INtegrated TOol chain for model-based design of CPSs

INTO-CPS External Data Broker FMU

(RabbitMQ FMU)

Version: 1.0

Date:

The INTO-CPS Association

http://into-cps.org

Contributors:

Casper Thule, Aarhus University, Centre for Digital Twins
Kenneth Lausdahl, Mjølner Informatics A/S
Cláudio Gomes, Aarhus University, Centre for Digital Twins

Editors:

Casper Thule, Aarhus University, Centre for Digital Twins

Document History

Ver	Date	Author	Description
1.0	January 23, 2019	Casper Thule	First version.

Abstract

This document concerns an implementation of an External Data Broker (EDB) FMU based on RabbitMQ. An EDB FMU brings external data into an FMI context by receiving data external to the FMI simulation and make it available via FMI outputs. The specific RabbitMQ based implementation is realised in RabbitMQ FMU.

Several of the constructs in the approach presented in this publication is general and can applied to multiple cases. For example, the RabbitMQ FMU can be adapted to process different messages by making changes to its static description file, without the need to recompile the binaries, which are already compiled for Mac, Linux and Windows. Additionally, the implementation can be made to fit other message-oriented or data-stream middlewares while still reusing most of the source.

As such, while RabbitMQ FMU is exemplified with a certain scenario, it can be applied generally and does not have a specific scenario tied to it.

Contents

1 Introduction

EDB is short for External Data Broker and EDB FMU is an FMU that brings external data into an FMI context. The implementation of EDB FMU described in this publication is based on RabbitMQ. However, this is just one way to implement an EDB FMU. Several of the constructs described in this publication are general and can be used with other message-oriented or data streaming middleware. For this reason, both EDB FMU and RabbitMQ FMU will be used throughout this document, where EDB FMU is generally applicable functionality, whereas RabbitMQ FMU is an implementation of EDB FMU specific to RabbitMQ. Thus, one could take the existing source code of RabbitMQ FMU and fit it to another middleware than RabbitMQ while still reusing most of the source code. Note, that this document assumes general knowledge of the Functional Mock-up Interface (FMI) [Mod19].

An overall approach to using an EDB FMU in a digital twin context is depicted in fig. 1, where the content within the *System Specific* frame will vary based on the system providing the data. The Log-Translator entity translates the system-specific log messages to digital twin compatible messages and publishes them to a Data Middleware Node (i.e. a RabbitMQ Node). The EDB FMU is configured via its static description to receives messages from the Data Middleware Node. The message content is then parsed and published via regular FMI outputs.

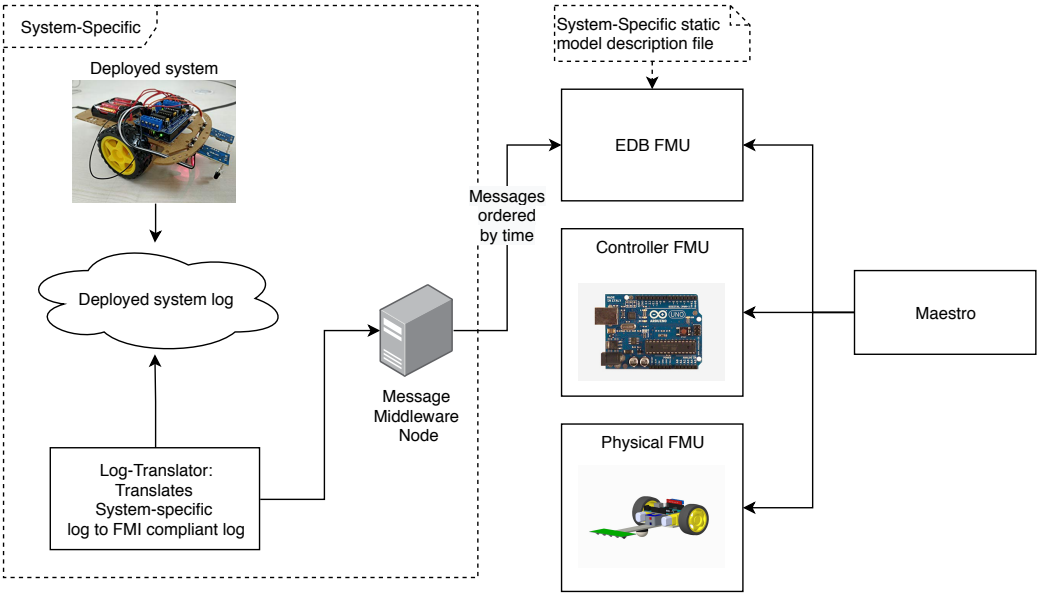

Figure 1: Interfacing Overview

This approach generalises the FMI enabling entity, the EDB FMU, such that it can be used for different kinds of systems with different kinds of logging facilities. The motivation behind this approach is that tools of the INTO-CPS Association should be generally applicable.

