lifecycle set /fmi_adapter_node configure
ros2 lifecycle set /fmi_adapter_node activate
```

from a second terminal. Now, the pendulum's angle is published on the topic `/revolute1_angle`.

3.3 Interface description and library-based use

Next, we first explain the whole interface of the fmi_adapter node – i.e., for node-based use – before we provide details on the library-based use.

Node-based use. From the previous examples, you already learned most interface elements in node-based use. In detail, these elements are:

- *fmu_path:* The most important interface element is the mandatory parameter `fmu_path`, which takes the path to the FMU to simulate in this node.
- *Subscribers for input variables:* The fmi_adapter node reads the model description of the FMU and creates a subscriber for each input variable of the FMU. The subscribers are named as the corresponding input variables, where special characters that are not allowed in ROS names are replaced with an underscore. Currently, the fmi_adapter supports 64-bit floating-point variables only – and creates subscribers of type `std_msgs/Float64` accordingly.
- *Publishers for output variables:* Analogously to subscribers, a publisher is created for each output variable of the FMU.
- *Parameters for FMU parameters and start values:* The fmi_adapter node also reads any FMU parameter and variable start values from potential ROS parameters. For this purpose, it queries the ROS parameter server for entries that match the name of a variable or parameter of the FMU (again replacing special characters by underscore). For each such entry being found, the value is read from the ROS parameter server and the correspondent FMU parameter or variable is initialized with that value.
- *Parameters for solver step-size and update period:* The optional parameter `step_size` specifies the duration in seconds to be used by the FMU's solver as solving step-size. If this parameter is given, it overwrites the step-size specified in the FMU's model description. Note that the ROS parameter `step_size` supports fixed values only, whereas the FMU's model description may also specify a variable step-size. The ROS parameter `update_period`

controls the frequency in which the output variables of the FMU are read and published on the corresponding topics. By choosing the update period higher than the step-size, the output rate can be reduced without affecting the solving precision.

Library-based use. In library-based use, the class `fmi_adapter::FMIAdapter` is instantiated with the path to an FMU, to wrap this FMU for further use with ROS data types and other ROS concepts. In addition to the FMU path, the constructor also takes an optional argument `stepSize`, which may be used to overwrite the step-size for the FMU's solver specified in the FMU's model description.

As explain above, the fmi_adapter package aims to map the major FMI concepts only and not to implement and map the whole FMI 2.0 API. The first simplification being made by the `FMIAdapter` class are the runtime states of an FMU. The FMI standard defines a set of runtime states for FMUs by a hierarchical state-machine [13, p. 103]. The major such states for co-simulation are *instantiated*, *InitializationMode*, *slaveInitialized*, and *terminated*. The `FMIAdapter` distinguishes between two states only: *InitializationMode*, in which parameters and initial values of variables can be set, and *slaveInitialized*, in which the actual simulation steps may be computed. The other two states are switched automatically in the constructor and destructor, respectively.

The Boolean member function `FMIAdapter::isInInitializationMode()` allows querying whether the FMU is still in initialization mode or has already entered the actual simulation state.

During initialization mode, the FMIAdapter provides the function `setInitialValue(..)` to set initial variable values as well as parameters by means of their name. The names of all variables and parameters can be obtained by the functions `getAllVariableNames()`, `getInputVariableNames()`, `getOutputVariableNames()`, and `getParameterNames()`. The parameters and variables may be also initialized from the ROS parameter server using the function `initializeFromROSParameters()`. This function queries the ROS parameter server for entries that match the name of a variable or parameter of the FMU as described above in node-based use. On ROS 2, the function `declareFromROSParameters()` has be to called first, to make potential parameter values set during launch available to the node.[4]

An `FMIAdapter` instance also allows querying whether the FMU's solver supports a variable communication step-size (by `canHandleVariableCommunicationStepSize()`) and to obtain the default step-size (by `getDefaultExperimentStep()`).

By invoking the function `exitInitializationMode(..)`, the `FMIAdapter` instance is set to *slaveInitialized* state. This function takes a ROS timestamp as argument. This timestamp defines the offset between the FMU's solver time (which typically starts at zero) and the ROS time (which is either the system

[4] The `declare_parameter` function was introduced in rclcpp in ROS 2 Dashing Diademata.

time or a simulated time). This offset is used as clock translation between ROS and the FMU in all further function calls.

Values for the input variables can be set programmatically by the following function:

```
FMIAdapter::setInputValue(variable-name, timestamp, value)
```

The second argument specifies the ROS timestamp from which on the given value is valid. For each input variable, the `FMIAdapter` stores the timestamp-value pairs as an input trajectory for the FMU. Thus, `setInputValue` may be called arbitrarily in advance of solving. By the constructor flag `interpolateInput`, it can be controlled whether the input trajectory is interpolated linearly between the given timestamp-value pairs or whether it is considered as a step function (i.e., piecewise constant).

The FMU simulation can be advanced by a single step by `doStep()`, which returns the ROS timestamp until that the FMU solver has advanced to. This function may be also called with user-defined step-size value instead of the default step-size from the `FMIAdapter` constructor or the FMU. The function `doStepsUntil(..)` advances the simulation to a given ROS timestamp (modulo step-size). During the computation, the solver reads the input values from the input trajectories according to their timestamps. This feature of input trajectories allows translating between different sampling/sensor rates easily.

After each simulation step, `getOutputValue(..)` allows retrieving the current value of an output variable by means of its name.

Code example for library-based use. As an example for library-based use, we consider at node that runs both sample FMUs (damped pendulum and transport delay) together synchronously. The following code snippets show the node's main function – here for ROS 1, a port to ROS 2 is straight-forward.

As a first step, the paths of the two FMUs are read from the parameter server, where exception handling in case of missing parameter values is omitted here. Then, the `FMIAdapter` class is instantiated for each FMUs using the same step-size of 20 ms.

```
#include <ros/ros.h>
#include <std_msgs/Float64.h>
#include <fmi_adapter/FMIAdapter.h>

int main(int argc, char** argv) {
  ros::init(argc, argv, "fmi_adapter_examples_integrated_node");
  ros::NodeHandle n("~");

  std::string pendulumFmuPath;
  n.getParam("pendulum_fmu_path", pendulumFmuPath);
  std::string delayFmuPath;
  n.getParam("delay_fmu_path", delayFmuPath);

  ros::Duration stepSize(0.02);
```

```
15  fmi_adapter::FMIAdapter pendulumAdapter(pendulumFmuPath, stepSize);
16  fmi_adapter::FMIAdapter delayAdapter(delayFmuPath, stepSize);
```

Next, some parameters of the FMUs are set manually. Here, we set the damping ratio d of the pendulum to 0.001 and the length of pendulum to 25 m. The delay is set to 5.5 s. Furthermore, the initial angle of the pendulum a is set to 1.3 radians.

```
17  pendulumAdapter.setInitialValue("d", 0.001);
18  pendulumAdapter.setInitialValue("l", 2.0);
19  delayAdapter.setInitialValue("d", 5.5);
20
21  pendulumAdapter.setInitialValue("a", 1.3);
```

By calling `exitInitializationMode(..)` with the same timestamp, both instances are put to the *slaveInitialized* state with exactly the same offset between the solver times and ROS time.

```
22  ros::Time now = ros::Time::now();
23  pendulumAdapter.exitInitializationMode(now);
24  delayAdapter.exitInitializationMode(now);
```

In this example, the simulation steps of the FMUs are triggered by a timer, but they could be triggered likewise by a subscriber or service. Both solvers are advanced in a synchronized fashion in the timer's callback function (a lambda expression). Due to the dependency between the two FMUs, first the damped pendulum FMU is advanced, then the transport delay. In between, the pendulum's current angle is obtained from the damped pendulum FMU and fed into the transport delay FMU. As a last step, the callback publishes the current angle as well as the delayed angle. As usual, the actual computation starts with `ros::spin()`.

