ITEA 2

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

Functional Mock-up Interface for Model Exchange and Co-Simulation

Document version: 2.0 Beta 4
August 10, 2012

Road map for Version 2.0			
Nov. 14, 2011	Beta 3	**First public draft.** The content is stable and no changes are planned to it. Further changes should be only "additional features".	Prototype implementations to verify the specification (Dymola, SimulationX, Silver, ...)
Aug. 10, 2012	Beta 4	**Second public draft**. Several improved features where issues detected/reported are fixed	
Oct. 2012	Beta 5	**Third public draft.** Issues detected/reported are fixed and precise, time event handling is added	
Dec. 2012	Release	**Release of FMI 2.0**	

Improvement suggestions for the public draft versions should be reported to the Public-FMI issue tracking system: https://resources.qtronic.de/trac/fmi_spec_public

The FMI design team is currently working on "Precise handling of time events for periodic and aperiodic sampled data systems". This enhancement is the final missing feature for FMI 2.0. It will be included in the Beta 5 version. This means that some additional elements and attributes are added to the xml file and the function calls for events (fmiInitializeXXX, fmiEventUpdate) might be slightly changed. Otherwise, no changes are expected.

The following features originally planned for 2.0 will be introduced in version 2.1 in a backwards compatible way:

- Direct support for arrays.
- Defining the interface of the FMU to an object diagram editor where FMU instances are used as graphical objects (combining input/output signals to ports, like physical ports or signal busses, defining the icon and the positioning of the ports at the icon).

modelisar

History

Version	Date	Remarks
1.0	2010-01-26	First version of FMI for Model Exchange
1.0	2010-10-12	First version of FMI for Co-Simulation
2.0 Beta 3	2011-11-14	First Public Beta Version of FMI for Model Exchange and Co-Simulation for version 2.0
2.0 Beta 4	2012-08-10	Second Public Beta Version of FMI for Model Exchange and Co-Simulation for version 2.0

License of this document

Abstract

This document defines the Functional Mock-up Interface (FMI), version 2.0 to (a) exchange dynamic models between tools and (b) define tool coupling for dynamic system simulation environments.

FMI for Model Exchange (chapter 3)

The intention is that a modeling environment can generate C code of a dynamic system model that can be utilized by other modeling and simulation environments. Models are described by differential, algebraic and discrete equations with time-, state- and step-events. If the C code describes a continuous system, then this system is solved with the integrators of the environment where it is used. The models to be treated by this interface can be large for usage in offline or online simulation, or can be used in embedded control systems on micro-processors.

FMI for Co-Simulation (chapter 4)

The intention is to provide an interface standard for coupling two or more simulation tools in a co-simulation environment. The data exchange between subsystems is restricted to discrete communication points. In the time between two communication points, the subsystems are solved independently from each other by their individual solver. Master algorithms control the data exchange between subsystems and the synchronization of all simulation solvers (slaves). Both, rather simple master algorithms, as well as more sophisticated ones are supported. Note, that the master algorithm itself is *not* part of the FMI standard.

FMI Common Concepts (chapter 2)

The two interface standards have many parts in common. In particular, it is possible to utilize several instances of a model and/or a co-simulation tool and to connect them hierarchically together. The interfaces are independent of the target environment because no header files are used that depend on the target environment (with exception of the data types of the target platform). This allows generating one dynamic link library that can be utilized in any environment on the same platform. A model, a co-simulation slave or the coupling part of a tool, is distributed in one zip file called FMU (Functional Mock-up Unit) that contains several files:

(1) An XML file contains the definition of all exposed variables in the FMU and other static information. It is then possible to run the FMU on a target system without this information, in other words with no unnecessary overhead.

(2) All needed model equations or the access to co-simulation tools are provided with a small set of easy to use C functions. A new caching technique allows a more efficient evaluation of the model equations than in other approaches. These C functions can either be provided in source and/or binary form. Binary forms for different platforms can be included in the same model zip file.

(3) The model equations or the co-simuation tool can be either provided directly in the FMU, or the FMU contains only a generic communication module that communicates with an external tool that evaluates or simulates the model. In the XML file information about the capabilities of the FMU are present, for example to characterize the ability of a co-simulation slave to support advanced master algorithms such as the usage of variable communication step sizes, higher order signal extrapolation, or others.

(4) Further data can be included in the FMUzip file, especially a model icon (bitmap file), documentation files, maps and tables needed by the FMU, and/or all object libraries or dynamic link libraries that are utilized.

A growing set of tools supports FMI. The actual list of tools is available at:

http://www.functional-mockup-interface.org/tools.html

About FMI 2.0

This version 2.0 is a major enhancement compared to FMI 1.0, where the FMI 1.0 Model Exchange and Co-Simulation standards have been merged, and many improvements have been incorporated, often due to practical experience when using the FMI 1.0 standards. New features are usually optional (need neither be supported by the tool that exports an FMU, nor by the tool that imports an FMU). Details are provided in appendix A.3.1. The appendix of the FMI 1.0 specification has been mostly moved in an extended and improved form to a companion document

"FunctionalMockupInterface-ImplementationHints.pdf"

where practical information for the implementation of the FMI standard is provided.

Conventions used in this Document

- Non-normative text is given in square brackets in italic font: [*especially examples are defined in this style.*].
- Arrays appear in two forms:

  (a) In the end-user/logical view, one- and two-dimensional arrays are used. Here the convention of linear algebra, the control community and the most important tools in this area is utilized, in other words the first element along one dimension starts at index one. In all cases of (a), the starting index is also explicitly mentioned at the respective definition of the array. Example: In the modelDescription.XML file, the inputs, outputs, states and derivatives are defined as ordered sets where the first element is referenced with index one (these indices are, for example used to define the sparseness structure of partial derivative matrices).

  (b) In the implementation view, one-dimensional C-arrays are used. In order to access an array element the C-convention is used, in other words the first element of array `x` is access with `x[0]` and the first element of input argument `x` for function `setContinuousStates(..)` is `x[0]`.

Contents

1. Overview

The FMI (Functional Mock-up Interface) defines an interface to be implemented by an executable called FMU (Functional Mock-up Unit). The FMI functions are used (called) by a simulation environment to create one or more instances of the FMU and to simulate them, typically together with other models. An FMU may either be self-integrating (FMI for Co-Simulation, chapter 4) or require the simulation environment to perform numerical integration (FMI for Model Exchange, chapter 3). The goal of this interface is that the calling of an FMU in a simulation environment is reasonably simple. No provisions are provided in this document how to generate an FMU from a modeling environment, with the exception of hints for implementation in the companion document "FunctionalMockupInterface-ImplementationHints.pdf ".

The FMI for Model Exchange interface defines an interface to the model of a dynamic system described by differential, algebraic and discrete-time equations and to provide an interface to evaluate these equations as needed in different simulation environments, as well as in embedded control systems, with explicit or implicit integrators and fixed or variable step-size. The interface is designed to allow the description of large models.

The FMI for Co-Simulation interface is designed both for the coupling of simulation tools (simulator coupling, tool coupling), and coupling with subsystem models, which have been exported by their simulators together with its solvers as runnable code. The goal is to compute the solution of time dependent coupled systems consisting of subsystems that are continuous in time (model components that are described by differential-algebraic equations) or time-discrete (model components that are described by difference equations, for example discrete controllers). In a block representation of the coupled system, the subsystems are represented by blocks with (internal) state variables $\mathbf{x}(t)$ that are connected to other subsystems (blocks) of the coupled problem by subsystem inputs $\mathbf{u}(t)$ and subsystem outputs $\mathbf{y}(t)$.

In case of tool coupling, the modular structure of coupled problems is exploited in all stages of the simulation process beginning with the separate model setup and pre-processing for the individual subsystems in different simulation tools. During time integration, the simulation is again performed independently for all subsystems restricting the data exchange between subsystems to discrete *communication points*. Finally, also the visualization and post-processing of simulation data is done individually for each subsystem in its own native simulation tool.