The following chapters describes the following in order:

Time Handling How system-time and simulation time is mapped and put in an FMI simulation context.

Data Handling How message content and the state of EDB FMU are coordinated.

Configuration How to configure an EDB/RabbitMQ FMU via the ModelDescription File

Example - Single Water Tank RabbitMQ This example demonstrates how RabbitMQ FMU acts as an External Data Broker in context of a Single Water Tank Digital Twin. It focuses solely on this part and does not contain a deployed system.

Example - Line Following Robot IN PROGRESS - This example demonstrates transmission of data from a deployed line following robot. The data is made available in the FMI co-simulation via RabbitMQ FMU. Thus, this is a example with both a deployed system, its digital twin and the RabbitMQ FMU.

Future Work Describes some ideas for future work.

2 Time Handling

This section concerns the digital twin and how data time is handled with relation to FMI. The description below is accompanied by fig. 2.

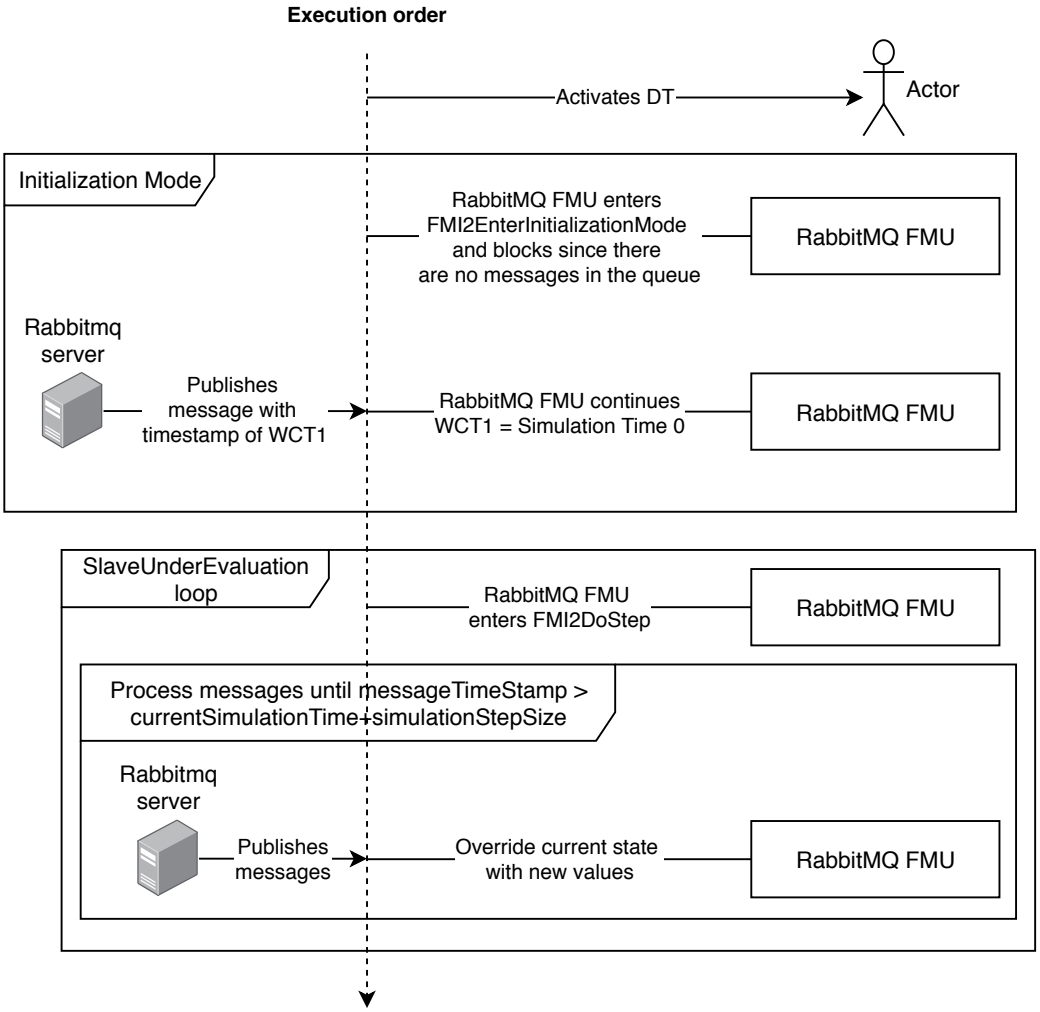

Figure 2: Time Handling in Simulation View

Invoking the function `fmi2EnterInitializationMode` on EDB FMU causes it to block until a message is available.
The time stamp of the first message ($WCT1$) received defines $simulationTime0$ and EDB FMU continues. Thus $WCT1$ = $simulationTime0$ and a mapping between WCT and simulation time is establishedcreated. Thus, any subsequent timestamps has $WCT1$ subtracted in order to map them to $simulationTime$.

This also implies that any message with a time stamp of $WCT0 < WCT1$ are ignored.

Invoking the function `fmi2DoStep` on EDB FMU causes it to process messages and keep executing until there is a message with a time stamp defined by $messageTimeStampInSimulationTime >= currentSimulationTime + simulationStepSize$. Such a message is stored in order to use it for the subsequent `fmi2DoStep` operation.

3 Data Handling

The data handling occurring within `fmi2DoStep` of EDB FMU is described below. All values of messages with time stamps (converted to simulation time) within the time interval $]currentSimulationTime, currentSimulationTime + simulationStepSize]$ overrides the current state of values in the received order, thus using zero-order hold. This depends on the Data Middleware Node providing the data in order according to time, as mentioned in TODO.

An example is given in fig. 3. First, `message x` is received and overwrites the value of `a` in the EDB FMU state.
Afterwards, `message y` is received and overwrites the value of `b` in the EDB FMU state.
Lastly, `message z` is received and overwrites the value of `a` in the EDB FMU state.
The example above implies that the value of `a` in `message x` is never outputted from the EDB FMU, since it has been overwritten by the value of `a` in `message z` within the same `fmi2DoStep` execution.