```
25  ros::Publisher anglePub = n.advertise<std_msgs::Float64>("angle",1000);
26  ros::Publisher delayedAnglePub = n.advertise<std_msgs::Float64>("
        ↪ delayed_angle", 1000);
27
28  ros::Timer timer = n.createTimer(stepSize, [&](const ros::TimerEvent&
        ↪ event) {
29    pendulumAdapter.doStepsUntil(event.current_expected);
30    double angle = pendulumAdapter.getOutputValue("a");
31    delayAdapter.setInputValue("x", event.current_expected, angle);
32    delayAdapter.doStepsUntil(event.current_expected);
33    double delayedAngle = delayAdapter.getOutputValue("y");
34    std_msgs::Float64 angleMsg;
35    angleMsg.data = angle;
36    anglePub.publish(angleMsg);
37    std_msgs::Float64 delayedAngleMsg;
38    delayedAngleMsg.data = delayedAngle;
39    delayedAnglePub.publish(delayedAngleMsg);
40  });
41
```

```
42   ros::spin();
43 }
```

3.4 Architecture and implementation details

Internally, fmi_adapter is based on the FMI Library [28], which is an implementation of the FMI 1.0 and 2.0 standard in the C programming language. Figures 7 and 8 illustrate the architectural layering for node-based use and library-based use, respectively. From the first figure, it can be seen that the generic fmi_adapter_node is based directly on the `FMIAdapter` class and makes intensive use of introspection functions to determine the names of the subscribers, publishers, and parameters to create.

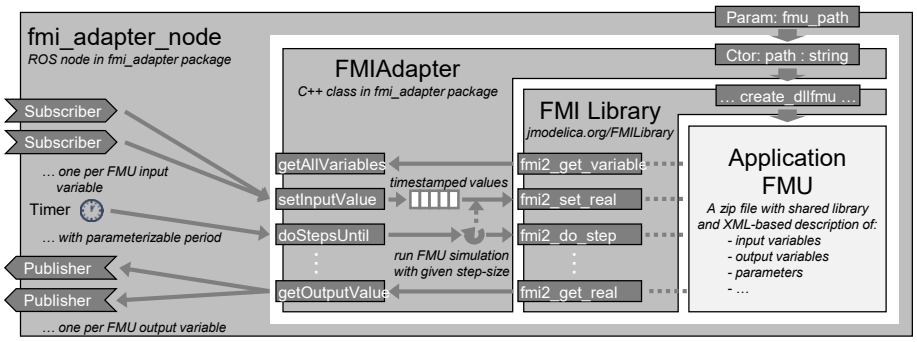

Fig. 7. Architecture diagram for node-based use

Use of FMI Library. The fmi_adapter package hides the FMI Library largely from the ROS developer. All FMI-specific types are translated to ROS types as explained in the introduction to this section. The `FMIAdapter` header file does not include any header from the FMI Library. Nevertheless, it provides access to the variable type `fmi2_import_variable_t` of the FMI Library, but only as a pointer to forward-declaration. The use of this pointer type allows avoiding repeated name resolution of the variables for computational efficiency.

The FMI Library sources are not included in the fmi_adapter package but downloaded on the fly from `jmodelica.org` (using CMake's `externalproject_add` function) when building it. For ROS 2, a separate vendor package named *fmilibrary_vendor* has been introduced for this purpose.

Time offset. As explained above, an `FMIAdapter` instance starts the FMU simulation from time zero on and stores the offset to ROS time to translate between the two clock representations. In fact, an FMU may also support other start

times. We nevertheless decided for the explicit offset and the start time zero since the FMI 2.0 standard represents time by a floating-point type, which implies a significant loss of precision when using the system clock for the FMU time.

Special characters in names. The FMI standard supports various characters in the variable names that are not allowed in parameter or topic names in ROS and ROS 2. Therefore, the static function `FMIAdapter::rosifyName(..)` has been introduced to replace these characters by underscores. An `FMIAdapter` instance automatically applies this function to all FMU variable and parameter names. The same underscores have to be inserted when referring to such names in source code, launch files, or parameter YAML files.

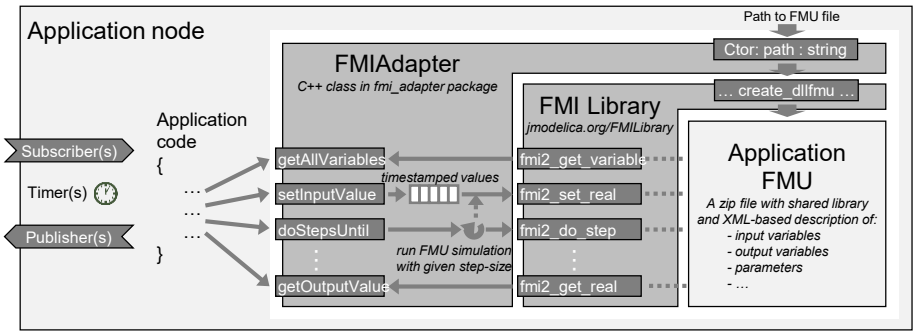

Fig. 8. Architecture diagram for library-based use

Parameter declaration in ROS 2. In ROS 2 Dashing, a parameter declaration mechanism was introduced in rclcpp. Amongst others, this mechanism allows to detect typos in parameter names and type errors at runtime. When using the automatic initialization of FMU parameters and start values from ROS parameters by the `initializeFromROSParameters()`, this requires declaring those ROS parameters first. Since their names depend on the FMU, the function `declareROSParameters(..)` has been introduced in the `FMIAdapter` class. This function should be called once before `initializeFromROSParameters()`.

Once a parameter is declared with a specific type by rclcpp's `declare_parameter(..)` function, it cannot be distinguished from the default value whether this value has been set explicitly (by launch or other nodes) or not. This, however, is necessary since an FMU may provide an internal mechanism to compute the start value for a variable without a specific default value. Therefore, `FMIAdapter::declareROSParameters(..)` declares all parameters with the data type `PARAMETER_NOT_SET`. By means of this data type, it can later decide on the parameter values that were given explicitly, to pass only those ones to the FMU.

Supported FMU data types. The FMI 2.0 standard defines six basic data types (`fmi2Real`, `fmi2Integer`, `fmi2Boolean`, `fmi2Char`, `fmi2String`, and `fmi2Byte`), which are mapped to the corresponding C types in the FMI Library. Currently, the fmi_adapter node and the `FMIAdapter` class support `fmi2Real` only, which is the type double in C/C++. We deem this sufficient for many practical applications, but other types could be added easily.

4 The gazebo-fmi plugins

Gazebo is a 3D simulation environment for robotics, which is integrated in ROS binary distributions [36]. From the point of view of physics simulation, Gazebo implements an abstraction interface over several rigid body dynamics physics engines: a modified version of Open Dynamics Engine (ODE), Bullet, Simbody, and DART.

For several applications, the dynamics captured by this kind of simulations are not enough to achieve the necessary level of simulation accuracy with respect to the real system. Examples of these applications are quadruped control, both model-based [5] and reinforcement learning based [6,7]. Other examples include the aerodynamics of a quadcopter and the dynamics of an underwater autonomous vehicle [37]. In most of these cases, the authors addressed the problem by augmenting the Gazebo simulator (or a simulator with similar characteristics) with manually coded C++ plugins that co-simulate the necessary dynamics together with the usual multibody simulation of Gazebo. This approach requires a lot of C++ expertise and profound knowledge of the API and architecture of Gazebo, whereas it prevents to easily exploit any existing modeling environment. A different approach was used in the case of [38] where the dynamics model used in the Gazebo plugin was not manually programmed in C++, but was rather generated from a Matlab model.

Gazebo-fmi is a set of plugins for the Gazebo simulator that permit to import co-simulation FMUs in Gazebo to use them to perform co-simulation with the main Gazebo multibody simulation. The plugins integrate the FMUs in the main Gazebo physics thread. Therefore, in co-simulation terminology, the Gazebo physics thread acts as *co-simulation master* while the loaded FMUs with their solvers and models are *co-simulation slaves*. This permits to use any existing software that supports FMI export for modeling the behavior of interest, and just integrate it in Gazebo by writing an appropriate configuration file, without the need of writing any custom C++ code. In particular, as of version 0.2, gazebo-fmi supports integration of FMUs in Gazebo using two plugins: `gazebo-fmi-actuator` to simulate actuator dynamics and `gazebo-fmi-single-body-fluid-dynamics` to simulate the approximate interaction between a single body and a surrounding fluid, such as air or water.

The *actuator dynamics* refers to the relation between the input to the physical motor used in the mechanical model (voltage in the case of electric motors, and other quantities for pneumatic and hydraulic motors), the position, velocity, and acceleration of output shaft of the transmission connected to the motor and the

actual effort transmitted by the output shaft on the mechanical structure. Out of the box, the physics engines used in Gazebo consider just the transmission output effort as an input to the simulation, and they only simulate basic joint friction behavior. By using gazebo-fmi's actuator plugin, it is possible instead to simulate the actuator dynamics such as reflected inertias, velocity/torques curves and series elastic actuators using an FMU.

The actuator plugin can be used transparently with the joint of any Gazebo model, regardless of how the joint is controlled: using the Gazebo internal PID controller or using ROS specific control plugin such as gazebo_ros_pkgs. This is possible by exploiting the Gazebo event system and by ensuring that the plugin is always running after the Gazebo PID update, but before the main Gazebo physics update, as explained in Subsection 4.4.

The *single body fluid dynamics* refers to the approximate models that are used for real-time simulations of vehicles that move in the air or in water: when a complete Computational Fluid Dynamics (CFD) simulation of the body-fluid interaction is not feasible, simplified models that just capture lift and drag for aerodynamics or drag, buoyancy, and added masses for hydrodynamics are used. By using gazebo-fmi's single body fluid dynamics plugin, it is possible to simulate these behaviors using an FMU.

This chapter describes the version 0.2 of gazebo-fmi. For updated information, please refer to the gazebo-fmi repository [8].

4.1 ROS environment configuration

Building from source. Although it can be used in a ROS environment, gazebo-fmi is a C++ project that does not depend directly on any ROS or ROS 2 library. For this reason, as of version 0.2, the recommended way of using it is building it from source. Gazebo-fmi's *build system* is based on plain CMake, so it can be built as a standalone CMake project or using any *build tool* that support plain CMake, such as catkin-tools [30] or colcon [32]. For a definition of *build system* and *build tool*, see [39].

There are two main dependencies for the gazebo-fmi plugins. The first one is the package containing development libraries of Gazebo, usually available in Ubuntu in a deb package called `libgazebo[ver]-dev`, where `[ver]` is the major version of Gazebo, either `7`, `8`, `9`, or `10` for gazebo-fmi v0.2 [40]. The second dependency is the FMI Library from JModelica.org [28]. This library is not included in the official Ubuntu software repository, but can be downloaded and built as a third-party library using CMake's `FetchContent` module [41] if the `USE_SYSTEM_FMILIBRARY` CMake option is set to `OFF`.

On a system in which Gazebo and its development libraries are installed, compiling the library without any build tool can be done with the following steps