The two interfaces have large parts in common. These parts are defined in chapter 2. In particular:

- FMI Application Programming Interface (C)
  All needed equations or tool coupling computations are evaluated by calling standardized "C" functions. "C" is used, because it is the most portable programming language today and is the only programming language that can be utilized in all embedded control systems.

- FMI Description Schema (XML)
  The schema defines the structure and content of an XML file generated by a modeling environment. This XML file contains the definition of all variables of the FMU in a standardized way. It is then possible to run the C code in an embedded system without the overhead of the variable definition (the alternative would be to store this information in the C code and access it via function calls, but this is neither practical for embedded systems nor for large models). Furthermore, the variable definition is a complex data structure and tools should be free how to represent this data structure in their programs. The selected approach allows a tool to store and access the variable definitions (without any memory or efficiency overhead of standardized access functions) in the programming language of the simulation environment, usually C++, C#, Java, or Python. Note, there are many free and commercial libraries in different programming languages to read XML files into an appropriate

data structure, see for example http://en.wikipedia.org/wiki/XML#Parsers, and especially the efficient open source parser SAX (http://sax.sourceforge.net/, http://en.wikipedia.org/wiki/Simple_API_for_XML).

An FMU (in other words a model without integrators, a runnable model with integrators, or a tool coupling interface) is distributed in one zip file. The zip file contains (more details are given in section 2.3):

- The FMI Description File (in XML format).
- The C sources of the FMU, including the needed run-time libraries used in the model, and/or binaries for one or several target machines, such as Windows dynamic link libraries (.dll) or Linux shared object libraries (.so). The latter solution is especially used if the FMU provider wants to hide the source code to secure the contained know-how or to allow a fully automatic import of the FMU in another simulation environment. An FMU may contain physical parameters or geometrical dimensions, which should not be open. On the other hand, some functionality requires source code.
- Additional FMU data (like tables, maps) in FMU specific file formats.

A schematic view of an FMU is shown in the next figure:

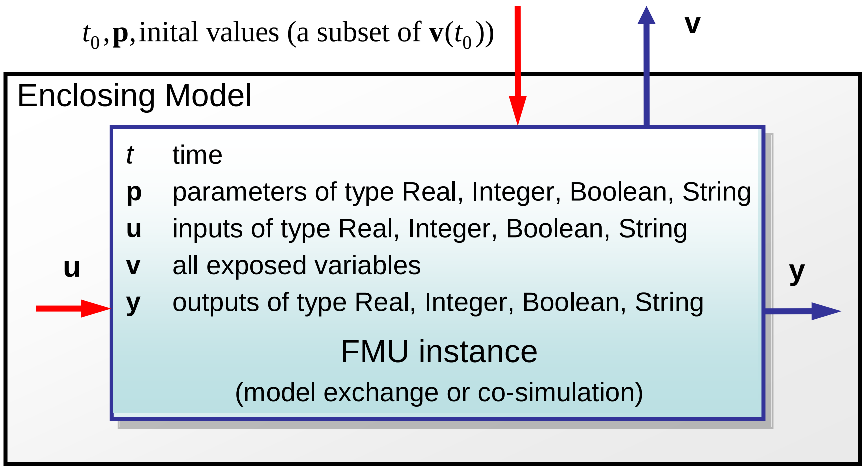

$t_0, \mathbf{p},$ inital values (a subset of $\mathbf{v}(t_0)$)
v
Enclosing Model
t time
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
y outputs of type Real, Integer, Boolean, String
u
y
FMU instance
(model exchange or co-simulation)

Figure 1: Data flow between the environment and an FMU. For details, see chapters 3 and 4.
Blue arrows: Information provided by the FMU.
Red arrows: Information provided to the FMU.

The publications currently available for FMI are referenced in section 5 “Literature”. Especially, *Andersson et.al. 2011, Bastian et.al. 2011, Blochwitz et.al. 2011, Brembeck et.al. 2011, Noll et.al. 2011, Schubert et.al. 2011, Thiele and Henriksson 2011*.

1.1 Properties and Guiding Ideas

In this section, properties are listed and some principles are defined that guided the low-level design of the FMI. This shall increase self consistency of the interface functions. The listed issues are sorted, starting from high-level properties to low-level implementation issues.

Expressivity: The FMI provides the necessary features that Modelica®, Simulink® and SIMPACK® models[1] can be transformed to an FMU.

[1] Modelica is a registered trademark of the Modelica Association, Simulink is a registered trademark of the MathWorks Inc., SIMPACK is a registered trademark of SIMPACK AG.

Stability: FMI is expected to be supported by many simulation tools world wide. Implementing such support is a major investment for tool vendors. Stability and backwards compatibility of the FMI has therefore high priority. To support this, the FMI defines 'capability flags' that will be used by future versions of the FMI to extend and improve the FMI in a backwards compatible way, whenever feasible.*Implementation*: FMUs can be written manually or can be generated automatically from a modeling environment. Existing manually coded models can be transformed manually to a model according to the FMI standard.

Processor independence: It is possible to distribute an FMU without knowing the target processor. This allows to run an FMU on a PC, a Hardware-in-the-Loop simulation platform or as part of the controller software of an ECU, e. g. as part of an AUTOSAR SWC. Keeping the FMU independent of the target processor increases the usability of the FMU and is even required by the AUTOSAR software component model. Implementation: using a textual FMU (distribute the C source of the FMU).

Simulator independence: It is possible to compile, link and distribute an FMU without knowing the target simulator. Reason: The standard would be much less attractive otherwise, unnecessarily restricting the later use of an FMU at compile time and forcing users to maintain simulator specific variants of an FMU. Implementation: using a binary FMU. When generating a binary FMU, e. g. a Windows dynamic link library (.dll) or a Linux shared object library (.so), the target operating system and eventually the target processor must be known. However, no run-time libraries, source files or header files of the target simulator are needed to generate the binary FMU. As a result, the binary FMU can be executed by any simulator running on the target platform (provided the necessary licenses are available, if required from the model or from the used run-time libraries).

Small run-time overhead: Communication between an FMU and a target simulator through the FMI does not introduce significant run time overhead. This is achieved by a new caching technique (to avoid computing the same variables several times) and by exchanging vectors instead of scalar quantities.

Small footprint: A compiled FMU (the executable) is small. Reason: An FMU may run on an ECU (Electronic Control Unit, for example a micro processor), and ECUs have strong memory limitations. This is achieved by storing signal attributes (names, units, etc.) and all other static information not needed for model evaluation in a separate text file (= Model Description File) that is not needed on the micro processor where the executable might run.

Hide data structure: The FMI for Model Exchange does not prescribe a data structure (a C struct) to represent a model. Reason: the FMI standard shall not unnecessarily restrict or prescribe a certain implementation of FMUs or simulators (whoever holds the model data), to ease implementation by different tool vendors.

Support many and nested FMUs: A simulator may run many FMUs in a single simulation run and/or multiple instances of one FMU. The inputs and outputs of these FMUs can be connected with direct feed through. Moreover, an FMU may contain nested FMUs.

Numerical Robustness: The FMI standard allows that problems which are numerically critical (for example time and state events, multiple sample rates, stiff problems) can be treated in a robust way.

Hide cache: A typical FMU will cache computed results for later reuse. To simplify usage and to reduce error possibilities by a simulator, the caching mechanism is hidden from the FMI. Reason: First, the FMI should not force an FMU to implement a certain caching policy. Second, this helps to keep the FMI simple. Implementation: The FMI provides explicit methods (called by the simulator) for setting properties that invalidate cached data. An FMU that chooses to implement a cache may maintain a set of 'dirty' flags, hidden from the simulator. A get method, e. g. to a state, will then either trigger a computation, or return cached data, depending on the value of these flags.