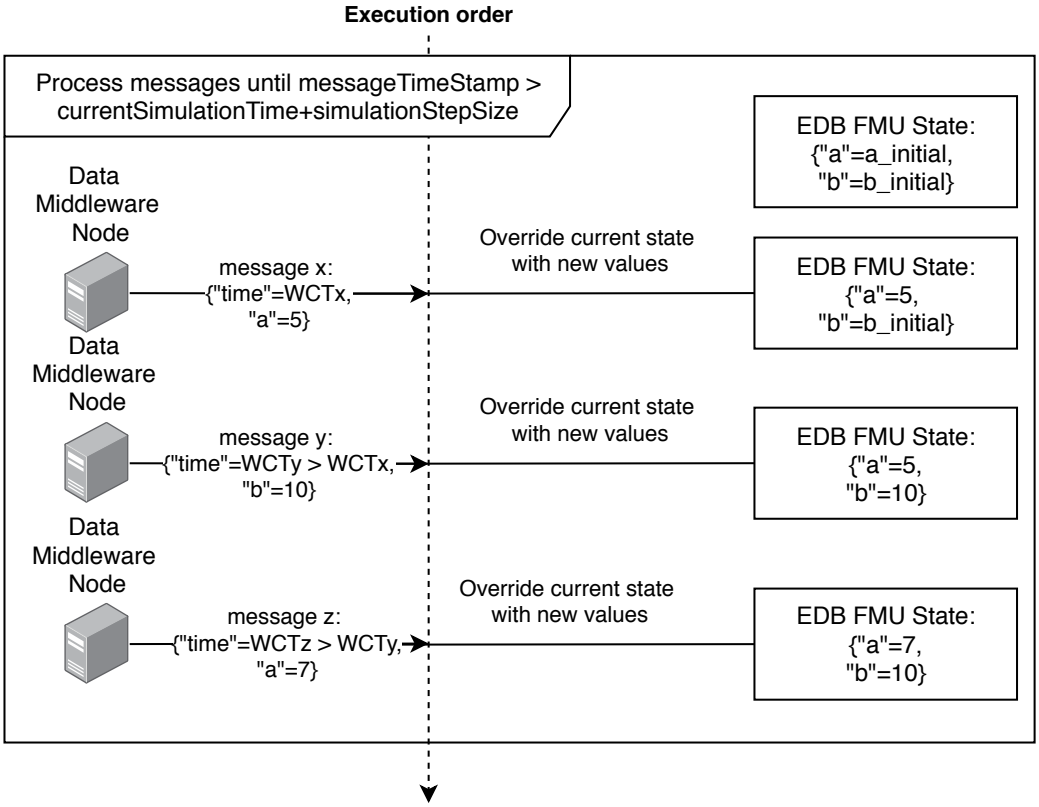

Figure 3: Data Handling in `fmi2DoStep`

4 Configuration

This section covers how to configure an EDB FMU based on the RabbitMQ implementation. The data format of the messages is JavaScript Object Notation (JSON), which is based on key/value pairs.

There are two key parts of the configuration: Configuring the data source and configuring the FMI outputs. The data source configuration specifies connection details related to the RabbitMQ server. The FMI outputs configuration specifies which key/value pairs of the messages to use.

This require changes to the FMU, but only to the static description file within the FMU. Thus, the precompiled binary still works, since it parses the static description file initially. Since RabbitMQ FMU has to parse the static description file it is necessary to store a copy of the model description file within the resources folder of the FMU.

4.1 Configuring the Data Source

The data source is configured via FMI parameters, as these can be set externally before `fmi2EnterInitializationMode`[1]. For this reason, it is possible to either preconfigure the parameters in the static description file in the resources folder of the FMU or set the values during execution of the FMU.

Listing 1 presents an example of configuring the data source. Notice the combination of `causality="parameter"` and `variability="fixed"` which implies that it is possible to set the values before `fmi2EnterInitializationMode`, as it is necessary since a start message is expected in the execution of `fmi2EnterInitializationMode` as described in section 2.

Listing 1: Data source configuration of RabbitMQ FMU

```
1 <ScalarVariable name="config.hostname" valueReference="0"
    ↳ variability="fixed" causality="parameter">
2   <String start="localhost"/>
3 </ScalarVariable>
4 <ScalarVariable name="config.port" valueReference="1"
    ↳ variability="fixed" causality="parameter">