```
git clone https://github.com/robotology/gazebo-fmi
cd gazebo-fmi
mkdir build && cd build
```

```
cmake CMAKE_INSTALL_PREFIX=<install\_prefix> ..
make
make install
```

where `<prefix>` is the installation prefix for the library. Once installed, the directory `<install_prefix>/lib` should be added to the Gazebo environment variable `GAZEBO_PLUGIN_PATH` so that the gazebo-fmi plugins can be found by Gazebo, and any directory that contains FMUs that should be loaded by the Gazebo should be added to the `GAZEBO_RESOURCE_PATH` environment variable.

Further instructions are given in the `README.md` of the gazebo-fmi repository [8].

Installation of modeling tools. Similarly to what discussed in Subsection 3.1, a complete discussion of all the modeling tools (cf. tools section of the FMI website [1]) that support FMI and can be used with the gazebo-fmi plugins is beyond the scope of this chapter.

Among all the different existing tools, for continuous integration and provide examples usage, we use the open source Modelica environment OpenModelica. Please navigate to the Download page of the OpenModelica website [33] for detailed instructions on how to install it from pre-built Debian packages. The OpenModelica website also provides instructions for installations under Windows and macOS as well as for building from source.

4.2 First applications

In this section, we will discuss basic examples shipped with gazebo-fmi. To run these examples, it is not necessary to configure the `GAZEBO_PLUGIN_PATH` and `GAZEBO_RESOURCE_PATH` environment variable of Gazebo manually, but it is sufficient to source the `setup-examples.sh` script, which will set all relevant environment variables, by:

```
source <install_prefix>/share/gazebo-fmi/examples/setup-examples.sh
```

As first example, we run a Gazebo world that contains two pendulum models, available at `gazebo-fmi/plugins/actuator/examples/damped_pendulum/damped_pendulum.world`. This world contains two pendulum models whose geometric and inertial parameters are identical to the Modelica pendulum used in Subsection 3.2. Discussion on the details of how the pendulum are modeled using the Simulation Description Format (SDF) of Gazebo are out of scope here, the interested readers can refer to Gazebo tutorials [42], such as the simple_gripper tutorial.

The difference between the models is in their joint connecting the pendulum to the fixed base. In the first model `pendulum_damped_via_sdf`, the joint is defined as

```
<joint name="joint" type="revolute">
  <parent>world</parent>
  <child>pendulum</child>
  <axis>
```

```
    <xyz>0.0 1.0 0.0</xyz>
    <dynamics>
      <damping>0.1</damping>
      <friction>0</friction>
      <spring_reference>0</spring_reference>
      <spring_stiffness>0</spring_stiffness>
    </dynamics>
  </axis>
</joint>
```

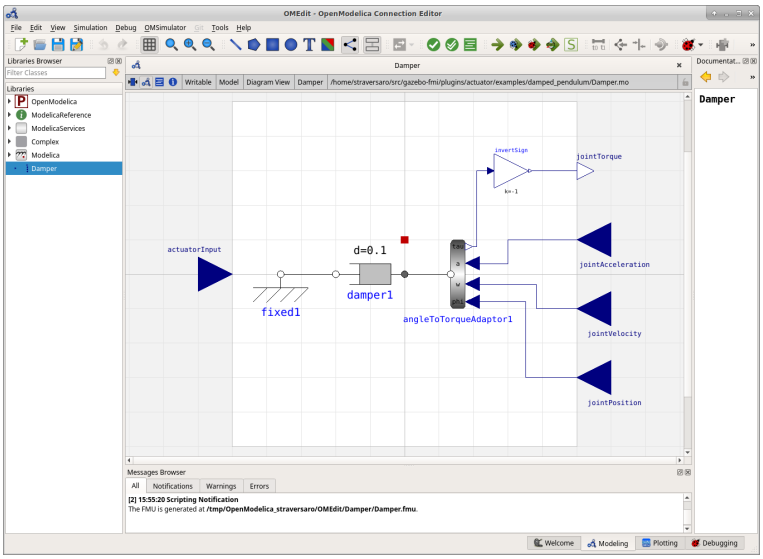

Fig. 9. Screenshot of OpenModelica Connection Editor with `Damper.mo` model.

In contrast, in the second model `pendulum_damped_via_fmi`, the joint is defined as

```
<joint name="joint" type="revolute">
  <parent>world</parent>
  <child>pendulum</child>
  <axis>
    <xyz>0.0 1.0 0.0</xyz>
    <dynamics>
      <damping>0.0</damping>
      <friction>0.0</friction>
      <spring_reference>0</spring_reference>
      <spring_stiffness>0</spring_stiffness>
    </dynamics>
  </axis>
</joint>
<plugin name="damper" filename="libFMIActuatorPlugin.so">
```

```
15    <actuator>
16      <name>joint_damper</name>
17      <joint>joint</joint>
18      <fmu>Damper.fmu</fmu>
19    </actuator>
20  </plugin>
```

The joint of the first model has a damping of 0.1, specified via SDF. The joint of the second model has a SDF damping of 0.0, but references a custom FMI actuator model named `Damper.fmu`. This FMU file should be generated from the `gazebo-fmi/plugins/actuator/examples/damped_pendulum/Damper.mo` Modelica model shown in Figure 9. This model has been obtained by taking the damper used in Subsection 3.2 and copying it in a "shell" model that exposes the variable necessary be the actuator plugin, that are discussed in Section 4.3.

To run the world, there is no need to generate the FMU manually: If the OpenModelica compiler `omc` was installed in the system when the project was configured, the `Damper.fmu` file will be automatically generated by the CMake build. It will then be sufficient to run, as long the `setup-examples.sh` has been correctly sourced, the following command:

