The Functional Mock-up Interface 3.0 - New Features Enabling New Applications

Andreas Junghanns[1] Torsten Blochwitz[2] Christian Bertsch[3] Torsten Sommer[4]
Karl Wernersson[5] Andreas Pillekeit[6] Irina Zacharias[6] Matthias Blesken[6] Pierre R. Mai[7]
Klaus Schuch[8] Christian Schulze[9] Cláudio Gomes[10] Masoud Najafi[11]

[1]Synopsys, Germany, `Andreas.Junghanns@synopys.com`
[2]ESI ITI GmbH, Germany, `Torsten.Blochwitz@esi-group.com`
[3]Robert Bosch GmbH, Germany, `Christian.Bertsch@de.bosch.com`
[4]Dassault Systemes GmbH, Germany, `Torsten.Sommer@3ds.com`
[5]Dassault Systemes AB, Sweden, `Karl.Wernersson@3ds.com`
[6]dSPACE GmbH, Germany `{APillekeit,IZacharias,MBlesken}@dspace.com`
[7]PMSF IT Consulting, Germany, `pmai@pmsf.de`
[8]AVL List GmbH, Austria, `Klaus.Schuch@avl.com`
[9]TLK-Thermo GmbH, Germany, `c.schulze@tlk-thermo.com`
[10]Aarhus University, Denmark, `claudio.gomes@ece.au.dk`
[11]Altair, France, `masoud@altair.com`

Abstract

The Functional Mock-up Interface (FMI) (Modelica Association 2021b) is a tool independent standard for the exchange of dynamic models and for co-simulation. FMI 2.0, released in 2014, is recognized as the de-facto standard in industry for exchanging models and tool coupling, and is currently supported by more than 160 simulation tools. Version 3.0 of the standard brings many new features that allow for advanced co-simulation algorithms and new use cases such as packaging and simulation of highly accurate virtual Electronic Control Units (vECUs). Besides Model-Exchange and Co-Simulation, a third interface type, Scheduled Execution, is defined for purely discrete, RTOS-like, simulation and supports preemption. Clocks allow the synchronization of events between Functional Mock-up Units (FMUs) and the importer. There is better support for data types including binary data and arrays. Advanced co-simulation approaches are enabled by intermediate variable access between communication points and allowing event handling. The composition of systems from FMUs is simplified by terminals that can bundle multiple signals. The concept of layered standards allows the extension of the FMI standard.
Keywords: FMI, FMU, Functional Mock-up Interface

1 Motivation

FMI 1.0 (Blochwitz 2011) and FMI 2.0 (Blochwitz 2012) were successfully adopted by industry and are currently supported by more than 160 simulation tools (Modelica Association 2021c). For many years stability was an important success factor of FMI, resulting in maintenance releases of FMI 2.0. However, it became clear that new use cases require improved capabilities that are addressed by the new version of the standard (Modelica Association 2021a), summarized next.

Virtual Electronic Control Units (vECUs). The ability to package control code into Functional Mock-up Units (FMUs) required some workarounds in FMI 2.0. With FMI 3.0, virtual electronic control units (vECUs) can be exported as FMUs in a more natural way using the following new and/or improved features: Terminals (subsection 3.3), clocks (subsection 3.6), new integer types and the new binary type (subsection 3.1), array variables and structural parameters (subsection 3.2), and the new interface type Scheduled Execution (subsection 2.3).

Advanced Co-Simulation. FMI 3.0 introduces, in its Co-Simulation interface type, the Event Mode (subsection 3.4), early return from `fmi3DoStep` (subsection 3.4), and the Intermediate Update Mode (subsection 3.5).

Improved Event Handling across FMUs. The new version of FMI provides an API to enable more flexible event handling and communication: the Synchronous Clocks API (subsection 3.6). For scenarios that are driven by events (e. g., supervisor control systems, engine control systems triggered by crankshaft angle sensors), the clocks API allows FMUs to communicate to the importer detailed information about the timing and cause of events. Moreover, the exact timing that events should happen is communicated unambiguously between FMU and importer, bypassing floating point representation issues. Most of these features are optional to not exclude simulation domains where such features are out of scope and tools that are not able to implement these optional features.

AI models. In Machine Learning and Artificial Intelligence (AI) new modeling and model training frameworks emerge. More and more, the created models shall interact with established modeling and simulation tools. FMI is a natural means to encapsulate and exchange AI-models with them. This can also lead to hybrid models formed of both physics-based- and AI-models. In order to enable the efficient training of AI-models encapsulated as FMUs, adjoint derivatives are needed, see subsection 3.7.

The following section gives an overview of the different interface types, and the main use cases they apply to. Then, section 3 details the new features. Section 4 introduces some examples that use the new features, and section 5 discusses the measures taken to improve the quality of the standard. Finally, section 6 concludes.

2 Interface Types

FMI 3.0 defines three main interface types: Co-Simulation (CS), Model Exchange (ME), and Scheduled Execution (SE). An FMU may implement one or more of the three interface types. It is a ZIP archive containing: an XML file, describing the model variables and structure; binary and/or source code implementations of the FMI API of the supported interface types; miscellaneous resources; and other related data.

All interface types share common functionality, such as the way variables and clocks are declared/interacted with, or common optional features like store/restore the complete FMU state, or definitions of terminals and icons.

Figure 1 ranks the different interface types according to their simplicity and flexibility trade-offs.

An implementation that interacts with an FMU using the FMI API is called importer.

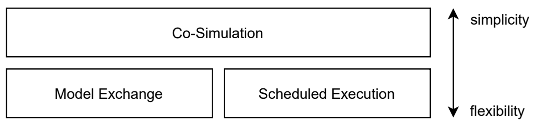

Figure 1. Comparison of interface types

2.1 Model Exchange

The Model Exchange interface exposes a simulation model as a hybrid ordinary differential equation (ODE) to a solver of an importer. Models are described by differential, algebraic and discrete equations, interleaved with time-, state- and step-events. The integration algorithm of the importer is responsible for advancing time, computing state variables, handling events, etc. Figure 2 shows the data flow for Model Exchange.