5   <Integer start="5672"/>
6 </ScalarVariable>
```

[1]This does not hold for all scalar variables with parameter as causality. Check the FMI standard to ensure that you have the correct configuration.

```
<ScalarVariable name="config.username" valueReference="2"
    ↳ variability="fixed" causality="parameter">
   <String start="guest"/>
</ScalarVariable>
<ScalarVariable name="config.password" valueReference="3"
    ↳ variability="fixed" causality="parameter">
   <String start="guest"/>
</ScalarVariable>
<ScalarVariable name="config.routingkey" valueReference="4"
    ↳ variability="fixed" causality="parameter">
   <String start="linefollower"/>
</ScalarVariable>
<ScalarVariable name="config.communicationtimeout"
    ↳ valueReference="6" variability="fixed"
    ↳ causality="parameter" description="Network read time out
    ↳ in seconds">
   <Integer start="60"/>
</ScalarVariable>
```

4.2 Configuring the FMI Outputs

The scalar variables with `causality="output"` within the static description are used by RabbitMQ FMU to define which key/value pairs to extract from the messages that it receives. Thus, the name attribute of the output scalar variable defines the key, whose paired value should be outputted as the given scalar variable. Furthermore, the type of the value type must match the type of the output scalar variable. An example is presented in Listing 2 where a single output of type `Real` is defined. Note, that the output scalar variable must be added to the `Outputs` element based on index as required by FMI, see Listing 3. The name of the output scalar variable matches the key/value pair `"level"` of a message such as the one exemplified in Listing 4

Listing 2: FMI Outputs scalar variable configuration of RabbitMQ FMU

```
<!-- index=1 -->
<ScalarVariable name="level" valueReference="20"
    ↳ variability="continuous" causality="output">
   <Real />
</ScalarVariable>
```