Support numerical solvers: A typical target simulator will use numerical solvers. These solvers require vectors for states, derivatives and zero-crossing functions. The FMU directly fills the values of such vectors provided by the solvers. Reason: minimize execution time. The exposure of these vectors conflicts somewhat with the 'hide data structure' requirement, but the efficiency gain justifies this.

Explicit signature: The intended operations, argument types and return values are made explicit in the signature. For example an operator (such as 'compute_derivatives') is not passed as an int argument but a special function is called for this. The 'const' prefix is used for any pointer that should not be changed, including 'const char*' instead of 'char*'. Reason: the correct use of the FMI can be checked at compile time and allows calling of the C code in a C++ environment (which is much stricter on ‘const’ as C is). This will help to develop FMUs that use the FMI in the intended way.

Few functions: The FMI consists of a few, 'orthogonal' functions, avoiding redundant functions that could be defined in terms of others. Reason: This leads to a compact, easy to use, and hence attractive API with a compact documentation.

Error handling: All FMI methods use a common set of methods to communicate errors.

Allocator must free: All memory (and other resources) allocated by the FMU are freed (released) by the FMU. Likewise, resources allocated by the simulator are released by the simulator. Reason: this helps to prevent memory leaks and runtime errors due to incompatible runtime environments for different components.

Immutable strings: All strings passed as arguments or returned are read-only and must not be modified by the receiver. Reason: This eases the reuse of strings.

Use C: The FMI is encoded using C, not C++. Reason: Avoid problems with compiler and linker dependent behavior. Run FMU on embedded target.

This version of the functional mock-up interface does not have the following desirable properties. They might be added in a future version:

- The interface is for ordinary differential equations in state space form (ODE). It is not for a general differential-algebraic equation system. However, algebraic equation systems inside the FMU are supported (for example the FMU can report to the environment to re-run the current step with a smaller step size since a solution could not be found for an algebraic equation system).
- Special features as might be useful for multi-body system programs, like SIMPACK, are not included.
- The interface is for simulation and for embedded systems. Properties that might be additionally needed for optimization are not included.
- No explicit definition of the variable hierarchy in the XML file.
- The number of states and number of event indicators are fixed for an FMU and cannot be changed.

1.2 Acknowledgements

This work was carried out within the ITEA2 MODELISAR project (project number: ITEA 2 – 07006, www.itea2.org/public/project_leaflets/MODELISAR_profile_oct-08.pdf).

- Daimler AG, DLR, ITI GmbH, Martin Luther University Halle-Wittenberg, QTronic GmbH and SIMPACK AG thank BMBF for partial funding of this work within MODELISAR (BMBF Förderkennzeichen: 01lS0800x).
- Dynasim AB thanks the Swedish funding agency VINNOVA (2008-02291) for partial funding of this work within MODELISAR.
- LMS Imagine thanks DGCIS for partial funding of this work within MODELISAR.

2. FMI Common Concepts for Model Exchange and Co-Simulation

In this chapter, the concepts are defined that are common for "model exchange" and for "co-simulation". In both cases, FMI defines an input/output block of a dynamic model where the distribution of the block, the platform dependent header file, several access functions, as well as the schema files are identical. The definitions that are specific to the particular case are defined in chapters 3 and 4.

Below, the term **FMU** (Functional Mock-up Unit) will be used as common term for a model in the "FMI for model exchange" format, or a co-simulation slave in the "FMI for co-simulation" format. Note, the interface supports several instances of one FMU.

2.1 FMI Application Programming Interface

This section contains the common interface definitions to execute functions of an FMU from a C program.

2.1.1 Header Files and Naming of Functions

Three header files are provided that define the interface of an FMU. In all header files the convention is used that all C functions and type definitions start with the prefix "fmi":

- "`fmiTypesPlatform.h` "
  contains the type definitions of the input and output arguments of the functions. This header file must be used both by the FMU and by the target simulator. If the target simulator has different definitions in the header file (for example "`typedef float fmiReal`" instead of "`typedef double fmiReal`"), then the FMU needs to be re-compiled with the header file used by the target simulator. Note, the header file platform for which the model was compiled can be inquired in the target simulator with `fmiGetTypesPlatform`, see section 2.1.4.
  [*Example for a definition in this header file:*
  ```
  typedef double fmiReal;
  ```
  ]

- "`fmiFunctionTypes.h`"
  contains `typedef` definitions of all function prototypes of an FMU. When dynamically loading an FMU, these definitions can be used to type-cast the function pointers to the respective function definition.
  [*Example for a definition in this header file:*
  ```
  typedef fmiStatus fmiSetTimeTYPE(fmiComponent, fmiReal);
  ```
  ]

- "`fmiFunctions.h`"
  contains the function prototypes of an FMU that can be accessed in simulation environments and that are defined in chapters 2, 3, and 4. This header file includes "`fmiTypesPlatform.h` " and "`fmiFunctionTypes.h`". Note, the header file version number for which the model was compiled, can be inquired in the target simulator with `fmiGetVersion`, see section 2.1.4.
  [*Example for a definition in this header file*[2]*:*
  ```
  DllExport fmiSetTimeTYPE fmiSetTime;
  ```
  ]

The goal is that both textual and binary representations of FMUs are supported and that several FMUs might be present at the same time in an executable (for example FMU A may use an FMU B). In order

[2] On Windows, "`DllExport`" is defined as "`__declspec(dllexport)`" in order to export the name for dynamic loading. On Non-Windows systems it is an empty definition.

for this to be possible, the names of the functions in different FMUs must be different or function pointers must be used. To support the first variant macros are provided in "`fmiFunctions.h`" to build the actual function names by using a function prefix that depends on how the FMU is shipped. Typically, FMU functions are used as follows:

```
// FMU is shipped with C source code, or with static link library
#define  FUNCTION_PREFIX MyModel_
#include "fmiFunctions.h"
< usage of the FMU functions >


// FMU is shipped with DLL/SharedObject
#define  FUNCTION_PREFIX
#include "fmiFunctions.h"
< usage of the FMU functions >
```

A function that is defined as "`fmiGetReal`" is changed by the macros to the actual function name:

- FMU is shipped with C source code, or with static link library:
  The constructed function name is "`MyModel_fmiGetReal`", in other words the function name is prefixed with the model name and an "_". As `FUNCTION_PREFIX` the "modelIdentifier" attribute defined in "fmiModelDescription.ModelExchange", or "fmiModelDescription.CoSimulation" is used, together with "_" at the end (see sections 3.3.1 and 4.3.1). A simulation environment can therefore construct the relevant function names by generating code for the actual function call. In case of a static link library, the name of the library is MyModel.lib on Windows, and libMyModel.a on Linux, in other words the "modelIdentifier" attribute is used as library name.
- FMU is shipped with DLL/SharedObject:
  The constructed function name is "`fmiGetReal`", in other words it is not changed. A simulation environment will then dynamically load this library and will explicitly import the function symbols by providing the FMI function names as strings. The name of the library is MyModel.dll on Windows or MyModel.so on Linux, in other words the "modelIdentifier" attribute is used as library name.

[*An FMU can be optionally shipped so that it basically contains only the communication to another tool (needsExecutionTool = true, see section 4.3.1). This is particularily common for co-simulation tasks. In FMI 1.0, the function names are always prefixed with the model name and therefore a DLL/Shared Object has to be generated for every model. FMI 2.0 improves this situation since model names are no longer used as prefix in case of DLL/Shared Objects: Therefore one DLL/Shared Object can be used for all models in case of tool coupling. If an FMU is imported into a simulation environment, this is usually performed dynamically (based on the FMU name, the corresponding FMU is loaded during execution of the simulation environment) and then it does not matter whether a model name is prefixed or not.*]

Since "`modelIdentifier`" is used as prefix of a C-function name it must fulfill the restrictions on C-function names (only letters, digits and/or underscores are allowed). [*For example if modelName = "A.B.C", then modelIdentifier might be "A_B_C"*]. Since "`modelIdentifier`" is also used as name in a file system, it must also fulfill the restrictions of the targeted operating systems. Basically, this means that it should be short. For example the Windows API only supports full path-names of a file up to 260 characters (see: http://msdn.microsoft.com/en-us/library/aa365247%28VS.85%29.aspx).