2.2 Co-Simulation

The Co-Simulation interface is designed both for the coupling of simulation tools, and the coupling of subsystem models, exported by their simulators together with their solvers as runnable code. The data exchange between

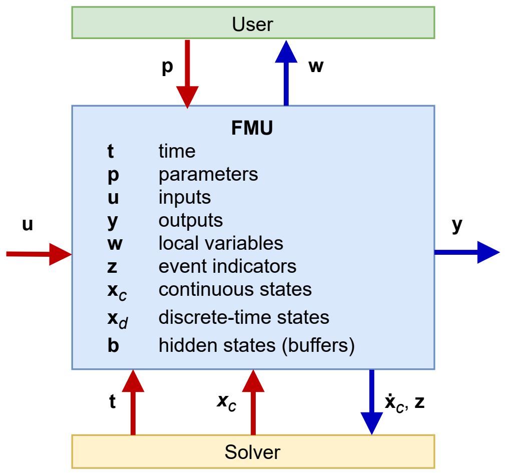

Figure 2. Schematic view of data flow between user, the solver of the importer and the FMU for Model Exchange.

FMUs is restricted to discrete communication points. In the time between two communication points the subsystem inside an FMU is solved independently by internal means. For FMI for Co-Simulation, the co-simulation algorithm is shielded from how the FMU advances time internally.

Figure 3 shows the data flow for Co-Simulation.

2.3 Scheduled Execution

The Scheduled Execution interface exposes individual model partitions (e. g., tasks of a control algorithm), to be orchestrated by a scheduler provided by the importer. The scheduler is responsible for advancing the overall simulation time, activating time-based and triggered clocks (for an explanation of clocks see subsection 3.6) for all exposed model partitions of a set of FMUs, and to activate the respective model partition. The Scheduled Execution interface addresses simulation use cases with the following properties, that typically hold when the importer has to communicate with external event sources/sinks that operate on independent individual timing schemes (e. g., real hardware, controller tasks on simulated or real controller units):

1. at any time (even for unpredictable events), an event shall be signaled to an FMU,
2. communication constraints (e. g., execution times, communication deadlines) that are not apparent at FMU simulation level but lead to timing requirements have to be fulfilled by the simulation algorithm,
3. priority information provided by the FMUs has to be evaluated and merged to an overall priority for available model partitions,
4. data shall move between the different FMU model partitions for the same or next activation time.

To address these properties, the Scheduled Execution interface provides support for preemptive multitasking: concurrent computation of model partitions of an FMU

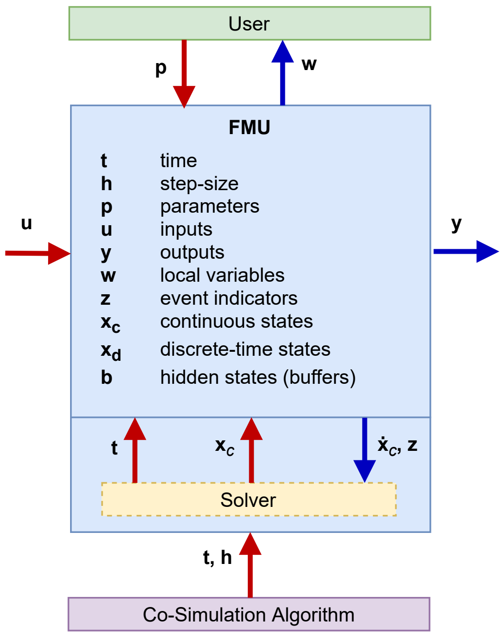

Figure 3. Schematic view of data flow between user, the co-simulation algorithm of the importer and the FMU for Co-Simulation. Compared to Figure 2, the solver is part of the FMU, and not part of the importer.

(i. e., a support of multiple rates) on a single computational resource (e. g., CPU-core). In the rare cases that the FMU has to be able to restrict the preemption for particular code sections, lock/unlock callback functions are provided by the importer. Note that cooperative multitasking for model partitions of an FMU is currently not covered by the interface description, and parallel computation of model partitions is therefore not part of the FMI 3.0 API. However, an FMU may internally use parallel computation on multiple cores, but this results in a binding to a supported operating system.

The FMU must declare the priorities of its model partitions, enabling a global computation order and preemption policy for model partitions across FMUs.

The Scheduled Execution interface has a different timing concept compared to FMI for Co-Simulation: a scheduler's activation of a model partition will compute the results of the model partition defined by an input clock for the current clock tick time t_i (and not for $t_i + h_i$, as defined for `fmi3DoStep` in FMI for Co-Simulation). This is required to handle activations of triggered input clocks which may tick at a time instant that is unpredictable for the simulation algorithm. Typically, hardware I/O or vECU software events belong to this category.

Figure 4 shows the data flow for Scheduled Execution.

3 New Features

3.1 Data Types

FMI 1.0 and FMI 2.0 used a minimal number of numeric data types for interface variables: `fmi2Boolean`,

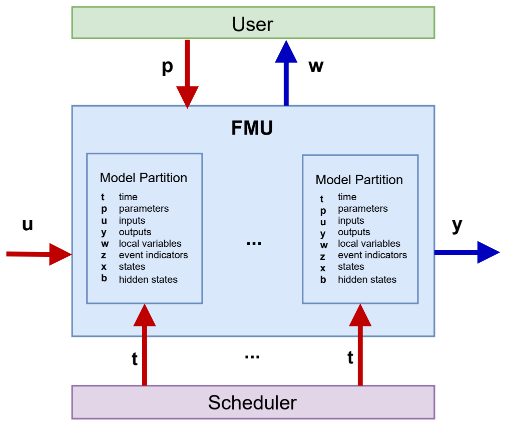

Figure 4. Schematic view of data flow between user, the scheduler of the importer and the FMU for Scheduled Execution

`fmi2Integer` and `fmi2Real`. When packaging numeric codes representing physical processes, these types and the fmi2String type, were sufficient.