Listing 3: FMI Outputs index configuration of RabbitMQ FMU

```
<Outputs>
   <Unknown index="1"/>
</Outputs>
```

Listing 4: Example of a JSON message.

```
1 {"time": 321, "level": 3.2}
```

5 Example Single Water Tank RabbitMQ

The example in this section concerns a water tank with constant inflow and a drain valve as depicted in fig. 4. The water level should stay within a parameterised minimum and maximum water level. This is a brief description, please see [MGP+17] for additional information on the models and co-simulation. Furthermore, historical data is passed in through RabbitMQ FMU. **The entire example including a docker-compose file for starting a local RabbitMQ Node is available at `https://github.com/INTO-CPS-Association/example-single_watertank_rabbitmq`.**

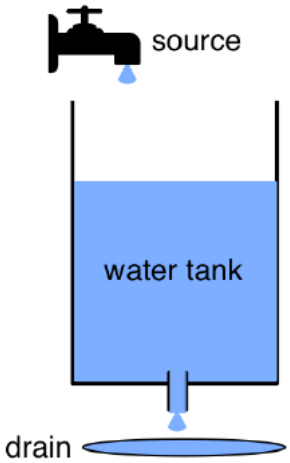

Figure 4: Water Tank

The example setup is shown in fig. 5 and consists of:

Maestro FMI Co-Simulation Engine

Tank FMU Models the physical part of the water tank and outputs the water level within the tank. It receives the state of the drain valve as input.

Controller FMU Models the controller of the drain valve. It outputs the state of the drain valve and receives the water level as input. Furthermore, it has the minimum water level and maximum water level as parameters

RabbitMQ Server runs a RabbitMQ node.

Data Contains historical water level data.

Python Reads the data file and inputs it to RabbitMQ.

RabbitMQ FMU Subscribes to RabbitMQ data.

Diff FMU Only part of the co-simulation for technical issues related to plotting outputs.

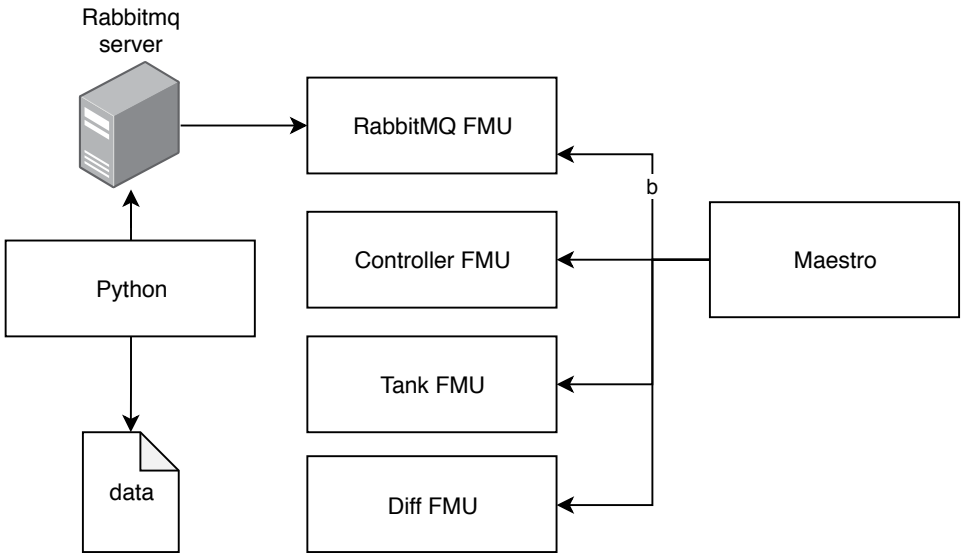

Figure 5: Water Tank

The result of executing the co-simulation is shown in TODO XX. The discrepancies between the Tank FMU output and the RabbitMQ FMU output is related to floating-point imprecision.

6 Example Line Following Robot with Deployed System and RabbitMQ - IN PROGRESS

The example in this section concerns a line following robot (LFR). An overview of the example is presented in fig. 6.

The deployed LFR publishes its sensor readings and actuator values to RabbitMQ Node Queue 1. However, the deployed LFR is unaware of its position on the track and the time. For this reason, a camera is used along with image recognition to detects its position. The Log-Translator correlates the data from the deployed LFR and the detection before publishing it to RabbitMQ Node Queue 2. As RabbitMQ FMU subscribes to RabbitMQ Node Queue 2 it receives the messages. The Controller FMU and the Physical FMU make up the models of the LFR, and the modelled LFR is aware of its position on the track.