```
gazebo -u -e simbody damped_pendulum.world
```

The positional argument `damped_pendulum.world` is the example world, while the additional option `-u` is necessary to start Gazebo in paused mode, to permit to configure the Gazebo plot window (accessible via the Windows → Plot menu) to plot the variables of interest. In particular, for the sake of the analyzing the experiment, it is recommended to plot the joint positions for the two pendulums, plotting respectively the `pendulum_damped_via_sdf/joint?0/position` and the `pendulum_damped_via_fmi/joint?0/position` quantities. Figure 10 shows the output of Gazebo plotting the positions of the pendulum joints.

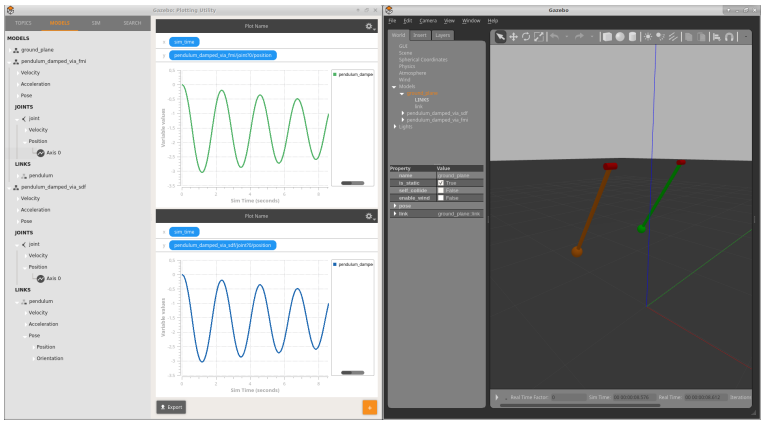

Fig. 10. Output by Gazebo for the angle of the two damped pendulums.

The additional option `-e simbody` is used to ensure that Gazebo uses Simbody [43], a physics engine that generally has lower numerical errors compared to the default physics engine, which is a modified version of the Open Dynamics Engine (ODE). In this particular example, running the simulation with Simbody ensures that the two pendulums are in sync, while running the example with ODE results in small differences that quickly make the two pendulum go out of sync.

4.3 Interface description

In this section, for the sake of clarity, only the input/output interface of each plugin is discussed. Links to the complete documentation are provided for users interested in the additional details.

Actuator plugin. The actuator plugin loads all FMUs that have at least the following four input variables and one output variable:

FMU variable name	Type	Description	Causality
`actuatorInput`	Real	Actuator input	`input`
`jointPosition`	Real	Joint position	`input`
`jointVelocity`	Real	Joint velocity	`input`
`jointAcceleration`	Real	Joint acceleration	`input`
`jointTorque`	Real	Joint torque	`output`

The units of the variables change depending of whether the joint is of *revolute* or *prismatic* type. The convention followed for units is the Gazebo's one of always using SI units.

An empty Modelica model that respects the interface required for an FMU to load by the actuator plugin is shown in Figure 11.

A complete documentation of the parameters support by the `gazebo-fmi-actuator` plugin is given in the `README.md` of the gazebo-fmi repository [8].

Single body fluid dynamics plugin. The single body fluid dynamics plugin loads FMUs that have at least the following three input variables and the six output variables:

FMU variable name	Type	Description	Causality
`relativeVelocity_x`	Real	Body-medium relative velocity (X)	`input`
`relativeVelocity_y`	Real	Body-medium relative velocity (Y)	`input`
`relativeVelocity_z`	Real	Body-medium relative velocity (Z)	`input`
`fluidDynamicForce_x`	Real	Force exerted on the body (X)	`output`
`fluidDynamicForce_y`	Real	Force exerted on the body (Y)	`output`
`fluidDynamicForce_z`	Real	Force exerted on the body (Z)	`output`
`fluidDynamicMoment_x`	Real	Moment exerted on the body (X)	`output`
`fluidDynamicMoment_y`	Real	Moment exerted on the body (Y)	`output`
`fluidDynamicMoment_z`	Real	Moment exerted on the body (Z)	`output`

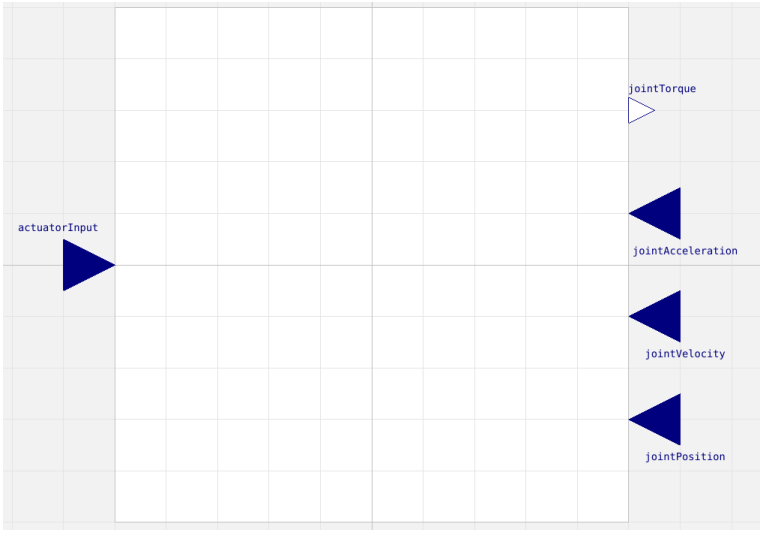

Fig. 11. Empty Modelica model that exposes the variables required for an FMU to be loaded by the actuator plugin.

All the linear velocity components are expressed in $\frac{\mathrm{m}}{\mathrm{s}}$, the linear force components are expressed in N, and the moment components are expressed in Nm. All the quantities are expressed with the body orientation and with respect to the body origin.

An empty Modelica model that respects the interface required for an FMU to load by the single-body-fluid-dynamics plugin is shown in Figure 12.

A complete documentation of the parameters support by the `gazebo-fmi-single-body-fluid-dynamics` plugin is given in the `README.md` of the gazebo-fmi repository [8].

4.4 Architecture and implementation details

Gazebo can be extended by six different types of plugins, which are C++ classes that inherit from a specific C++ interface. In particular, Gazebo supports *World*, *Model*, *Sensor*, *System*, *Visual*, and *GUI*. Each type of plugin is used to customize a different aspect of the simulation. As the gazebo-fmi plugins are used to augment Gazebo models with custom dynamical systems and mainly access model information, they are implemented as Model plugins. For more information on Gazebo plugins, see Gazebo's hello_world tutorial [42].

From the execution point of view, the entry point of all plugins is the `Load()` function, which is called by the Gazebo main physics simulation thread once when the entity that contains the plugin is loaded. During the load phase, the gazebo-fmi plugin searches for the specified FMU in the directory given in `GAZEBO_RESOURCE_PATH` and loads it using the FMI Library. If there is an error in the

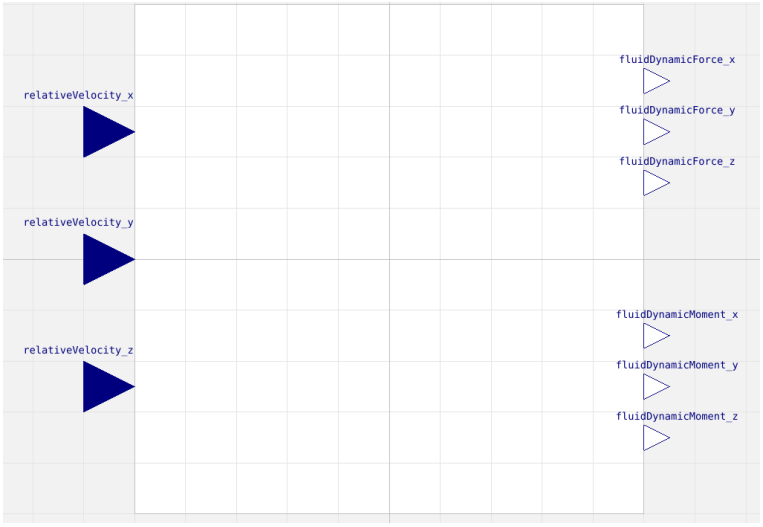

Fig. 12. Empty Modelica model that exposes the variables required for an FMU to be loaded by the single-body-fluid-dynamics plugin.

configuration, the specified FMU is not found, or there is an error during loading, the plugin prints an error message. This message is visible by launching Gazebo with the `--verbose` option. Note that the Gazebo model is still loaded as Gazebo does not expose any overall abort functionality in the case the loading a plugin fails.

Once the FMU is loaded, the co-simulation pattern requires that an FMU co-simulation step is performed at any Gazebo's physics engine update-step, and that after the co-simulation step data is exchanged between the FMU and the Gazebo physics engine. This interaction is implemented via events. The Gazebo Event System is an API that permits to register callbacks for specific events of the simulation cycle. These callbacks are executed by the Gazebo physics thread, ensuring proper synchronization between the Gazebo physics update step and the FMI integration. Examples of events include `Events::worldUpdateBegin`, used for callbacks that need to executed during the begin of a Gazebo physics update step; `Events::beforePhysicsUpdate`, used for callbacks that are executed after the collisition are detected, but before the physics's engine integrator computed the new positions and velocities of the model, and `Events::worldUpdateEnd`, for code that needs to be executed at the end of the update step. Most Model plugins, including middleware-specific plugins such as the one contained in `gazebo_ros_pkgs`, typically just execute their periodic code during the `Events::worldUpdateBegin` callback.

Depending on the specific gazebo-fmi plugin, the actual FMI step and exchange of input/output data with the Gazebo physics engine is done during different events.

Actuator plugin. The goal of the actuator plugin is to be used transparently with the joint of any Gazebo model, regardless of how the joint is controlled: using the Gazebo internal PID controller or using middleware specific control plugin such as gazebo_ros_pkgs or gazebo-yarp-plugins.

All this software control the behavior of the joints by calling the function `Joint::SetForce` in the main Gazebo physics thread, as part of the callback of the `Events::worldUpdateBegin` event. For this reason, the actuator plugin code runs *after* the joint control code has run in the `events::worldUpdateBegin` event, but *before* the actual physics engine is updated.

This is achieved by running in the callback of the event `events::beforePhysicsUpdate`. The actuator plugins reads the value that was set in the `Joint::SetForce` function and it passes it to the FMU as the `actuatorInput`. It then runs the FMU simulation, and it substitutes the value that was set in the `Joint::SetForce` with the `jointTorque` output of the FMU. For more information, this logic can be found in the `FMIActuatorPlugin::BeforePhysicsUpdateCallback` function code.

Single body fluid dynamics plugin. The single body fluid dynamics plugin code is called during the `events::worldUpdateBegin` event. The plugin reads the velocity of the link to which the plugin is associated and the velocity of the wind, and computes from these two quantities the relative velocity, which is then passed to the FMU. The computed force/torque is then added back to the link using Gazebo APIs to add external forces to links.

The mechanism of the plugin was inspired by the Gazebo plugin `LiftDragPlugin`, described in the Gazebo's Aerodynamics tutorial [42].

5 Use-cases

In this section, we report on the use of the fmi_adapter in a ROS-based autonomous intra-logistics vehicle and on the implementation of an electrical actuator model based on gazebo-fmi.

5.1 FMU-based controller for a self-driving vehicle in intra-logistics