System simulation has enlarged its focus to include controller code (vECUs) to achieve high-quality cyber-physical systems simulations. While packing vECUs as 1.0 or 2.0 FMUs has been practiced since 2010, the restrictions in interface data types introduced noticeable overhead for conversion and copying. Moreover, new automotive applications for system simulation, like Advanced Driver Assistant Systems and Autonomous Drive system components exchange more and more non-numeric data, like object lists, images or even video streams.

Therefore, FMI 3.0 now supports a large set of integer types (signed and unsigned, from 8 to 64 bit) and both 32- and 64-bit floating-point variables. Moreover, binary variables have been introduced to allow the efficient exchange of non-numeric values. Binary variables can be attributed in the `modelDescription.xml` with a `mimeType` to allow proper interpretation of their content.

3.2 Array Variables

In former versions of FMI, only scalar variables were supported; array variables had to be expressed using naming conventions. FMI 3.0 supports array variables natively: Each variable (defined in the `modelDescription.xml`) can have a constant number of dimensions (thus making the variable a multidimensional array variable). The size of a dimension can either be constant or may reference a structural parameter.

A simple example for using arrays would be a generic matrix multiplication FMU. Such an FMU would expose two structural parameters, defining the size of the 1-dimensional input variable, the size of the 1-dimensional output variable and the sizes of the 2-dimensional array (matrix) parameter variable.

A structural parameter can be, like any other parameter, `constant`, `fixed`, or `tunable`. Changes to structural parameters are restricted to special simulation modes:

- Configuration Mode allows changes to `fixed` and

`tunable` structural parameters and can be reached from the mode Instantiated (before Initialization Mode).
- Reconfiguration Mode allows changes to `tunable` structural parameters. This mode can be reached from Event Mode in Model Exchange, from Step Mode in Co-Simulation and from Clock Activation Mode in Scheduled Execution.

Hence, the sizes of the matrix FMU example do not need to be "constant" after the FMU got generated but could be "fixed" after the FMU gets instantiated or even "tuned" during simulation.

Note that changing the value of a structural parameter might also change the number of continuous states or event indicators.

While the primary use case for structural parameters is for arrays with variable dimension sizes, there can be structural parameters that are not used in dimension elements, e. g., an `fmi3String` containing a file name to read data from.

FMI 3.0 defines serialization orders for accessing array variables (as well as corresponding derivatives or dependencies).

3.3 Terminals and Icons

Terminals define semantic groups of variables to ease connecting compatible signals on system level. This definition adds an additional layer to the interface description of the FMUs. It does not change the causality of the variables (i. e., inputs and outputs), but enables the definition of physical and bus-like connectors that require special handling on the system level by the importer (e. g., bus frames, flow and stream variables). Terminals can contain terminals to form hierarchies.

The matching rule of a terminal describes the rules for variable matching in a connection of terminals. There are three predefined matching rules: `plug`, `bus`, and `sequence`. The terminal kind can be used to define domain specific member variable sequences, member names and order, or high level restrictions for connections. A member variable always refers to a variable declaration in the `ModelVariables` element. The variable kind of a terminal member variable defines how the connection of this variable has to be implemented. There are three predefined matching rules: `signal` (i. e., common signal flow), `inflow` and `outflow` (i. e., Kirchhoff's current law). Finally, the concept of stream connectors is utilized (Rüdiger Franke et al. 2009). The `TerminalStreamMemberVariable` is used for variables which fulfill the balance equation for transported quantities.

To define domain-specific terminals, additional information, like the specific physical meaning of a `TerminalMemberVariable` or sign conventions, must be standardized. The FMI 3.0 standard itself does not define such domain specific terminals but enables other (layered) standards (see section 3.7) to do so. The XML attributes `matchingRule`, `terminalKind` and `variableName`, are defined as `xs:normalizedString` and not as `xs:enumeration` which would restrict their extensibility. The FMI 3.0 specification defines a certain set of values for these attributes (e. g., `"signal"` or `"inflow"`) and a dedicated semantics. Other standards might introduce other values and other semantics. Importers which do not understand such definitions can ignore them and use the traditional input/output approach specified by the `causality` attribute of FMI variables. So if a specific vendor definition is not supported, then the importer can ignore the terminal definition and rely on the information in the `ModelVariables` element.

The graphical representation of the FMU icon can be provided as png file. Additionally a coordinate system, the FMU icon extent, and the placement of each terminal can be defined. The graphical representation of each terminal is also provided as png file. In addition to the png files, svg files can be provided for high quality rendering.

Both, the terminal definitions and the graphical representations are defined in the optional XML file `terminalsAndIcons.xml`.

3.4 Event handling in Co-Simulation

The Co-Simulation interface of FMI 1.0 and FMI 2.0 is popular because many different simulation mechanisms can be abstracted into one function call `fmi2DoStep` to let the FMU execute one communication time step. FMI 3.0 extends the Co-Simulation interface with a number of mechanisms to more flexibly control execution of the FMU over time.

Event Mode – The importer can interrupt the Step Mode to transition the FMU into Event Mode. In Event Mode a different set of equations is active inside the FMU and discrete variables may change their values. The importer can solve algebraic loops of the system the FMU is part of and may step the FMU through a series of super-dense time instants, each such step potentially using a different discrete state of the FMU.