Thus, it is possible to realise the digital twin of the deployed LFR.

The example including a docker-compose file for starting a local RabbitMQ Node is IN PROGRESS and you can follow it at `https://github.com/INTO-CPS-Association/example-DT-line_following_robot`.
See Issues in the repository for future work on the example.

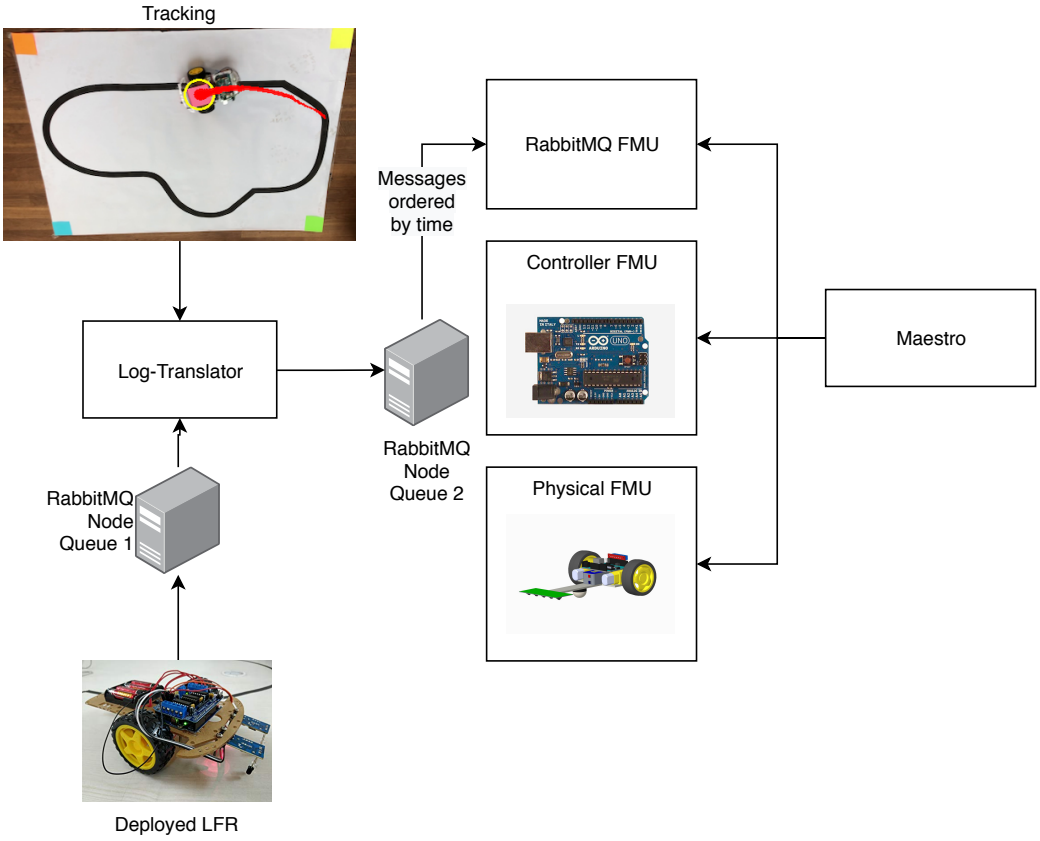

Figure 6: Water Tank

7 Future Work

This section is not to be considered final and is subject to change in future revisions.

Some of the immediate future work is making the current research available at the INTO-CPS Association github: `https://github.com/INTO-CPS-Association/`.

Other immediate future work is related to the LFR example, which was described in section 6, and it is described at: `https://github.com/INTO-CPS-Association/example-DT-line_following_robot/issues`

References

[MGP+17] Martin Mansfield, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 3. Technical report, INTO-CPS Deliverable, D3.6, December 2017.

[Mod19] Modelica Association Project FMI. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads`, October 2019.

A List of Acronyms

XML	Extensible Markup Language
JSON	JavaScript Object Notation
EDB	External Data Broker
FMU	Functional Mock-up Unit
FMI	Functional Mock-up Interface