2.1.2 Platform Dependent Definitions (fmiTypesPlatform.h)

To simplify porting, no C types are used in the function interfaces, but the alias types defined in this section. All definitions in this section are provided in the header file "`fmiTypesPlatform.h `".

```
#define fmiTypesPlatform "standard32"
```

A definition that can be inquired with `fmiGetTypesPlatform`. It defines the platform for which this header file is provided. A platform is a combination of machine, compiler, and operating system. The default definition “standard32” defines a standard 32-bit platform:

```
fmiComponent           : 32 bit pointer
fmiComponentEnvironment: 32 bit pointer
fmiFMUstate            : 32 bit pointer
fmiValueReference      : 32 bit
fmiReal                : 64 bit
fmiInteger             : 32 bit
fmiBoolean             : 32 bit
fmiString              : 32 bit pointer
fmiByte                :  8 bit
```

typedef void* `fmiComponent;`

This is a pointer to an FMU specific data structure that contains the information needed to process the model equations or to process the co-simulation of the respective slave. This data structure is implemented by the environment that provides the FMU, in other words the calling environment does not know its content and the code to process it must be provided by the FMU generation environment and must be shipped with the FMU.

typedef void* `fmiComponentEnvironment;`

This is a pointer to a data structure in the simulation environment that calls the FMU. Via this pointer, data from the modelDescription.XML file [*(for example mapping of valueReferences to variable names)*] can be transferred between the simulation environment and the logger function (see section 2.1.5).

typedef void* `fmiFMUstate;`

This is a pointer to a data structure in the FMU that saves the internal FMU state of the actual or a previous time instant. This allows to restart a simulation from a previous FMU state (see section 2.1.7)

typedef unsigned int `fmiValueReference;`

This is a handle to a (base type) variable value of the model. Handle and base type (such as `fmiReal`) uniquely identify the **value** of a variable. Variables of the same base type that have the same handle, always have identical values, but other parts of the variable definition might be different [*(for example min/max attributes)*].

All structured entities, like records or arrays, are “flattened” into a set of scalar values of type `fmiReal`, `fmiInteger` etc. An `fmiValueReference` references one such scalar. The coding of `fmiValueReference` is a “secret” of the environment that generated the FMU. The interface to the equations only provides access to variables via this handle. Extracting concrete information about a variable is specific to the used environment that reads the Model Description File in which the value handles are defined.

If a function in the following sections is called with a wrong “`fmiValueReference`” value [*(for example setting a constant with a `fmiSetReal(..)` function call)*], then the function has to return with an error (`fmiStatus = fmiError`, see section 2.1.3).

#define `fmiUndefinedValueReference ((fmiValueReference) (-1))`

If `fmiValueReference` is undefined, it has the value `fmiUndefinedValueReference` which is the largest value of `unsigned int`. This value might be used, for example as return argument of `fmiGetStateValueReferences`, (see section 3.2.3) in order to hide the meaning of a state.

```
typedef double      fmiReal   ;  // Real number (64 bits)
typedef int         fmiInteger;  // Integer number (32 bits)
typedef int         fmiBoolean;  // Boolean number
                                 // (32 bit, two values: fmiFalse, fmiTrue)
typedef const char* fmiString ;  // Character string
                                 // ('\0' terminated, UTF8 encoded)
typedef char        fmiByte   ;  // Byte (8 bits)

#define fmiTrue  1
#define fmiFalse 0
```

These are the basic data types used in the interfaces of the C functions. More data types might be included in future versions of the interface. In order to keep flexibility, especially for embedded systems or for high performance computers, the exact data types or the word length of a number are not standardized. Instead, the precise definition (in other words the header file "`fmiTypesPlatform.h` ") is provided by the environment where the FMU shall be used. In most cases, the definition above will be used. If the target environment has different definitions and the FMU is distributed in binary format, it must be newly compiled and linked with this target header file.

If an `fmiString` variable is passed as input argument to a function and the string will be used after the function has returned, the string must be copied (not only the pointer) and stored in the internal FMU memory, because there is no guarantee for the lifetime of the string after the function has returned.

If an `fmiString` variable is passed as output argument from a function and the string shall be used in the target environment, the whole string must be copied (not only the pointer). The memory of this string may be deallocated by the next call to any of the FMI interface functions (the string memory might also be just a buffer, that is reused)

2.1.3 Status Returned by Functions

This section defines the "`status`" flag (an enumeration of type `fmiStatus` defined in file "`fmiFunctionTypes.h`") that is returned by all functions to indicate the success of the function call:

```
typedef enum { fmiOK,
               fmiWarning,
               fmiDiscard,
               fmiError,
               fmiFatal,
               fmiPending } fmiStatus;
```

Status returned by functions. The status has the following meaning

- `fmiOK` – all well
- `fmiWarning` – things are not quite right, but the computation can continue. Function "`logger`" was called in the model (see below) and it is expected that this function has shown the prepared information message to the user.
- `fmiDiscard` – this return status is only possible, if explicitly defined for the corresponding function (currently[3]: `fmiSetReal`, fmiSetInteger, `fmiSetContinuousStates`, `fmiGetReal`, `fmiGetDerivatives`, `fmiGetEventIndicators`, `fmiDoStep`):
  For "model exchange": It is recommended to perform a smaller step size and evaluate the model equations again, for example because an iterative solver in the model did not converge or because a

[3] fmiSetReal, fmiSetInteger and fmiSetContinuousStates could check whether the input arguments are in their validity range. If not, these functions could return with fmiDiscard.

function is outside of its domain (for example sqrt(<negative number>)). If this is not possible, the simulation has to be terminated.
For "co-simulation": `fmiDiscard` is returned also if the slave is not able to return the required status information. The master has to decide if the simulation run can be continued.
In both cases, function "`logger`" was called in the FMU (see below) and it is expected that this function has shown the prepared information message to the user if the FMU was called in debug mode (`loggingOn` = `fmiTrue`). Otherwise, "`logger`" should not show a message.

- `fmiError` – the FMU encountered an error. The simulation cannot be continued with this FMU instance.
  For "model exchange": `fmiFreeModelInstance` must be called afterwards.
  For "co-simulation": If one of the functions (except `fmiDoStep`) returns `fmiError`, the simulation cannot be continued and `fmiFreeInstance` must be called afterwards.
  Further processing is possible after this call; especially other FMU instances are not affected. Function "`logger`" was called in the FMU (see below) and it is expected that this function has shown the prepared information message to the user.

- `fmiFatal` – the model computations are irreparably corrupted for all FMU instances. Function "`logger`" was called in the FMU (see below) and it is expected that this function has shown the prepared information message to the user. It is not possible to call any other function for any of the FMU instances.

- `fmiPending` – is returned only from the co-simulation interface, if the slave executes the function in an asynchronous way. That means the slave starts to compute but returns immediately. The master has to call `fmiGetStatus(..., fmiDoStepStatus)` to determine, if the slave has finished the computation. Can be returned only by `fmiDoStep` and by `fmiGetStatus` (see section 4.2.4).

2.1.4 Inquire Platform and Version Number of Header Files

This section documents functions to inquire information about the header files used to compile its functions.

```
const char* fmiGetTypesPlatform();
```

Returns the name of the set of (compatible) platforms of the "`fmiTypesPlatform.h` " header file which was used to compile the functions of the FMU. The function returns a pointer to the static variable "`fmiPlatform`" defined in this header file. The standard header file, as documented in this specification, has version "`standard32`" (so this function usually returns "`standard32`").

```
const char* fmiGetVersion();
```

Returns the version of the "`fmiFunctions.h`" header file which was used to compile the functions of the FMU. The function returns "`fmiVersion`" which is defined in this header file. The standard header file as documented in this specification has version "`2.0`" (so this function usually returns "`2.0`").