Early Return – In order to allow for larger h_i time steps in `fmi3DoStep` calls, importer and FMU must be able to interrupt such long `fmi3DoStep` before reaching $t_i + h_i$, in case something "special" happened:

1. The FMU can return from `fmi3DoStep` early, announcing an internal event and requesting a transition to Event Mode.
2. The importer can request the FMU to return early from `fmi3DoStep` during Intermediate Update Mode (see next Section) to prevent the FMU from computing beyond an event recently discovered by the importer (e. g., triggered by another FMU).

The FMU indicates via the capability flag `hasEventMode` if it supports Event Mode.

The importer informs the FMU via argument `eventModeUsed` and `earlyReturnAllowed` of function `fmi3InstantiateCoSimulation` if it supports event handling.

3.5 Intermediate Update Mode

The order of the error in a co-simulation is dominated by the order of the errors made due to the approximation of inputs (Arnold, Clauß, and Schierz 2014). At the same time, higher order input approximation schemes may lead to instabilities, depending on the system being simulated. As such, it is important to provide some degree of control over the input approximations being performed, and allow the importer to obtain some of the intermediate outputs of the FMU. Additionally, the importer may change continuous and discrete input variables between `fmi3DoStep` calls in a way that is hard for the FMU to predict. This causes problems such as excessively small internal solver stepping (due to the discontinuities introduced at communication times) and loss of accuracy (Busch 2016).

FMI 3.0 introduces the Intermediate Update Mode to alleviate these issues, allowing the Importer and FMU to exchange intermediate values for variables. The FMU can call back into the importer to enter this Intermediate Update Mode and ask the importer to update its continuous input variables and allow it to query its continuous output variables (e. g., to supply other FMUs with updates to their inputs). This mechanism replaces `fmi2SetRealInputDerivatives` for input interpolation. This callback mechanism allows the FMU to maintain its internal solver state while new continuous inputs are being set. The FMU can hint to the importer to keep the changes to the continuous input variables within a certain smoothness (see `recommendedIntermediateInputSmoothness`) to optimize convergence.

The Intermediate Update Mode serves a number of other purposes as well:

1. FMUs can inform the importer about pending events.
2. The importer can ask the FMU to return early from an `fmi3DoStep` (see previous Section).
3. In Scheduled Execution, the FMU can inform the importer/scheduler about a clock activation.

The FMU signals via the capability flag `providesIntermediateUpdate` if it supports this feature. The importer provides the pointer to the callback function via argument `intermediateUpdate` of functions `fmi3InstantiateCoSimulation` and `fmi3InstantiateScheduledExecution` as non-NULL if it supports this feature.

3.6 Clocks

System simulation requires the coordination of events across simulation components, in both Model Exchange and Co-Simulation. If these components are packaged as 1.0 or 2.0 FMUs, the importer and FMUs need to use floating point time and epsilon environments with all the known issues to locate such global events. Rüdiger Franke et al. (2017) proposed how to introduce clocks in FMI.

In FMI 3.0, clocks are introduced to allow precise coordination of global events, see Cláudio Gomes et al. (2021). An FMI clock is a special variable that can be active or inactive. When active, the corresponding model partition (a set of equations associated to a clock) becomes active while in Event Mode or Clock Activation Mode.

FMU variables that change only when a specific clock ticks are called clocked variables and are assigned to this clock in the `modelDescription.xml`.

With clocks synchronized across FMUs, algebraic loops can now be solved properly during such global events.

FMI 3.0 distinguishes between time-based clocks and triggered clocks. The latter are raised when something unexpected (e. g., a state event) happens and can be connected to other triggered clocks. The importer forwards that clock activation to the triggered input clocks during Event Mode or Clock Activation Mode.

Time-based clocks come in a few different flavors (see standard document for details) and all require the importer to determine the proper time instant when to activate such clocks - even if the FMU receiving such a time-base input clock defines the period or next event time itself. This is an important distinction to note: the FMU defines the period or next activation, but the importer has the final say at which time instant to actually activate the clock. This is especially important for fixed-step solvers where some flexibility might be required to transition events to one of the communication points.

3.7 Adjoint derivatives

FMI 3.0 offers an additional interface function to calculate partial derivatives. While directional derivatives calculating $v_{\text{sensitivity}} = \mathbf{J} \cdot v_{\text{seed}}$ for the Jacobi matrix $\mathbf{J}$ where already supported in FMI 2.0, now also adjoint derivative calculation is supported by the new interface function `fmi3GetAdjointDerivative`, calculating $v^T_{\text{sensitivity}} = v^T_{\text{seed}} \cdot \mathbf{J}$. They are used, e. g., in AI frameworks, where they are called “vector gradient products” (VGPs). There adjoint derivatives are used in the backpropagation process to perform gradient-based optimization of parameters using reverse mode automatic differentiation (AD). Typically, reverse mode automatic differentiation is more efficient for this use case than forward mode AD, as explained in (Baydin et al. 2015).

3.8 Support for Layered Standards

In order to enable the backward-compatible extension of the FMI standard in minor releases and between minor releases, the FMI project intends the use of the layered standard mechanism to introduce new features in a fully backward-compatible and optional way. A layered standard defines extensions to the base FMI standard by specifying either standardized annotations, standardized extra files in the FMU, and/or support for additional MIME types, e. g., for the interpretation of variables of type `fmi3Binary`.

A layered standard can include a single or combined set of extension mechanisms from this set. The layered standard is thus considered to be layered on top of the definitions and extensions mechanisms provided by the respective FMI base standard.

Layered standards can fall into three categories:

1. Layered standards defined by third parties, without any representations by the FMI project for their suitability or content, or even knowledge by the FMI project about their existence.
2. Layered standards defined by third parties that are endorsed by the FMI project and listed on the FMI project website.
3. Layered standards can be defined/adopted and published by the FMI project itself, making them FMI project layered standards.

Layered standards that have achieved enough adoption or importance to be included into the base standard set could be incorporated into a new minor or major release version of the base standard as an optional or mandatory appendix, making support for this layered standard optional or required for conformance with the newly published minor release version of the base standard.