Industrial logistics is an important application area of autonomous robotics, which gave rise to many new innovative products and solutions such as the *ActiveShuttle* from Bosch Rexroth [44], which is considered in this use-case. In the last years, a number of elaborate algorithms for task scheduling, coordination and path planning for fleets of self-driving vehicles in such applications have been proposed. Prerequisite to apply these strategies is a reliable vehicle motion control. Trajectories commanded by the planner need to be properly executed by the drive platform to ensure that the goals of the mission are met in time and space.

Model-based design is a well-established approach to develop and implement motion control strategies. Model-in-the-loop (MiL) simulations allow the early

validation and test of the controller design against a plant model. Based on a physical model of the plant, dedicated controllers such as model predictive control (MPC) or feedback linearization can be designed that take the physical properties and system dynamics into account.

Exporting the validated controller model directly from the modeling and simulation environment as an FMU allows utilizing the fmi_adapter to integrate the model-based control function directly into a ROS-based software architecture. This generic and efficient workflow to enhance the capabilities of a robotic system with an advanced control function is demonstrated in the following by the self-driving intra-logistics vehicle *ActiveShuttle DevKit*, which is the predecessor of today's product.

The analysis of various operation scenarios revealed that the transitions between moving straight and turning on the spot are prone to high bore friction (cf. Figure 14) at the caster wheels. Typically, it is assumed that the passive caster wheels follow the commanded vehicle trajectory perfectly without interference. In practice, however, it can be observed that this is not necessarily the case. After standstill, a significant counteracting force occurs when the current orientation of the caster wheels is not compliant with the new commanded direction of motion.

To avoid locking under these potentially critical conditions, the *Path Filter* control function was developed avoiding high bore friction at the caster wheels. The self-contained control function was then integrated with the existing ROS-based motion planner and motion controller to improve the robustness and overall performance of the differential drive vehicle.

Model-based control design and validation in Modelica. Following the outlined model-based control design process, a plant model of the ActiveShuttle DevKit was developed to replicate the undesired behavior of differential drive vehicles with caster wheels.

The plant model is implemented in the Modelica language based on the open-source PlanarMechanics library [45]. The model takes the kinematic structure of the six-wheeled ActiveShuttle DevKit into account, as illustrated in Figure 13. Lateral, longitudinal, and rotational inertial forces in the xy plane are considered by the planar body at the center of gravity. The distribution of the normal gravitational force to the kinematically overconstrained ground contact forces at the wheels is resolved by taking the chassis kinematics and orientation of the caster wheels into account.

In order to account for the physical effect of bore friction at the caster wheels properly, a tire model has been developed considering a contact patch as illustrated in Figure 14.

The approach of [46] to describe the bore friction characteristic, see Figure 15 on the left, has been improved as described in [47] by the characteristic in Figure 15 on the right. In contrast to the other implementation does this model account for a continuous decrease of the bore torque with increasing rolling ve-

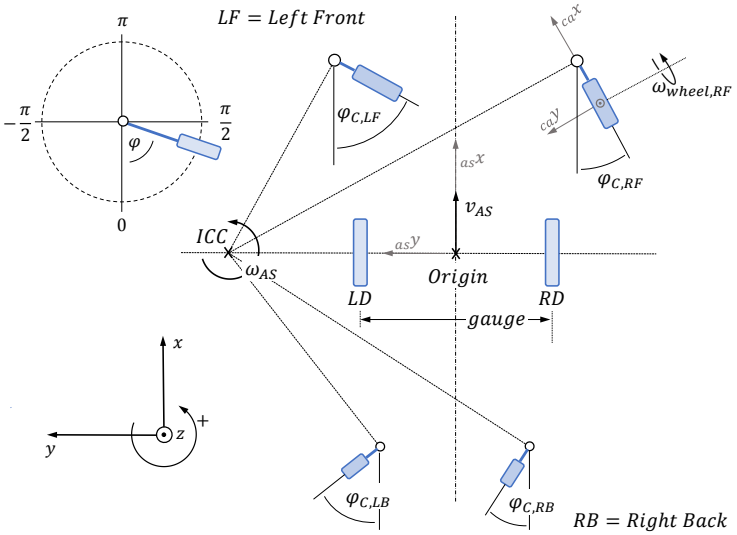

Fig. 13. ActiveShuttle DevKit motion state of driven and caster wheels.

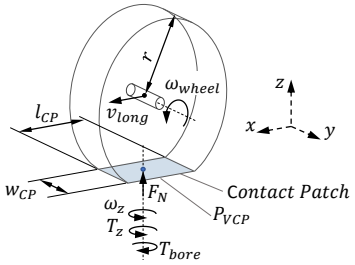

Fig. 14. Properties of the tire road contact considered in the wheel model.

locity ω_{wheel}, which is crucial to properly model the locking condition with ω_z and ω_{wheel} close to zero.

Based on this caster wheel model and the plant model of the ActiveShuttle DevKit, the critical operation scenarios were successfully simulated and analyzed, as depicted in Figure 16.

While moving straight all caster wheels are facing forward with the instantaneous center of curvature (ICC) being located in infinite lateral distance from the origin according to Figure 13. If the vehicle is stopped and a turn on the spot is introduced, the new position of the ICC will be at the origin. Accordingly, the orientations of the caster wheels are no longer compliant with the commanded motion state. This implies that all caster wheels have to be turned by an angle of $\Delta\phi_i$. As the rolling velocities of the caster wheels are initially zero, the maximum bore torque applies at all four caster wheels and has to be overcome simultaneously in addition to the inertial torque of the vehicle. Under the given

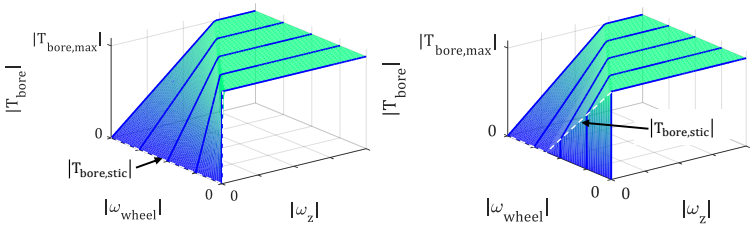

Fig. 15. Bore friction characteristics on the left as proposed by [46], on the right the enhanced variant as applied to the ActiveShuttle DevKit plant model.

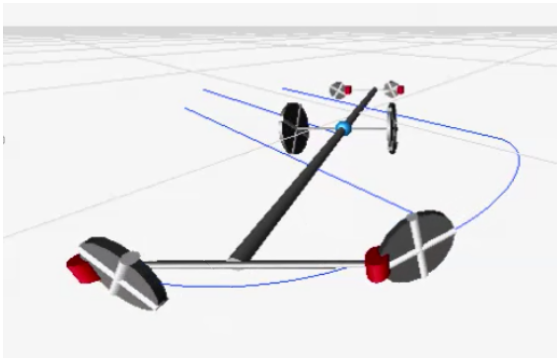

Fig. 16. 3D Animation of the simulated turning on the spot of the ActiveShuttle DevKit with *Path Filter* in Dymola.

geometric relations of the ActiveShuttle DevKit, the counteracting friction forces applied at the mountings of the caster wheels have a longer lever arm than the driving forces at the driven wheels. Under full load and for high stiction coefficients (rubber tires on concrete floor) the counteracting forces lead to high peak current of the drives and can even exceed the power limits of the drives.

Based on this simulation model, a control strategy to enhance the existing control architecture was developed and tested. The basic idea behind the *Path Filter* is to circumvent the high bore friction torque for $\omega_{wheel} \approx 0$ by forcing the motion controller to follow a trajectory that first sets the caster wheels into a rolling motion before introducing the turn.

Following this approach, the Path Filter can be implemented as a function with the commanded left and right drive velocities as inputs and the corrected command velocities as outputs. The input velocities are used to estimate the motion state of the caster wheels $[\phi_i, \omega_i]$ in order to determine the gap $\Delta\phi$ between the current and the desired orientation of the caster wheels. The following relation is then used to determine the corrected motion state of the caster wheels, which is finally projected back onto the velocities of the driven wheels and provided as filtered command value

$$\varphi_{C,fil} = \hat{\varphi}_C + k \cdot \Delta\varphi_C \tag{1}$$

to the motion controller, with

$$k = min(1, |\frac{\omega_C}{\omega_{max}}|). \tag{2}$$

When the ActiveShuttle DevKit is at standstill, the motion controller is forced to start moving in the direction of the current orientation of the caster wheels $\hat{\varphi}_C$. With increasing rolling velocity of the caster wheels ω_C, the influence of the correcting term is decreased linearly to zero for $\omega_C \geq \omega_{max}$. In other words, the path filter describes a trajectory in the bore friction characteristic of the caster wheels that avoids the peak friction torques (cf. Figure 15). ω_{max} represents the slope of k. Small values reduce the time Δt that is necessary to transfer between estimated and desired state, but at the same time describe a trajectory in the bore friction characteristic that encounters higher bore torques. Hence, the choice of ω_{max} can be seen as a trade-off between deviation from the commanded trajectory and required effort.

Controller integration using the fmi_adapter. In order to integrate the previously described path filter functionality into the existing ROS-based navigation architecture, the fmi_adapter has been utilized as described in Section 3. Therefore, the tested and validated function has been exported directly from the simulation environment. This was performed simply by opening the path filter block of the Modelica model and using the FMU for Co-Simulation export command.

In the navigation architecture of the ActiveShuttle DevKit, drive commands are represented as `TwistStamped` messages (from the `geometry_msgs` package) representing the lateral and rotational velocity in the corresponding fields `twist.linear.x` and `twist.angular.z`. This is a very common representation in ROS for velocity commands for differential drive robots. In the navigation stack of the ActiveShuttle DevKit, the desired velocity is sent as twist message from the *path tracker* node to the *engine driver* node on a topic named `/velDes`. The path tracker node determines the desired velocity based on the input from the global *path planner*. The engine driver node translates the twist command into motor speed commands for the left and right drive.