2.1.5 Creation and Logging of FMU Instances

This section documents functions that deal with instantiation and logging of FMUs.

```
fmiComponent fmiInstantiateXXX(fmiString  instanceName, fmiString fmuGUID,
                               fmiString  fmuResourceLocation,
                               const fmiCallbackFunctions* functions,
                               fmiBoolean                  visible,
                               fmiBoolean                  loggingOn);
```

FMUs for model exchange and for co-simulation are instantiated with different function calls (`fmiInstantiateModel` – section 3.2.1, `fmiInstantiateSlave` – section 4.2.1). Both functions have identical arguments that are defined below:

The functions return a new instance of an FMU. If a null pointer is returned, then instantiation failed. In that case, "`functions->logger`" was called with detailed information about the reason. An FMU can be instantiated many times. This function must be called successfully, before any of the following functions can be called. For co-simulation, this function call has to perform all actions of a slave which are necessary before a simulation run starts (for example loading the model file, compilation...).

Argument `instanceName` is a unique identifier for the FMU instance. It is used to name the instance, for example in error or information messages generated by one of the `fmiXXX` functions. It is not allowed to provide a null pointer and this string must be non-empty (in other words must have at least one character that is no white space). [*If only one FMU is simulated, as instanceName attribute* `modelName` *or* `modelExchange/CoSimulation.modelIdentifier` *from the XML schema* `fmiModelDescription` *might be used.*]

Argument `fmuGUID` is used to check that the modelDescription.XML file (see section 2.3) is compatible with the C code of the FMU. It is a vendor specific globally unique identifier of the XML file (for example it is a "fingerprint" of the relevant information stored in the XML file). It is stored in the XML file as attribute "guid" (see section 2.2.1) and has to be passed to the fmiInstantiateXXX function via argument `fmuGUID`. It must be identical to the one stored inside the fmiInstantiateXXX function. Otherwise the C code and the XML file of the FMU are not consistent to each other. This argument cannot be null.

Argument `fmuResourceLocation` is an URI according to the IETF RFC3986 syntax to indicate the absolute path to the "Resource" directory of the unzipped FMU archive. The following protocols must be understood:

- Mandatory: "file://"
- Optional: "http://", "https://", "ftp://"
- Reserved "fmi://" for FMI for PLM.

[*Example: An FMU is unzipped in directory "C:\temp\MyFMU", then fmuResourceLocation = "file://C:/temp/MyFMU/resources". Functions fmiInstantiateModel and fmiInstantiateSlave are then able to read all needed resources from this directory, for example maps or tables used by the FMU.*]

Argument `functions` provide callback functions to be used from the FMU functions to utilize resources from the environment (see type `fmiCallbackFunctions` below).

Argument `visible = fmiFalse` defines that the interaction with the user should be reduced to a minimum (no application window, no plotting, no animation, etc.), in other words the FMU is executed in batch mode. If `visible = fmiTrue`, the FMU is executed in interactive mode and the FMU might require to explicitly acknowledge start of simulation / instantiation / initialization (acknowledgment is non-blocking).

If `loggingOn=fmiTrue`, debug logging is enabled. If `loggingOn=fmiFalse`, debug logging is disabled.

```
typedef struct {
        void  (*logger)(fmiComponentEnvironment componentEnvironment,
                        fmiString instanceName,
                        fmiStatus status,
                        fmiString category,
                    fmiString message, ...);
        void* (*allocateMemory)(size_t nobj, size_t size);
        void  (*freeMemory)    (void* obj);
        void  (*stepFinished)  (fmiComponentEnvironment componentEnvironment,
                                fmiStatus status);
        fmiComponentEnvironment componentEnvironment;
} fmiCallbackFunctions;
```

The struct contains pointers to functions provided by the environment to be used by the FMU. Additionally, a pointer to the environment is provided (componentEnvironment) that needs to be passed to the “logger” function, in order that the logger function can utilize data from the environment, such as mapping a `valueReference` to a string. In the unlikely case that fmiComponent is also needed in the logger, it has to be passed via argument componentEnvironment. Argument componentEnvironment may be a null pointer.
The componentEnvironment pointer is also passed to the `stepFinished(..)` function in order that the environment can provide an efficient way to identify the slave that called `stepFinished(..)`.

In the default `fmiFunctionTypes.h` file, `typedefs` for the function definitions are present to simplify the usage. This is non-normative. The functions have the following meaning:

Function **logger**:
Pointer to a function that is called in the FMU, usually if a `fmiXXX` function does not behave as desired. If “`logger`” is called with “`status = fmiOK`”, then the message is a pure information message. “`instanceName`” is the instance name of the model that calls this function. “`category`” is the category of the message. The meaning of “`category`” is defined by the modeling environment that generated the FMU. Depending on this modeling environment, none, some or all allowed values of “`category`” for this FMU are defined in the `modelDescription.XML` file via element “`fmiModelDescription.LogCategories`”, see section 0. Only messages are provided by function `logger` that have a category according to a call to `fmiSetDebugLogging` (see below). Argument “`message`” is provided in the same way and with the same format control as in function “`printf`” from the C standard library. [*Typically, this function prints the message and stores it optionally in a log file.*]

All string-valued arguments passed by the FMU to the logger may be deallocated by the FMU directly after function `logger` returns. The environment must therefore create copies of these strings if it needs to access these strings later.

The logger function will append a line break to each message when writing messages after each other to a terminal or a file (the messages may also be shown in other ways, for example as separate text-boxes in a GUI). The caller may include line-breaks (using "\n") within the message, but should avoid trailing line breaks.

Variables are referenced in a message with “#<Type><ValueReference>#” where <Type> is “r” for `fmiReal`, “i” for `fmiInteger`, “b” for `fmiBoolean` and “s” for `fmiString`. If character “#”shall be included in the message, it has to be prefixed with “#”, so “#” is an escape character. [*Example:*

A message of the form
  “`#r1365# must be larger than zero (used in IO channel ##4)`”
might be changed by the `logger` function to
  “`body.m must be larger than zero (used in IO channel #4)`”
if “`body.m`” is the name of the `fmiReal` variable with `fmiValueReference = 1365`.]

Function **allocateMemory**:

Pointer to a function that is called in the FMU if memory needs to be allocated. If attribute "`canNotUseMemoryManagementFunctions = true`" in `fmiModelDescription.ModelExchange / CoSimulation`, then function `allocateMemory` is not used in the FMU and a void pointer can be provided. If this attribute has a value of "`false`" (which is the default), the FMU must not use `malloc`, `calloc` or other memory allocation functions. One reason is that these functions might not be available for embedded systems on the target machine. Another reason is that the environment may have optimized or specialized memory allocation functions. `allocateMemory` returns a pointer to space for a vector of `nobj` objects, each of size "`size`" or `NULL`, if the request cannot be satisfied. The space is initialized to zero bytes [*(a simple implementation is to use `calloc` from the C standard library)*].

Function **freeMemory**:

Pointer to a function that must be called in the FMU if memory is freed that has been allocated with `allocateMemory`. If a null pointer is provided as input argument `obj`, the function shall perform no action [*(a simple implementation is to use `free` from the C standard library; in ANSI C89 and C99, the null pointer handling is identical as defined here)*]. If attribute "`canNotUseMemoryManagementFunctions = true`" in `fmiModelDescription.ModelExchange / CoSimulation`, then function `freeMemory` is not used in the FMU and a void pointer can be provided.Function **stepFinished**:
Optional call back function to signal if the computation of a communication step of a co-simulation slave is finished. A null pointer can be provided. In this case `fmiDoStep` has to be carried out synchronously. If a pointer to a function is provided, it must be called by the FMU after a completed communication step.

```
fmiStatus fmiSetDebugLogging(fmiComponent c, fmiBoolean loggingOn,
                             size_t nCategories, const fmiString categories[]);
```

If `loggingOn=fmiTrue`, debug logging is enabled, otherwise it is switched off.
If `loggingOn=fmiTrue` and `nCategories > 0`, then only debug messages according to the `categories` argument shall be printed via the logger function. Vector `categories` has `nCategories` elements. The allowed values of "`category`" are defined by the modeling environment that generated the FMU. Depending on the generating modeling environment, none, some or all allowed values for "`categories`" for this FMU are defined in the `modelDescription.XML` file via element "`fmiModelDescription.LogCategories`", see section 0.