Examples for layered standards currently developed by the FMI Project will support XCP (ASAM 2021) and Automotive Networks. Further layered standards could define standardized terminals for certain domains.

3.9 Build Configuration for Source Code FMUs

To better support the exchange of FMUs with source code implementations, FMI 3.0 introduces build configurations consisting of an XML document that specifies a set of source files and abstracted build information, like preprocessor definitions, include paths or library dependencies that are needed for building the supplied source code. These can be cross-platform or platform-specific, which gives the importer the ability to choose the correct build configuration for a certain platform.

Here is an example for a build configuration:

```xml
<BuildConfiguration modelIdentifier="
    PlantModel" description="Build
    configuration for desktop platforms">
<SourceFileSet language="C99">
  <SourceFile name="fmi3Functions.c"/>
  <SourceFile name="solver.c"/>
</SourceFileSet>
<SourceFileSet language="C++11">
  <SourceFile name="model.c"/>
  <SourceFile name="logging/src/logger.c"/>
  <PreprocessorDefinition name="FMI_VERSION"
      value="3"/>
  <PreprocessorDefinition name="LOG_TO_FILE"
      optional="true"/>
  <PreprocessorDefinition name="LOG_LEVEL"
      value="0" optional="true">
    <Option value="0" description="Log infos,
        warnings and errors"/>
    <Option value="1" description="Log
        warnings and errors"/>
    <Option value="2" description="Log only
        errors"/>
  </PreprocessorDefinition>
  <IncludeDirectory name="logging/include"/>
</SourceFileSet>
<Library name="hdf5" version="&gt
    ;=1.8,!=1.8.17,&lt;1.10" external="true"
    description="HDF5"/>
```

4 Examples

In this section, we show some examples where the features, introduced in the previous section, are used. Some of these examples are being developed as reference FMUs, with their source code made available online[1].

4.1 Supervisory Control System

We start with an example that highlights the use of new features of FMI for CS. Consider the example scenario shown in Figure 5, where the FMUs are connected in a typical feedback control loop, with a monitoring and adaptation loop, and the plant has been decoupled into two FMUs. Using the new data types, structured information can be communicated from the controller to the supervisor, and using array variables or terminals, the connections between the controller and plant can be simplified. Thanks to the intermediate value update and early return mechanisms, the rate of sampling of the Controller can be decoupled from the rate of sampling of the supervisor, which in turn can be decoupled from the step size used in the co-simulation.

The intermediate value update enables setting inputs and accessing outputs between the communication points. It can be used to implement advanced co-simulation algorithms to increase stability and reduce the coupling errors between the two plant FMUs. Some examples are given below:

1. The plant inputs can be extrapolated ensuring continuity of signals (Busch 2016).
2. If a conserved quantity such as energy is transported between the plant FMUs, then the additional information enables a reduction of the coupling errors. One available algorithm is the nearly energy-preserving coupling element (Benedikt et al. 2013; Sadjina et al. 2017).
3. Transmission Line Modelling (TLM) systems contain TLM connections which can be interpreted as physically-motivated delayed connections. So introducing a delay between the FMUs, trajectories instead of scalars can be exchanged between the plant FMUs to improve stability and performance (Fritzson, Ståhl, and Nakhimovski 2007; Ochel et al. 2019).

For more advanced co-simulations algorithms, we refer the reader to Cláudio Gomes et al. (2018, Section 4).

[1] `https://github.com/modelica/Reference-FMUs`

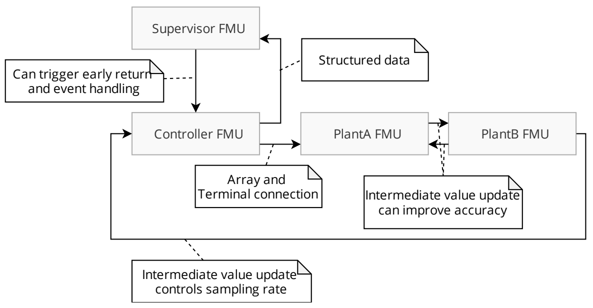
Supervisor FMU
Structured data
Can trigger early return and event handling
Controller FMU
PlantA FMU
PlantB FMU
Array and Terminal connection
Intermediate value update can improve accuracy
Intermediate value update controls sampling rate

Figure 5. Example supervisory control system and how the new features of FMI CS can improve its simulation.

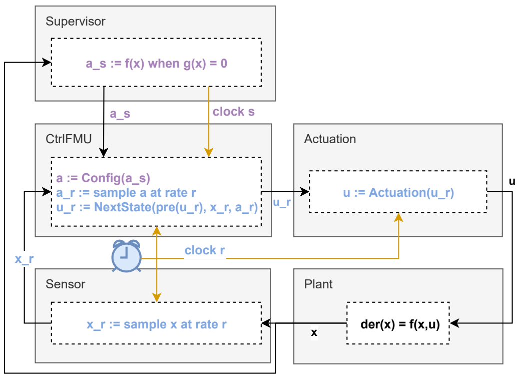

Figure 6. Example clocked control system.

Finally, the Supervisor FMU may reconfigure the Controller FMU when a certain condition is met. When this happens, the Supervisor FMU may signal that stepping phase needs to be halted, and event handing is needed.

4.2 Clocked System

When it is important to make explicit the sampling rate of different subsystems, the Synchronous Clocks API can be used. To illustrate this, consider Figure 6, that shows a variant of the scenario of Figure 5, except the controller FMU has been partitioned across different FMUs. The figure sketches the `CtrlFMU` equations, but note that the importer has no access to these (it can only query the FMU for the values of the output variables). The `CtrlFMU`, every $1/r$ seconds (with r denoting both a clock r and its frequency), gets a sample from the `Plant` (produced by the `Sensor`), and calculates its next state, based on the previous state `pre(u_r)`, the sampled value `x_r`, and some configuration parameter `a` that is calculated by the `Supervisor`. The latter, depending on the `Plant` dynamics, the sampling rate of which we ignore, may decide to reconfigure the `Controller`. By introducing a triggered input clock `s` and a time-based input clock `r`, it is made clear who is responsible for the unambiguous activation of the clocks: the `Supervisor` controls `s`, and the importer controls when to activate `r` (even though, depending on the clock attributes, the `CtrlFMU` may recommend a sample rate that the importer will then obey). Furthermore, no approximate floating point comparisons are needed to know which equations have to be active when entering Event Mode.