In order to consume the previously exported *Path Filter FMU*, a new node named `PathFilter` is added using the fmi_adapter library. This new node consists of one function `main` with only 25 lines of code. The path filter node is placed between the path tracker and the engine driver node, as depicted in Figure 17, by receiving a `TwistStamped` message on the `/velDes` topic and publishing the resulting filtered velocity command outputs as `TwistStamped` message published on a new `/velDesFil` topic.

The `PathFilter` node feeds the Path Filter FMU with the values of `twist.linear.x` and `twist.angular.z` using the `setInputValue` interface. The FMU is

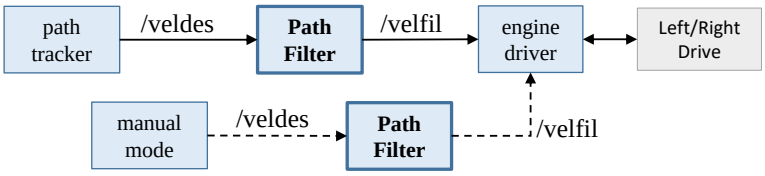

Fig. 17. ROS graph with the path filter FMU node integrated into the ActiveShuttle DevKit navigation architecture (solid lines) and the test setup (dashed lines).

then updated to the current time. The resulting filtered velocity commands are read through the `getOutputValue` interface from the FMU and published.

Integrating the `PathFilter` node in the existing architecture, required only two lines in the corresponding ROS launch file to be changed: A new line for the `PathFilter` node was added and the input topic for the engine driver node changed to `/velDesFil`.

Application and test results. For test purposes, a simplified setup of the control architecture has been applied using the *manual* mode, as depicted in Figure 17 (dashed lines). In this setup the `PathTracker` node is deactivated and the desired motion is published to the `/velDes` topic by a node that is interfaced with a joystick. This way it was straightforward to try out the following motion profile:

forward → stop → turn→ stop → forward → stop → backward

The profile was driven with and without path filter. The ActiveShuttle DevKit was fully loaded with 150 kg, which leads to a total weight of ≈ 200 kg. The relevant topics `/velDes`, `/velDesFil`, and `/enginedata` were recorded with the `rosbag` tool. The latter topic holds the messages with the measured motor speeds and currents. In addition, video recordings documented the movement of the caster wheels.

The measured motor currents for the first part of the motion profile (turning on the spot) are plotted in Figure 18. Comparing the solid and dashed curves shows that the total effort necessary to perform the turn on the spot is significantly reduced by the path filter. The peak currents are reduced by 30% from 18 A to 12 A. Consequently, also the oscillations are significantly smaller which can also be observed in the video recordings.

Conclusions. The use-case of extending the motion controller of a self-driving vehicle for intra-logistics demonstrates an efficient model-based design workflow

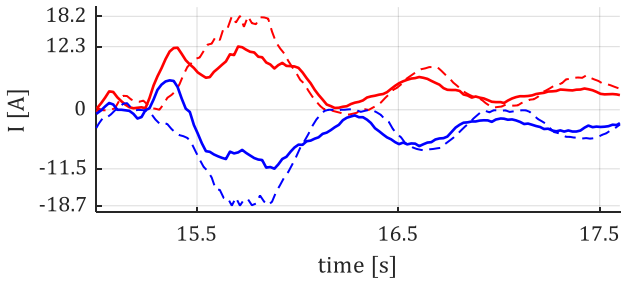

Fig. 18. Motor currents left/right (red/blue) with Path Filter (solid) and without Path Filter (dashed) for a turning-on-the-spot movement.

for ROS enabled by the fmi_adapter. Through the fmi_adapter, it is possible to directly integrate a control function designed and validated in a simulation environment as FMU into an existing ROS architecture. This allows taking full advantage of a simulation environment to analyze the system behavior, try out different control strategies, develop and parameterize a control function and verify its proper function in a safe environment before applying the approach to the actual robot.

Using the fmi_adapter package, a minimal implementation effort is required to integrate the control function exported as FMU from the simulation environment as node into a ROS control architecture.

The application and test of the path filter on the ActiveShuttle DevKit show that the proposed small enhancement of the motion controller can

1. reduce the risk of entering a locking condition when trying to perform a turn on the spot after stand still by reducing the peak currents,
2. reduce the power consumption of turns on the spot,
3. reduce oscillations, and
4. increase the durability of the hardware components by reducing the jerk.

The example of the path filter impressively highlights how much the performance of a robot can be improved by considering the impact of physical effects (in this case bore friction) in the control software, and how the fmi_adapter enables efficient model-based design workflows for ROS relying on open standards.

With the up-coming eFMI (FMI for Embedded Systems) standard [48], currently developed in the ITEA3 project EMPHYSIS (Sept. 2016 - Aug. 2020) [17], this approach will also be applicable to real-time systems and devices with very limited computation power such as microcontrollers or other embedded devices, enabling new workflows for model-based control and diagnosis function development [27].

5.2 Electrical actuator simulation using gazebo-fmi actuator plugin

The physics engines used by Gazebo are specialized simulation tools that simulate the dynamics of rigid body systems, i.e., systems composed by rigid bodies

(called *links*) interconnected by joints. To simulate joints that are actuated such as the one of robots, these engines take as one of their simulation inputs the force applied on the joint, which is usually called joint *effort*, or joint *torque* for revolute joints. For this reason, any joint position controller that is implemented in Gazebo to drive the joint uses as its output the joint effort, which is then passed to the simulation engine. With respect to real robots, this implies that the simulation is neglecting how the effort is actually applied on the joint in the real robot, for example via electric or hydraulic motors.

This has two main implications: First, some relevant aspects of the dynamics of the overall system may be missing from the simulation, as noted in [5,6,7]. Second, unless the robot system implements some sort of joint torque feedback, the typical joint controller running on a real robot does not directly set the effort in the joint but rather computes some value for the actuator-specific input signal such as *voltage* or *current* for electric motors. This means that the gains used and tuned in the Gazebo joint controllers cannot be directly used on the real robot, as the order of magnitude and units may be completely different. For example, for a voltage-controlled electrical actuator mounted on a revolute joint the proportional gain tuned in Gazebo is expressed as $\frac{\mathrm{Nm}}{\mathrm{rad}}$, while the corresponding gain on the real robot is expressed in $\frac{\mathrm{V}}{\mathrm{rad}}$.

Modeling at least some basic aspects of the actuation dynamics permits to reduce, with limited modeling effort, the simulation-to-reality gap and the portability of gains between the simulated robot and the real robot.

Simple electrical actuator. For an example of this use-case, we will discuss the simple electrical actuator example shipped with `gazebo-fmi` itself, available at `gazebo-fmi/plugins/actuator/examples/simple_electrical_actuator`.

While for the sake of simplicity this example simulates just a one degrees-of-freedom robot, the order of magnitude of the parameters and the elements of the models have been chosen based on the actuator models used for humanoid robot balancing and walking, in particular on the actuators of the iCub humanoid robot [9,49].

This directory contains an example Gazebo world that contains two one degrees of freedom manipulators. The models are identical, with the difference that one (called `no_actuator_model`) uses the standard Gazebo actuation model, while the other (called `simple_electrical_actuator_model`) is using a simple model of an electrical actuator that is contained in the `SimpleElectricalActuator.fmu`, which is generated from the `SimpleElectricalActuator.mo` Modelica model.

The Modelica model for the simple electrical actuator is shown in Figure 19. Its `actuatorInput` is the voltage of the motor, expressed in Volt. Directly after the input, the voltage is limited between $-48\,\mathrm{V}$ and $+48\,\mathrm{V}$ by a limiter. After that, the voltage drives a simple electrical circuit composed by a resistor with a resistance of $1\,\Omega$, and a back electromotive force (EMF) component with a motor gain of $0.1\,\frac{\mathrm{Nm}}{\mathrm{A}}$. The mechanical output of the EMF drives a rotational inertia of $10^{-5}\,\mathrm{kg\,m^2}$, which represents the inertia of the motor's rotor. The inertia then drives a ideal gear box with a reduction ratio of 100, which is then connected

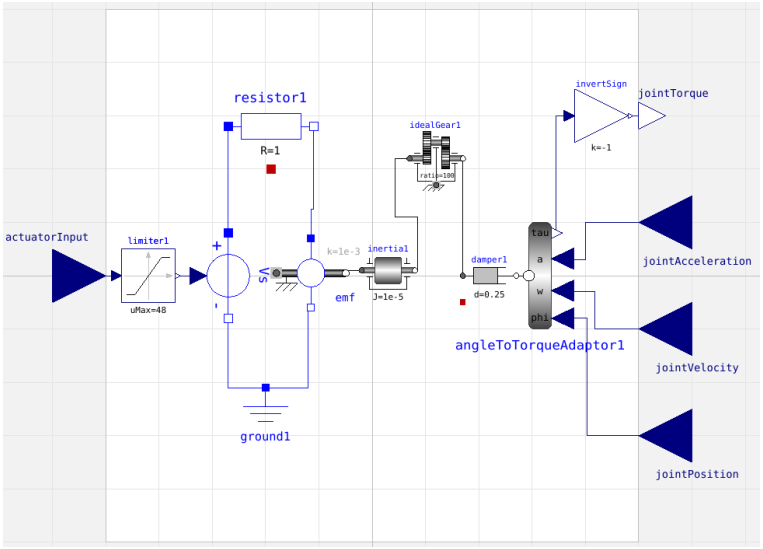

Fig. 19. Modelica model for a simple electrical actuator

to a damper with a damping coefficient of 0.25 $\frac{\mathrm{Nms}}{\mathrm{rad}}$, which is connected to the Gazebo model itself.

Run the example. As discussed in Section 4.2, to run this example, it is not necessary to manually configure the `GAZEBO_PLUGIN_PATH` and `GAZEBO_RESOURCE_PATH` environment variable of Gazebo, but it is sufficient to directly source the `<install_prefix>/share/gazebo-fmi/examples/setup-examples.sh`. The script will set all the environment variables necessary to run the examples shipped with gazebo-fmi:

```
source <install_prefix>/share/gazebo-fmi/examples/setup-examples.sh
```

At this point, you can launch the simulation using the following command:

```
gazebo simple_electrical_actuator.world
```

The two models will start in the 0 position. You can change the target position of the internal Gazebo joint controllers using the `gz joint` command:

```
gz joint -m no_actuator_model -j joint --pos-p 10.0 --pos-d 3.0 --pos-t
    ↪ 1.0 && gz joint -m simple_electrical_actuator_model -j joint --pos
    ↪ -p 10.0 --pos-d 3.0 --pos-t 1.0
```

The `pos-t` option specifies the target of the joint position controller, while `pos-p` and `pos-d` specify the position and derivative gains of the internal Gazebo PID joint controller. Interestingly, due to the use of the low-level electrical actuator model, the gains that are able to stabilize with minimal overshoot and oscillations the `no_actuator_model` are not a good fit instead for the `simple_electrical_`

`actuator_model`, as shown in Figure 20. The idea is that in this way the Gazebo model, as long as the electrical actuation is similar to the real model, can be used for the tuning (at least preliminary) of the gains of the real joint torque controllers.

To set a different set of gains that instead is able to reduce the oscillations on the `simple_electrical_actuator_model`, execute the following command:

```
gz joint -m simple_electrical_actuator_model -j joint --pos-p 109.0 --pos
    ↪ -d 60.0 --pos-t 1.0
```

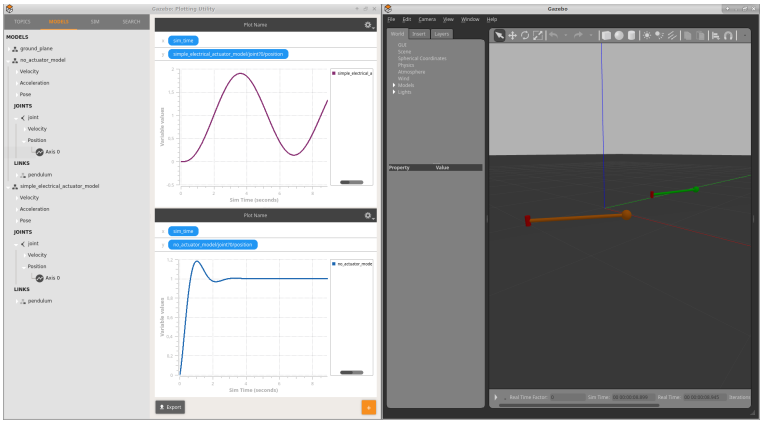

Fig. 20. Plot in Gazebo for the angle of the two controlled joints.

6 Final remarks

In this chapter, two approaches to bridge the gap between classical simulation tools and ROS and Gazebo have been presented, based on the open FMI standard. Two practical use-cases of FMI have illustrated the many benefits of the proposed integrations in terms of enhanced system analysis and simulation capabilities, reusability of models, and increased development efficiency.

In the next years, with the upcoming FMI 3 standard [1] and the development of FMI for Embedded Systems (eFMI) [27], the Functional Mock-up Interface will become even more relevant for robotic systems and software engineering. The fmi_adapter is a first step to allow ROS developers to benefit from these developments. More features are desirable. Similarly, more gazebo-fmi plugins are imaginable, for example to simulate complex surface properties in mobile robotics. Therefore, we cordially invite to contribute to the corresponding repositories.

References

1. Modelica Association: Functional Mock-up Interface (FMI) Website. Retrieved 10 August 2019 from https://fmi-standard.org/.
2. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-Simulation: A Survey. ACM Computing Surveys **51**(3) (May 2018) 49:1–49:33
3. Robert Bosch GmbH: fmi_adapter package for ROS. Retrieved 10 August 2019 from https://github.com/boschresearch/fmi_adapter.
4. Robert Bosch GmbH: fmi_adapter package for ROS 2. Retrieved 10 August 2019 from https://github.com/boschresearch/fmi_adapter_ros2.
5. Neunert, M., Boaventura, T., Buchli, J.: Why off-the-shelf physics simulators fail in evaluating feedback controller performance – a case study for quadrupedal robots. In: Advances in Cooperative Robotics: Proceedings of the 19th International Conference on CLAWAR 2016. World Scientific (2016) 464–472
6. Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., Hutter, M.: Learning agile and dynamic motor skills for legged robots. Science Robotics **4**(26) (2019)
7. Tan, J., Zhang, T., Coumans, E., Iscen, A., Bai, Y., Hafner, D., Bohez, S., Vanhoucke, V.: Sim-to-Real: Learning Agile Locomotion For Quadruped Robots. In: Proceedings of Robotics: Science and Systems XIV. (June 2018)
8. Italian Institute of Technology: gazebo-fmi plugins. Retrieved 10 August 2019 from https://github.com/robotology/gazebo-fmi/.
9. Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., von Hofsten, C., Rosander, K., Lopes, M., Santos-Victor, J., Bernardino, A., Montesano, L.: The iCub humanoid robot: An open-systems platform for research in cognitive development. Neural Networks **23**(8) (2010) 1125–1134
10. ITEA: Modelisar Impact Story. Retrieved 21 April 2019 from https://itea3.org/project/impact-stream/modelisar-impact-story.html.
11. MathWorks, Inc.: What Is an S-Function? Retrieved 29 April 2019 from https://de.mathworks.com/help/simulink/sfg/what-is-an-s-function.html.
12. Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.-V., Wolf, S.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: Proceedings of the 8th International Modelica Conference, Dresden, Germany, Linköping University Electronic Press, Linköpings universitet (2011)
13. Modelica Association: Functional Mock-up Interface for Model Exchange and Co-Simulation (Standard), July 2014. Retrieved 29 April 2019 from https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf.
14. Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proceedings of the 9h International Modelica Conference, Munich, Germany, Linköping University Electronic Press, Linköpings universitet (2012)
15. Modelica Association: Modelica Website. Retrieved 10 August 2019 from https://www.modelica.org/.
16. Bertsch, C., Ahle, E., Schulmeister, U.: The Functional Mockup Interface – seen from an industrial perspective. In: Proceedings of the 10th International Modelica Conference, Lund, Sweden (March 2014) 27–33

17. ITEA: 15016 EMPHYSIS Embedded systems with physical models in the production code software. Retrieved 25 May 2019 from https://itea3.org/project/emphysis.html.
18. PKWARE Inc.: ZIP Specification. Retrieved 21 April 2019 from https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.
19. Modelon AB: FMI Compliance Checker (FMUChecker). Retrieved 10 August 2019 from https://github.com/modelica-tools/FMUComplianceChecker.
20. Dassault Systèmes: FMPy. Retrieved 10 August 2019 from https://github.com/CATIA-Systems/FMPy.
21. Modelica Association: FMI Cross Check Rules. Retrieved 26 May 2019 from https://github.com/modelica/fmi-cross-check/.
22. Dassault Systèmes: Test FMUs. Retrieved 10 August 2019 from https://github.com/CATIA-Systems/Test-FMUs.
23. Bertsch, C., Mukbil, A., Junghanns, A.: Improving Interoperability of FMI-supporting Tools with Reference FMUs. In: Proceedings of the 12th International Modelica Conference, Prague, Czech Republic (May 2017) 533–540
24. Bastian, J., Clauss, C., Wolf, S., Schneider, P.: Master for Co-Simulation Using FMI. In: Proceedings of the 8th International Modelica Conference, Dresden, Germany (March 2011) 115–120
25. Benedikt, M., Watzenig, D., Zehetner, J., Hofer, A.: EPCE – A nearly energy-preserving coupling element for weak-coupled problems and co-simulation. In: Proceedings of the International Conference on Computational Methods for Coupled Problems in Science and Engineering. (June 2013) 1–12
26. Bertsch, C., Neudorfer, J., Arumugham, E.A.S.S., Ramachandran, K., Thuy, A.: FMI for physical models on automotive embedded targets. In: Proceedings of the 11th International Modelica Conference, Versailles, France (September 2015) 43–50