2.1.6 Getting and Setting Variable Values

All variable values of an FMU are identified with a variable handle called "value reference". The handle is defined in the `modelDescription.XML` file (as attribute "`valueReference`" in element "`ScalarVariable`"). Element "`valueReference`" might not be unique for all variables. If two or more variables of the same base data type (such as `fmiReal`) have the same `valueReference`, then they have identical values but other parts of the variable definition might be different [*(for example min/max attributes)*].

The actual values of the variables that are defined in the `modelDescription.XML` file can be inquired after initialization of the FMU with the following functions:

```
fmiStatus fmiGetReal   (fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        fmiReal value[]);
```

```
fmiStatus fmiGetInteger(fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        fmiInteger value[]);
fmiStatus fmiGetBoolean(fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        fmiBoolean value[]);
fmiStatus fmiGetString (fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        fmiString value[]);
```

Get actual values of variables by providing the variable handles. [*These functions are especially used to get the actual values of output variables if a model is connected with other models. Since state derivatives are also ScalarVariables, it is possible to get the value of a state derivative. This is useful when connecting FMUs together. Furthermore, the actual value of every variable defined in the modelDescription.XML file can be determined at the actually defined time instant (see section 2.2.7).*]

- Argument "`vr`" is a vector of "`nvr`" value handles that define the variables that shall be inquired.
- Argument "`value`" is a vector with the actual values of these variables.
- The string returned by `fmiGetString` must be copied in the target environment, because the allocated memory for this string might be deallocated by the next call to any of the fmi interface functions or it might be an internal string buffer that is reused.
- Note: `fmiStatus = fmiDiscard` is possible for `fmiGetReal` only, but not for `fmiGetInteger`, `fmiGetBoolean`, `fmiGetString`, because these are discrete variables and their values can only change at an event instant where `fmiDiscard` does not make sense.

It is also possible to set the values of certain variables at particular instants in time using the following functions:

```
fmiStatus fmiSetReal   (fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        const fmiReal value[]);
fmiStatus fmiSetInteger(fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        const fmiInteger value[]);
fmiStatus fmiSetBoolean(fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        const fmiBoolean value[]);
fmiStatus fmiSetString (fmiComponent c, const fmiValueReference vr[], size_t nvr,
                        const fmiString value[]);
```

Set independent parameters, inputs, start values and re-initialize caching of variables that depend on these variables.

- Argument "`vr`" is a vector of "`nvr`" value handles that define the variables that shall be set.
- Argument "`value`" is a vector with the actual values of these variables.
- All strings passed as arguments to `fmiSetString` must be copied inside this function, because there is no guarantee of the lifetime of strings, when this function returns.
- Note: `fmiStatus = fmiDiscard` is only possible for `fmiSetReal` and `fmiSetInteger`.

Restrictions on using the "`fmiSetReal/Integer/Boolean/String`" functions

- For model exchange (see also section 3.2.4):
  1. The setting is with respect to the time defined with the last call to `fmiSetTime`.
  2. These functions shall not be called on constants (`ScalarVariable.variability="constant"`).
  3. These functions can be called on inputs (`ScalarVariable.causality = "input"`), and on tunable parameters (`causality="parameter"`, `variability="tunable"`), after calling `fmiInstantiateModel` and before `fmiFreeModel`.
  4. Additionally, these functions can be called on variables that have a "`ScalarVariable /`

<type> / `start`" attribute, after calling `fmiInstantiateModel` and before calling `fmiInitializeModel`. If these functions are not called on a variable with a "`start`" attribute, then the "`start`" value of this variable in the C functions is this "`start`" value (so this `start` value must be stored both in the XML file and in the C functions).

- For co-simulation (see also section 0):
  1. The setting is with respect to the beginning of a communication time step.
  2. These functions can only be called after calling `fmiInstantiateSlave` and before `fmiFreeSlave`.
  3. These functions shall not be called on constants (`ScalarVariable.variability = "constant"`).
  4. These functions can be called on inputs (`ScalarVariable.causality = "input"`), and on tunable parameters (`causality="parameter"`, `variability="tunable"`), after calling `fmiInstantiateSlave` and before `fmiTerminateSlave`.
  5. For non-tunable parameters (`ScalarVariable.causality = "parameter"`" and `ScalarVariable.variability = "fixed"`) the functions can only be called between `fmiInstantiateSlave` and `fmiInitializeSlave`.

  If no set function is called for a variable, it is initialized by the slave to its default value.

For co-simulation FMUs, additional functions are defined in section 4.2.3 to set and inquire derivatives of variables with respect to time in order to allow interpolation.

2.1.7 Getting and Setting the Complete FMU State

The FMU has an internal state consisting of all values that are needed to continue a simulation. This internal state consists especially of the values of the continuous states, discrete states, iteration variables, parameter values, input values, file identifiers and FMU internal status information. With the functions of this section, the internal FMU state can be copied and the pointer to this copy is returned to the environment. The FMU state copy can be set as actual FMU state, in order to continue the simulation from it. [Examples, for using this feature.*:*

- *For iterative for co-simulation master algorithms (get the FMU state for every accepted communication step; if the follow-up step is not accepted, restart co-simulation from this FMU state).*
- *For nonlinear Kalman filters (get the FMU state just before initialization; in every sample period, set new continuous states from the Kalman filter algorithm based on measured values; integrate to the next sample instant and inquire the predicted continuous states that are used in the Kalman filter algorithm as basis to set new continuous states).*
- *For nonlinear model predictive control (get the FMU state just before initialization; in every sample period, set new continuous states from an observer, initialize and get the FMU state after initialization. From this state, perform many simulations that are restarted after the initialization with new input signals proposed by the optimizer).*]

Furthermore, the FMU state can be serialized and copied in a byte vector: [*This can be, for example used to perform an expensive steady-state initialization, copy the received FMU state in a byte vector and store this vector on file. Whenever needed, the byte vector can be loaded from file, can be deserialized and the simulation is restarted from this FMU state, in other words from the steady-state initialization.*]

```
fmiStatus fmiGetFMUstate (fmiComponent c, fmiFMUstate* FMUstate);
fmiStatus fmiSetFMUstate (fmiComponent c, fmiFMUstate  FMUstate);
fmiStatus fmiFreeFMUstate(fmiComponent c, fmiFMUstate* FMUstate);
```

`fmiGetFMUstate` makes a copy of the internal FMU state and returns a pointer to this copy (`FMUstate`). If on entry *FMUstate == NULL, a new allocation is required. If *FMUstate != NULL, then *FMUstate points to a previously returned FMUstate that has not been modified since. In particular, fmiFreeFMUstate had not been called with this FMUstate as an argument. [*Function fmiGetFMUstate typically reuses the memory of this FMUstate in this case and returns the same pointer to it, but with the actual FMUstate.*]

`fmiSetFMUstate` copies the content of the previously copied `FMUstate` back and uses it as actual new FMU state. The `FMUstate` copy does still exist.

`fmiFreeFMUstate` frees all memory and other resources allocated with the `fmiGetFMUstate` call for this `FMUstate`. The input argument to this function is the `FMUstate` to be freed. If a null pointer is provided, the call is ignored. The function returns a null pointer in argument `FMUstate`.