We refer the reader to (Cláudio Gomes et al. 2021) and the FMI standard, for more details about Synchronous Clocks.

4.3 Terminals

This section illustrates an exemplary terminal definition of an electrical pin in the FMU XML file.

```
<Terminal name="Pin1" matchingRule="plug">
  <TerminalMemberVariable
    variableKind="inflow"
    memberName="i"
    variableName="Current" />
  <TerminalMemberVariable
    variableKind="signal"
    memberName="v"
    variableName="Voltage" />
</Terminal>
```

"`Voltage`" and "`Current`" reference to the following FMU-variables:

```
<Float64
  name="Current"
  valueReference="2"
  description="Current output"
  variability="continuous"
  causality="output" />
<Float64
  name="Voltage"
  valueReference="1"
  description="Voltage input"
  variability="continuous"
  causality="input"
  start="0" />
```

The `variableKind` attribute indicates to the importer that Kirchhoff's current law should be applied to the variable `Current`, while `Voltage` should be treated as common signal.

A Modelica tool, for example, can use this XML description to generate a `connector` which automatically matches to `Modelica.Electrical.Analog.Interfaces.Pin` of the Modelica Standard Library since the same names for the member variables are used and "`i`" is marked as flow variable. Similar mechanisms are possible with VHDL-AMS or other tools which support a terminal or bus concept which goes beyond single signals.

Even though the FMU can be imported into acausal tools/languages such as Modelica, the FMU itself is causal. An acausal model can be used to generate several FMUs with different computational causality. The computational causality of the FMU is defined by the causality attributes in the XML file. In the example above "`current`" is always an output of this specific FMU. Cases where the importer prefers a different computational causality than provided by the FMU have to be handled by the importer.

4.4 Virtual Electronic Control Unit

The following example illustrates how FMI 3.0 features can be used for packaging vECUs inside FMUs. It is a reduced version of the example used in the layered standard proposal for automotive network communication.

This layered standard proposal uses clocks, binary variables, naming conventions for variable names and terminals to use FMI 3.0 mechanisms as transport layer for automotive network communication between simulation components. Clocks describe the timing of their respective network frames, to exactly communicate the sending of each frame. Terminals are used to describe the composition of these network frames from PDUs and signals, hierarchically, allowing simplified handling of entire signal groups in system composition tools. Binary signals are used to represent frames in their raw network specific encoding, serializing internal PDUs and signals. For consistent encoding and decoding of these binary signals, standardized network description files are included inside the FMU in the `/extra` directory. Referencing existing standard description files allows reusing existing tools for both exporting and importing such FMUs, while ensuring the same semantics are used for both sender and receiver of signals encoded according to these standards.

The example shows how to describe a CAN message that updates 2 signals, each represented as a `Float32` variable. Naming conventions, described in the layered standard, can be used to match the signals, the corresponding binary variable representing the raw frame data and the clock variable determining the timing of the CAN message (here `POWERTRAIN::tcuSensors_FRAME` and their corresponding triggered `Clock` variable `POWERTRAIN::tcuSensors_CLOCK`). The triggered clock variable controls the time at which a message is set, and should be connected to another clock at the source of the message.

```
<fmiModelDescription fmiVersion="3.0-alpha.6"
  modelName="Network4FMI"
  instantiationToken="Network4FMI">
  <ModelVariables>
    <Float32 name="POWERTRAIN::tcuSensors::
        tcuSensors::vCar"
      valueReference="1001" causality="input"
      variability="discrete" start="0" clocks="
         1004"/>
    <Float32 name="POWERTRAIN::tcuSensors::
        tcuSensors::oilTemp"
      valueReference="1002" causality="input"
      variability="discrete" start="20" clocks=
         "1004"/>
    <Binary name="POWERTRAIN::tcuSensors_FRAME
        "
      valueReference="1003" causality="input"
      variability="discrete" clocks="1004"/>
    <Clock name="POWERTRAIN::tcuSensors_CLOCK"
      valueReference="1004" causality="input"
      variability="clock" interval="triggered"/
         >
...
  </ModelVariables>
</fmiModelDescription>
```

```
<fmiTerminalsAndIcons fmiVersion="3.0-alpha6"
    >
  <Terminals>
    <Terminal terminalKind="bus" name="
        POWERTRAIN" matchingRule="bus"
      description="Powertrain CAN bus defined
          with dbc file">
      <Terminal terminalKind="frame" name="
          tcuSensors" matchingRule="bus">
        <TerminalMemberVariable variableKind="
            signal"
          variableName="POWERTRAIN::
            tcuSensors_FRAME" />
        <TerminalMemberVariable variableKind="
            signal"
          variableName="POWERTRAIN::
            tcuSensors_CLOCK" />
        <Terminal terminalKind="pdu" name="
            tcuSensors" matchingRule="bus">
          <TerminalMemberVariable variableKind="
              signal"
            variableName="POWERTRAIN::tcuSensors
                ::tcuSensors::vCar"
            memberName="vCar" />
          <TerminalMemberVariable variableKind="
              signal"
            variableName="POWERTRAIN::tcuSensors
                ::tcuSensors::oilTemp"
            memberName="oilTemp" />
        </Terminal>
      </Terminal>
    </Terminal>
    ...
  </Terminals>
  <Annotations>
    <Annotation type="ECU" />
  </Annotations>
</fmiTerminalsAndIcons>
```

4.5 Scheduled Execution

In order to illustrate the preemption support, we consider the example of a single FMU, illustrated in Figure 7, where an FMU declares three input clocks and one output clock. Each input clock, when activated, instructs the importer, that acts as a task scheduler, to execute the corresponding model partition as soon as possible.