27. Lenord, O., Jarmolowitz, F., Kiesenhofer, W., Rath, K., Obertopp, T.: Towards an Integrated Tool Chain from Physical Models to Diagnosis Functions. In: Proceedings of the 12th MODPROD Workshop 2018 on Cyber-Physical Systems of Systems, Linköping, Sweden (February 2018)
28. JModelica.org: FMI Library. Retrieved 10 April 2019 from https://jmodelica.org/.
29. SFI Offshore Mechatronics Research Centre: FMI4cpp. Retrieved 10 April 2019 from https://github.com/NTNU-IHB/FMI4cpp/.
30. Open Source Robotics Foundation: Catkin Command Line Tools. Retrieved 26 April 2019 from https://catkin-tools.readthedocs.io/.
31. Kitware, Inc. and Contributors: CMake Documentation – Modules: ExternalProject. Retrieved 26 April 2019 from https://cmake.org/cmake/help/latest/module/ExternalProject.html.
32. Thomas, D.: colcon – collective construction. Retrieved 26 April 2019 from https://colcon.readthedocs.io/.
33. Open Source Modelica Consortium: OpenModelica. Retrieved 10 August 2019 from https://openmodelica.org/download/download-linux.
34. QTronic: FMU SDK. Retrieved 10 August 2019 from https://www.qtronic.de/en/fmu-sdk/.
35. Open Robotics: Management of nodes with managed lifecycles. Retrieved 28 May 2019 from https://index.ros.org/doc/ros2/Tutorials/Managed-Nodes/.
36. Koenig, N., Howard, A.: Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Volume 3. (Sep. 2004) 2149–2154

37. Manhes, M.M.M., Scherer, S.A., Voss, M., Douat, L.R., Rauschenbach, T.: UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation. In: OCEANS 2016 MTS/IEEE Monterey. (September 2016)
38. Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., von Stryk, O.: Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo. In Noda, I., Ando, N., Brugali, D., Kuffner, J.J., eds.: Simulation, Modeling, and Programming for Autonomous Robots, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 400–411
39. Thomas, D., Contributors: A universal build tool. Retrieved 27 April 2019 from `https://design.ros2.org/articles/build_tool.html`.
40. Open Source Robotics Foundation: Install Gazebo using Ubuntu packages. Retrieved 27 April 2019 from `http://gazebosim.org/tutorials?tut=install_ubuntu`.
41. Kitware, Inc. and Contributors: CMake Documentation – Modules: FetchContent. Retrieved 27 April 2019 from `https://cmake.org/cmake/help/v3.14/module/FetchContent.html`.
42. Open Source Robotics Foundation: Gazebo Tutorials. Retrieved 10 August 2019 from `http://gazebosim.org/tutorials`.
43. Sherman, M.A., Seth, A., Delp, S.L.: Simbody: multibody dynamics for biomedical research. Procedia IUTAM **2** (2011) 241–261
44. Bosch Rexroth: ActiveShuttle – setting your intralogistics in motion. Retrieved 30 May 2019 from `https://www.boschrexroth.com/en/xc/products/product-groups/assembly-technology/topics/intralogistics/template-neuprodukt-seite-6`.
45. Zimmer, D.: PlanarMechanics Library 1.4.0 (2017-01-12). Retrieved 31 May 2019 from `https://github.com/dzimmer/PlanarMechanics`.
46. Zimmer, D., Otter, M.: Real-time models for wheels and tyres in an object-oriented modelling framework. Vehicle System Dynamics (2010)
47. Schröder, N., Lenord, O., Lange, R.: Enhanced Motion Control of a Self-Driving Vehicle Using Modelica, FMI and ROS. In: Proceedings of the 13th International Modelica Conference, Regensburg, Germany, Linköping University Electronic Press, Linköpings universitet (March 2019)
48. EMPHYSIS Project: EMPHYSIS – Functional Mock-up Interface for Embedded Systems. Retrieved 25 May 2019 from `https://emphysis.github.io/`.
49. Nava, G., Pucci, D., Nori, F.: Momentum control of humanoid robots with series elastic actuators. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). (Sep. 2017) 2185–2191

Authors' Biographies

Ralph Lange joined Bosch Corporate Research in 2013. He is currently leading an internal research project on methods, tools, and frameworks for systems and software engineering in robotics. In this context, he is also principal investigator in the EU project OFERA, which brings ROS 2 on microcontrollers in an open-source initiative named micro-ROS. Ralph actively follows the development of ROS 2. Amongst others, he researches execution management and run-time reconfiguration concepts for ROS 2. Before joining Bosch, Ralph worked as a software engineer for TRUMPF Machine Tools on a graphical programming system for punching and punch-laser machines. He obtained his PhD in the field

of scalable context-aware systems in 2010 at the Institute for Parallel and Distributed Systems of the University of Stuttgart, Germany.

Silvio Traversaro joined the Italian Institute of Technology (IIT) as a Postdoctoral Researcher in 2017. As part of the iCub Tech facility at IIT, he is currently working on industrial and commercial applications of robotics research in several sectors, including automation for the metal processing industry. Furthermore, he collaborates with the Dynamic Interaction Control research line at IIT, working on software architectures and estimation techniques for legged walking, aerial humanoid robotics, and reinforcement learning. He obtained his PhD in Robotics from the University of Genoa, working at the Italian Institute of Technology on dynamics modeling, estimation, and identification applied to the iCub humanoid robot.

Oliver Lenord joined Bosch Corporate Research in 2016. He is currently leading the European publicly funded ITEA3 project EMPHYSIS aiming to develop the eFMI (FMI for Embedded Systems) standard. He is founding member and previous chairman of the OSMC (Open Source Modelica Consortium). He is also supporting the systems and software engineering research project in robotics in terms of model-based design methods. From 2011 until 2016, he worked as product manager of the Mechatronics Concept Designer at Siemens PLM in Cypress, California, USA. After his PhD in the field of robotics at the Institute of Mechatronics at the University of Duisburg-Essen, Germany, his first position in the industry was at Bosch Rexroth AG, where he was leading a simulation and engineering software development group for hydraulic applications and industry automation.

Christian Bertsch joined Robert Bosch GmbH in 2001. Within Bosch Corporate Research, he is currently leading a research project on advanced model-based functions and their numerical realization on embedded systems. There he also investigated the extension of FMI towards embedded systems – as will be standardized in the ITEA3 EMPHYSIS project. He is a personal member of the Modelica Association, represents Bosch in the FMI Steering Committee, and actively participates in the further development of the FMI Standard. Within Bosch, he coordinates the FMI activities both internally with training and consulting, and in collaboration with tool vendors. Before joining Bosch, he graduated in Mathematics and Physics with a focus on numerical mathematics at the University of Heidelberg, Germany.