These functions are only supported by the FMU, if the optional capability flag `canGetAndSetFMUstate` in `fmiModelDescription.ModelExchange / CoSimulation` in the XML file is explicitly set to `true` (see sections 3.3.1 and 4.3.1).

```
fmiStatus fmiSerializedFMUstateSize(fmiComponent c, fmiFMUstate FMUstate,
                                     size_t *size);
fmiStatus fmiSerializeFMUstate     (fmiComponent c, fmiFMUstate FMUstate,
                                     fmiByte serializedState[], size_t size);
fmiStatus fmiDeSerializeFMUstate   (fmiComponent c,
                                     const fmiByte serializedState[],
                                     size_t size, fmiFMUstate* FMUstate);
```

`fmiSerializedFMUstateSize` returns the `size` of the byte vector, in order that `FMUstate` can be stored in it. With this information, the environment has to allocate an `fmiByte` vector of the required length `size`.

`fmiSerializeFMUstate` serializes the data which is referenced by pointer `FMUstate` and copies this data in to the byte vector `serializedState` of length `size`, that must be provided by the environment.

`fmiDeSerializeFMUstate` deserializes the byte vector `serializedState` of length `size`, constructs a copy of the FMU state and returns `FMUstate`, the pointer to this copy. [*The simulation is restarted at this state, when calling fmiSetFMUState with FMUstate.*]

These functions are only supported by the FMU, if the optional capability flags `canGetAndSetFMUstate` and `canSerializeFMUstate` in `fmiModelDescription.ModelExchange / CoSimulation` in the XML file are explicitly set to `true` (see sections 3.3.1 and 4.3.1).

[*Typically, simulation environments and FMUs have restrictions on the calling of fmiSetFMUState. Especially, if a simulation is running, fmiSetFMUState can only set an FMU state back to a time instant that is before the current (integration) time instant and setting it for a future time instant is not possible.*]

2.1.8 Getting Partial Derivatives

It is optionally possible to provide evaluation of partial derivatives in an FMU. For Model Exchange, this means computing the partial derivatives at a particular time instant. For Co-Simulation, this means to compute the partial derivatives at a particular communication point. Two functions are provided (1) to compute the partial derivative matrices and (2) to compute directional partial derivatives.

```
fmiStatus fmiGetPartialDerivatives(fmiComponent c,
                   fmiStatus (*setMatrixElement)(
                                 fmiComponentEnvironment componentEnvironment,
                                 void* data, fmiInteger row,
                                 fmiInteger col, fmiReal value),
                     void* A, void* B, void* C, void* D)
```

The input/output behavior of the continuous part of an FMU is defined by the following equations:

$$\frac{d\mathbf{x}}{dt}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$

$$\mathbf{y}_{cont}(t) = \mathbf{g}_{cont}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$

where $\mathbf{u}_{cont}$ are continuous inputs of type `Real`, $\mathbf{u}_{disc}$ are discrete inputs, $\mathbf{x}$ are continuous states, $\mathbf{y}_{cont}$ are continuous outputs of type `Real`, *t* is time. [*Since partial derivatives are computed by this function only for continuous variables, the input vector* $\mathbf{u}_{cont}$ *and the output vector* $\mathbf{y}_{cont}$ *contain here only variables with* `variability = "continuous"` *and type* = `"Real"`.]
`fmiGetPartialDerivatives` provides the partial derivatives of the functions **f** and **g** with respect to their continuous arguments, provided the respective capability flags defined with the attributes of `fmiModelDescription.ModelExchange` or `.CoSimulation` are set to true.

Argument A_{ij} returns $\frac{\partial f_i}{\partial x_j}$, argument B_{ij} returns $\frac{\partial f_i}{\partial u_{cont,j}}$,

argument C_{ij} returns $\frac{\partial g_{cont,i}}{\partial x_j}$, argument D_{ij} returns $\frac{\partial g_{cont,i}}{\partial u_{cont,j}}$.

The input and output vectors are ordered according to `fmiModelDescription.ModelStructure` where all inputs and outputs that do not have `variability = "continuous"` and type "`Real`" are removed.
Argument "c" is an FMU instance. Arguments `A, B, C, D` are pointers to data structures defined by the calling environment that are used to represent the partial derivative matrices. If one of these arguments is a null pointer on entry, no data structure is provided and the function shall not compute the respective partial derivatives.

The matrices can be defined as dense or as sparse matrices in a format of choice [*(for example sparse triplet, column compressed storage or row compressed storage)].* The FMU interacts with the matrix data structures by calling the function:

```
   fmiStatus (*setMatrixElement)(void* data, fmiInteger row,
                                 fmiInteger col, fmiReal value)
```

This function is called by the FMU to set an entry of a Jacobian matrix. The arguments of this function are:

- componentEnvironment: The componentEnvironment argument provided to the FMU when instantiating it (see section 2.1.5). [*In case of error, this argument might be used to print a message in the environment.*]
- `data`: The `A, B, C,` or `D` argument.
- `row`: Row index of the element to be set (`row` ≥ 1; the first row element has index 1).
- `col`: Column index of the element to be set (`col` ≥ 1; the first column element has index 1).
- `value`: The matrix element value.

Values that are not explicitly set are assumed to be zero [*in other words the environment has to initialize data with zero*].

[`fmiGetPartialDerivatives` *may compute the partial derivatives by numerical differentiation taking into account the sparseness of the matrix, or (preferred) by analytic derivatives via the automatic differentiation method. The computed partial derivative matrices can be utilized for the following purposes:*

- *Numerical integrators of stiff methods need the* `A` *matrix.*

- *If the FMU is connected with other FMUs, the* `A,B,C,D` *matrices are needed in order to compute the* `A`*-matrix for the system of the connected FMUs.*
- *If the FMU shall be linearized, the* `A,B,C,D` *matrices are needed.*
- *If the FMU is used as the model for an extended Kalman filter, matrices* `A` *and* `C` *are needed.*

]

```
fmiStatus fmiGetDirectionalDerivative(fmiComponent c,
                                    const fmiValueReference v_ref[], size_t nv,
                                    const fmiValueReference z_ref[], size_t nz,
                                    const fmiReal dv[],
                                          fmiReal dz[])
```

If the capability attribute "`providesDirectionalDerivatives`" is `true`, this function can be called to compute the directional derivatives with respect to the continuous variables of the equations:

$$\frac{d\mathbf{x}}{dt}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$
$$\mathbf{y}_{cont}(t) = \mathbf{g}_{cont}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$

A subset of this equation system is defined as:

$$\mathbf{z} = \mathbf{h}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$

where

- **z** is a vector with elements of the state derivatives and the outputs, in other words $\dot{x}_i$ and/or $y_{cont,j}$.
- **h** is the corresponding vector of functions for computing the state derivatives and the outputs contained in the vector **z**, in other words f_i and/or g_j.

`fmiGetDirectionalDerivative` computes a linear combination of the partial derivatives of **h** with respect to the selected independent continuous-time variables **v** (in other words **v** is a subset of **x**, and **u**$_{cont}$):

$$\Delta\mathbf{z} = \frac{\partial \mathbf{h}}{\partial \mathbf{v}} \Delta\mathbf{v}$$

Accordingly, it computes the directional derivative vector $\Delta\mathbf{z}$ (`dz`) from the seed vector $\Delta\mathbf{v}$ (`dv`).

[*Example:*
Assume an FMU has the output equations

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} g_1(x, u_1, u_3, u_4) \\ g_2(x, u_1) \end{bmatrix}$$

and this FMU is connected, so that y_1,u_1,u_3 appear in an algebraic loop. Then the nonlinear solver needs a Jacobian and this Jacobian can be computed (without numerical differentiation) provided the partial derivative of y_1 with respect to u_1 and u_3 is available. Depending on the environment where the FMUs are connected, these derivatives can be provided (a) with one function call to compute the directional derivatives with respect to these two variables (in other words $z=y_1$, $v = \{u_1, u_3\}$) and then the environment uses appropriate seed values zero and one, or two function calls are provided (in other words $z_1= z_2=y_1$, $v_1=u_1$, $v_2=u_3$) with seed values of one.