A model partition, or just partition, represents code that should be scheduled (e.g on a real-time simulator or offline simulator running real-time scenarios) as soon as an input clock ticks. Partitions contain arbitrary code that reads the inputs of the FMU, writes to the FMU's local variables (which can be shared among tasks) and outputs, and can trigger other clocks or update their interval. The inputs to each partition are set by the importer immediately before executing that partition. In Figure 7, u_m^c's partition reads and writes the shared variable x_m, and either updates the interval of v_m^c or ticks y_m^c.

In Figure 7, input clock u_m^c ticks every 10 ms and w_m^c ticks every 50 ms, therefore, every 5th tick, both clocks will tick simultaneously. When that happens, the scheduler needs to know whose task has the highest priority. As a result, the FMU needs to declare a priority level for each input clock. In Figure 7, u_m^c's task (the one executing Partition 1) should be executed before w_m^c's (Partition 3).

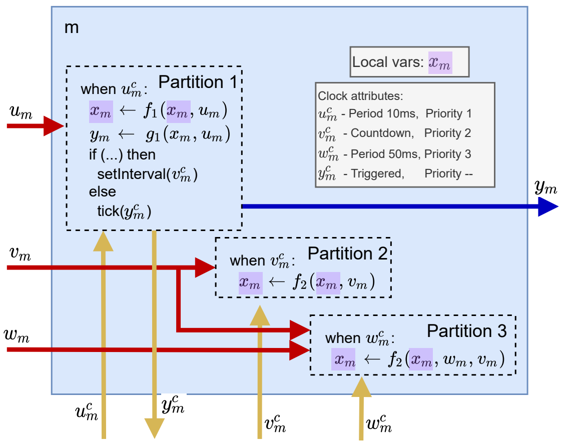
m
Local vars: x_m
Clock attributes:
u_m^c - Period 10ms, Priority 1
v_m^c - Countdown, Priority 2
w_m^c - Period 50ms, Priority 3
y_m^c - Triggered, Priority --
Partition 1
when u_m^c:
$x_m \leftarrow f_1(x_m, u_m)$
$y_m \leftarrow g_1(x_m, u_m)$
if (...) then
setInterval(v_m^c)
else
tick(y_m^c)
Partition 2
when v_m^c:
$x_m \leftarrow f_2(x_m, v_m)$
Partition 3
when w_m^c:
$x_m \leftarrow f_2(x_m, w_m, v_m)$
u_m
v_m
w_m
y_m
u_m^c
y_m^c
v_m^c
w_m^c

Figure 7. Example FMU implementing the SE interface.

Because tasks can be preempted, in rare cases, the FMU has to be able to restrict the preemtion for particular sections, such as updating a shared variable. As such, the FMU must inform the importer of when it should not be interrupted, to avoid race conditions. Since partitions can trigger and update the interval of other clocks, the FMU may use the Intermediate Update Mode, in the middle of the calculation of a partition, to inform the importer that a clock is about to tick or has a new interval, so that the importer can schedule the corresponding tasks.

Figure 8 illustrates a possible execution trace of the tasks corresponding to the partitions declared in Figure 7. At the initial wall-clock time, both Task 1 and 3 are scheduled to execute. Since Task 1 has higher priority, it runs first, and Task 3 is delayed. While executing Task 1, the FMU informs the importer that v_m^c's task (Task 2) should be scheduled to run at wall-clock time t_2. At wall-clock time t_2, Task 1 is still executing, so Task 2 is delayed until wall clock time t_3. At wall-clock time t_3, Task 2 starts executing, but note that the activation time of Task 2 is still its scheduled time t_2. This is where the wall-clock time t_3 differs from the simulated time t_2. At t_4, Task 2 is preempted, because of Task 1. Finally, after being delayed substantially, Task 3 gets to execute, with its simulated time t_0.

We refer the reader to (Cláudio Gomes et al. 2021) and the FMI standard, for more details about the Scheduled Execution API.

5 Quality Improvement Measures and Prototypical Implementations

The development of new features followed the FMI development process (Modelica Association 2015) with the creation of FMI Change Proposals (FCP) providing the use cases, suggested changes and partial prototypes.

During the development of the FMI 3.0 standard, the text was completely restructured and several concepts were unified between the different interface types in a common concepts section. The state machines were unified between the different interface types.

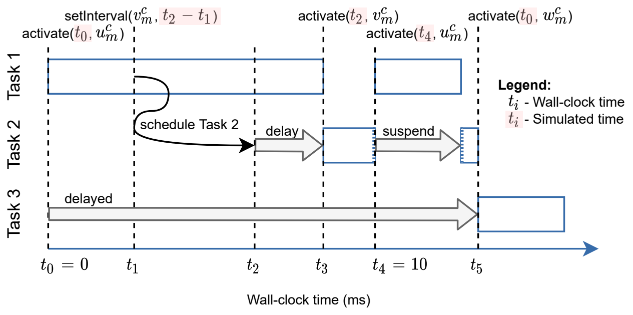

Figure 8. Example execution trace of Figure 7. Task *N* corresponds to execution of partition *N*, as detailed in Figure 7.

In order to streamline the standard text, implementation specific hints will be singled out in an *FMI implementers guide*.

Reference FMUs (*Reference FMUs* 2021) were created to showcase and test certain features of FMI. These FMUs and code snippets extracted from them, are continuously compiled and example XML files automatically validated against the schema files in a continuous integration environment to ensure correctness of the examples included into the standard document.

Several tools were used to validate prototypes of FMI 3.0 features: Altair Activate, Dymola, fmpy (*fmpy* 2021), Model.CONNECT™, Silver, SimulationX, among others.

6 Summary and Outlook

The success of the Modelica Association's FMI standard 1.0 and 2.0 has created the desire to improve simulation efficiency and accuracy, as well as to enable new use cases. FMI 3.0 introduces a number of new features and improvements to address many of the needs found in current industrial and research applications. More importantly, opening up to layered standards will help to address many of the future needs not yet envisioned and to address industry specific requirements best addressed by special-purpose extensions without weighting down the core FMI standard document.

The developers of the current version of FMI are well aware of the current challenges of the simulation community. In particular, simulation efficiency will be the focus of future FMI development.

Another important drive for future FMI versions is harmonization with other complementary Modelica Association standards, such as SSP, DCP and eFMI. Finding structural ways to make these standards more compatible and therefore easier to implement, will increase each standard's added value.

Acknowledgements

The authors wish to thank all the contributors to the FMI specification. Contributions of ESI ITI GmbH were carried out within the project FMI4BIM, funded by the German Federal Ministry for Economic Affairs and Energy.

Supported by:

Federal Ministry for Economic Affairs and Energy

on the basis of a decision by the German Bundestag

References

Arnold, Martin, Christoph Clauß, and Tom Schierz (2014). "Error Analysis and Error Estimates for Co-Simulation in FMI for Model Exchange and Co-Simulation v2.0". In: *Progress in Differential-Algebraic Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 107–125. ISBN: 978-3-662-44926-4. DOI: 10.1007/978-3-662-44926-4_6.

ASAM (2021). *ASAM MCD-1 XCP Standard*. URL: https://www.asam.net/standards/detail/mcd-1-xcp/wiki/ (visited on 2021-04-18).

Baydin, Atilim Gunes et al. (2015). *Automatic differentiation in machine learning*. URL: https://arxiv.org/abs/1502.05767 (visited on 2021-05-06).

Benedikt, M et al. (2013-06). "NEPCE-A Nearly Energy Preserving Coupling Element for Weak-Coupled Problems and Co-Simulation". In: *IV International Conference on Computational Methods for Coupled Problems in Science and Engineering, Coupled Problems*. Ibiza, Spain, pp. 1–12.

Blochwitz, Torsten et al. (2011). "The Functional Mockup Interface for Tool independent Exchange of Simulation". In: *8th International Modelica Conference*. URL: http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf.

Blochwitz, Torsten et al. (2012). "Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models". In: *8th International Modelica Conference*. URL: https://lup.lub.lu.se/search/ws/files/5428900/2972293.pdf.

Busch, Martin (2016-09). "Continuous Approximation Techniques for Co-Simulation Methods: Analysis of Numerical Stability and Local Error". In: *Journal of Applied Mathematics and Mechanics* 96.9, pp. 1061–1081. ISSN: 00442267. DOI: 10.1002/zamm.201500196.

fmpy (2021). URL: https://github.com/CATIA-Systems/FMPy (visited on 2021-04-20).

Franke, Rüdiger et al. (2009). "Stream Connectors – An Extension of Modelica for Device-Oriented Modeling of Convective Transport Phenomena". In: *Proceedings of the 7th International Modelica Conference* (Como, I, September 20–22, 2009). Ed. by Francesco Casella. Linköping Electronic Conference Proceedings. Linköping: Linköping University Electronic Press, pp. 108–121. DOI: 10.3384/ecp09430078. URL: http://dx.doi.org/10.3384/ecp09430078.

Franke, Rüdiger et al. (2017). "Discrete-time models for control applications with FMI". In: *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*. 132. Linköping University Electronic Press, pp. 507–515. URL: https://2017.international.conference.modelica.org/proceedings/html/submissions/ecp17132507_FrankeMattssonOtterWernerssonOlssonOchelBlochwitz.pdf.

Fritzson, Dag, Jonas Ståhl, and Iakov Nakhimovski (2007). "Transmission Line Co-Simulation of Rolling Bearing Applications". In: *48th Conference on Simulation and Modelling*. Göteborg, Sweden: Citeseer, pp. 24–39.

Gomes, Cláudio et al. (2018). "Co-Simulation: A Survey". In: *ACM Computing Surveys* 51.3, 49:1–49:33. DOI: 10.1145/3179993.

Gomes, Cláudio et al. (2021). "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations". In: *Proceedings of the 14th International Modelica Conference*. 14th International Modelica Conference. online: Linköping University Electronic Press, Linköpings Universitet, to be published.

Modelica Association (2015-07). *FMI Development Process And Communication Policy*. URL: https://github.com/modelica/fmi-standard.org/blob/master/assets/FMI_DevelopmentProcess_1.0.pdf.

Modelica Association (2021a-04). *Functional Mock-up Interface Specification, v3.0beta.1*. URL: https://github.com/modelica/fmi-standard/releases/tag/v3.0-beta.1.

Modelica Association (2021b). *FMI Website*. URL: https://fmi-standard.org/ (visited on 2021-04-18).

Modelica Association (2021c). *FMI Website*. URL: https://fmi-standard.org/tools/ (visited on 2021-04-18).

Ochel, Lennart et al. (2019-02). "OMSimulator - Integrated FMI and TLM-Based Co-Simulation with Composite Model Editing and SSP". In: *The 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, pp. 69–78. DOI: 10.3384/ecp1915769.

Reference FMUs (2021). URL: https://github.com/modelica/Reference-FMUs (visited on 2021-05-02).

Sadjina, Severin et al. (2017-07). "Energy Conservation and Power Bonds in Co-Simulations: Non-Iterative Adaptive Step Size Control and Error Estimation". In: *Engineering with Computers* 33.3, pp. 607–620. ISSN: 1435-5663. DOI: 10.1007/s00366-016-0492-8.