Note, a direct implementation of this function provides (a) the directional derivative for all independent continuous variables, (b) sets the seed-values for all these variables to zero, (c) changes the seed-value of the z-variables to the given values, (d) computes the directional

derivative for all independent continuous variables with the set seed-values.
]

2.2 FMI Description Schema

All static information related to an FMU is stored in the text file `modelDescription.XML` in XML format. Especially, the FMU variables and their attributes such as name, unit, default initial value etc. are stored in this file. The structure of this XML file is defined with the schema file "`fmiModelDescription.xsd`". This schema file utilizes the following helper schema files:

```
fmiAnnotation.xsd
fmiScalarVariable.xsd
fmiType.xsd
fmiVariableDependency.xsd
fmiUnit.xsd
```

In this section these schema files are discussed. The normative definition are the above mentioned schema files[4]. Below, optional elements are marked with a "dashed" box. The required data types (like: xs:normalizedString) are defined in the XML-schema standard: http://www.w3.org/TR/XMLschema-2/. The types used in the fmi schema files are:

XML	Description (http://www.w3.org/TR/XMLschema-2/)	*Mapping to C*
xs:double	IEEE double-precision 64-bit floating point type [*In order to not loose precision, a number of this type should be stored on an XML file with at least **16** significant digits; for example 2/3 should be stored as `0.6666666666666667`*]	double
xs:int	Integer number with maximum value 2147483647 and minimum value -2147483648 (32 bit Integer)	int
xs:unsignedInt	Integer number with maximum value 4294967295 and minimum value 0 (unsigned 32 bit Integer)	unsigned int
xs:boolean	Boolean number. Legal literals: false, true, 0, 1	char
xs:string	Any number of characters	char*
xs:normalizedString	String without carriage return, line feed, and tab characters	char*
xs:dateTime	Date, time and time zone (for details see the link above). Example: 2002-10-23T12:00:00Z (noon on October 23, 2002, Greenwich Mean Time)	tool specific

The first line of an XML file must contain the encoding scheme of the XML file, such as:

```
<?XML version="1.0" encoding="UTF-8"?>
```

A specific encoding scheme is not required for the fmi XML file (modelDescription.xml). Typical schemes are "ISO-8859-1" or "UTF-8". The FMI schema files (*.xsd) are stored in "UTF-8". Note, the definition of an encoding scheme is a prerequisite, in order for the XML file to contain letters outside of the 7 bit ANSI ASCII character set, such as German umlauts, or Asian characters. If another encoding scheme than "UTF-8" is used, then the non-ASCII characters in string variables need to be transformed to UTF-8

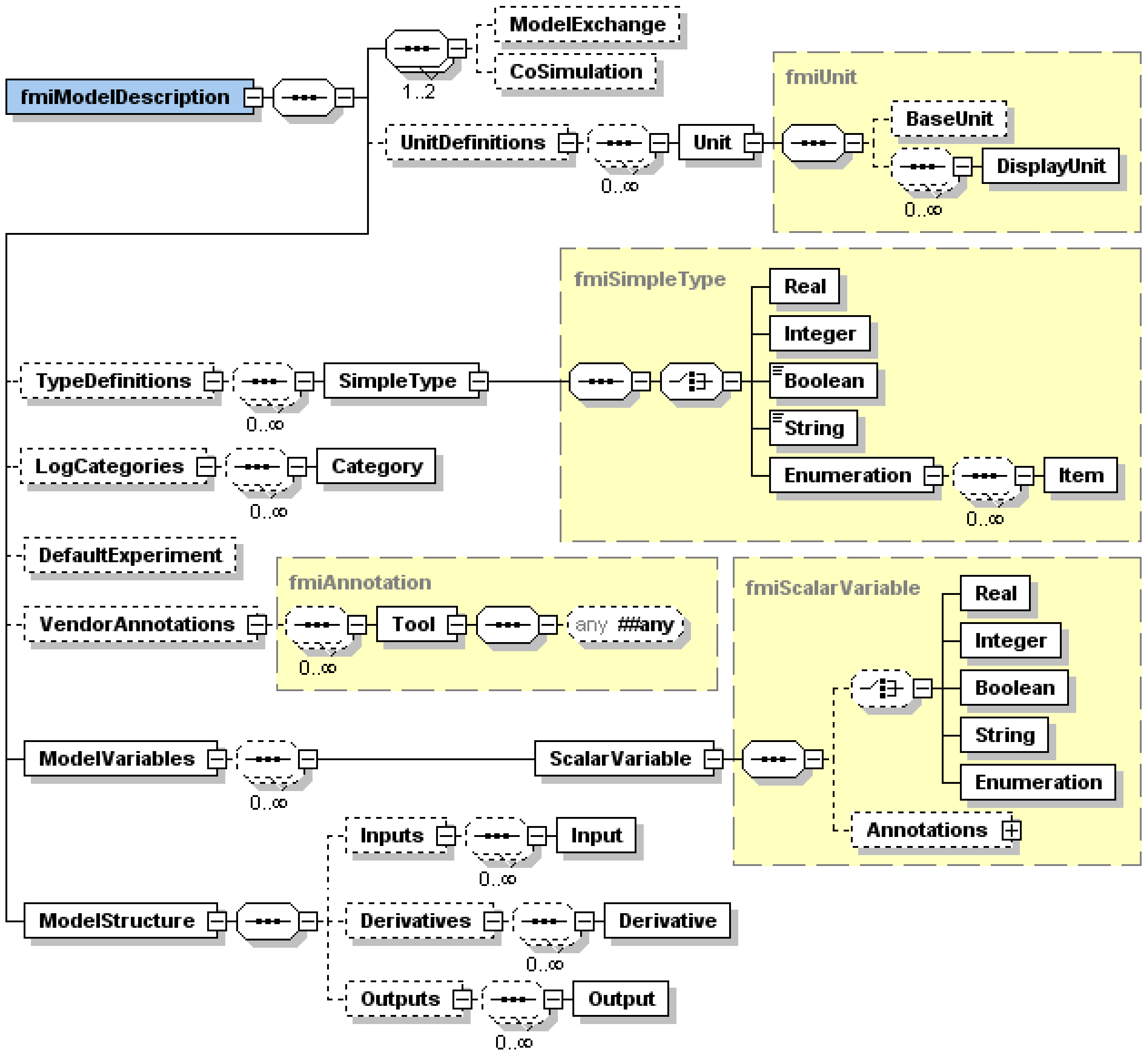
[4] Note, the screenshots of this section have been generated from the schema files with the tool "Altova XMLSpy" (www.altova.com). With the enterprise edition of XMLSpy it is possible to automatically generate C++, C# and Java code that reads an XML file of fmiModelDescription.xsd. An efficient open source XML parser is SAX (http://sax.sourceforge.net/, http://en.wikipedia.org/wiki/Simple_API_for_XML). All data from the XML file is only defined via "attributes" and not via "elements". Therefore, only an "attribute" handler needs to be defined for a SAX parser.

when reading them from file, because the FMI calling interface requires that strings are encoded in UTF-8.

Note, child information items, such as “elements” in a “sequence” are **ordered lists** according to document order, whereas attribute information items are **unordered sets** (see http://www.w3.org/TR/XML-infoset/#infoitem.element). The FMI schema is based on ordered lists in a sequence and therefore parsing must preserve this order. [*For example the information stored in* `ModelVariables.Derivatives` *is only correct if this property is fulfilled*].

2.2.1 Definition of an FMU (fmiModelDescription)

This is the root-level schema file and contains the following definition (the figure below contains all elements in the schema file. Data is defined by attributes to these elements):



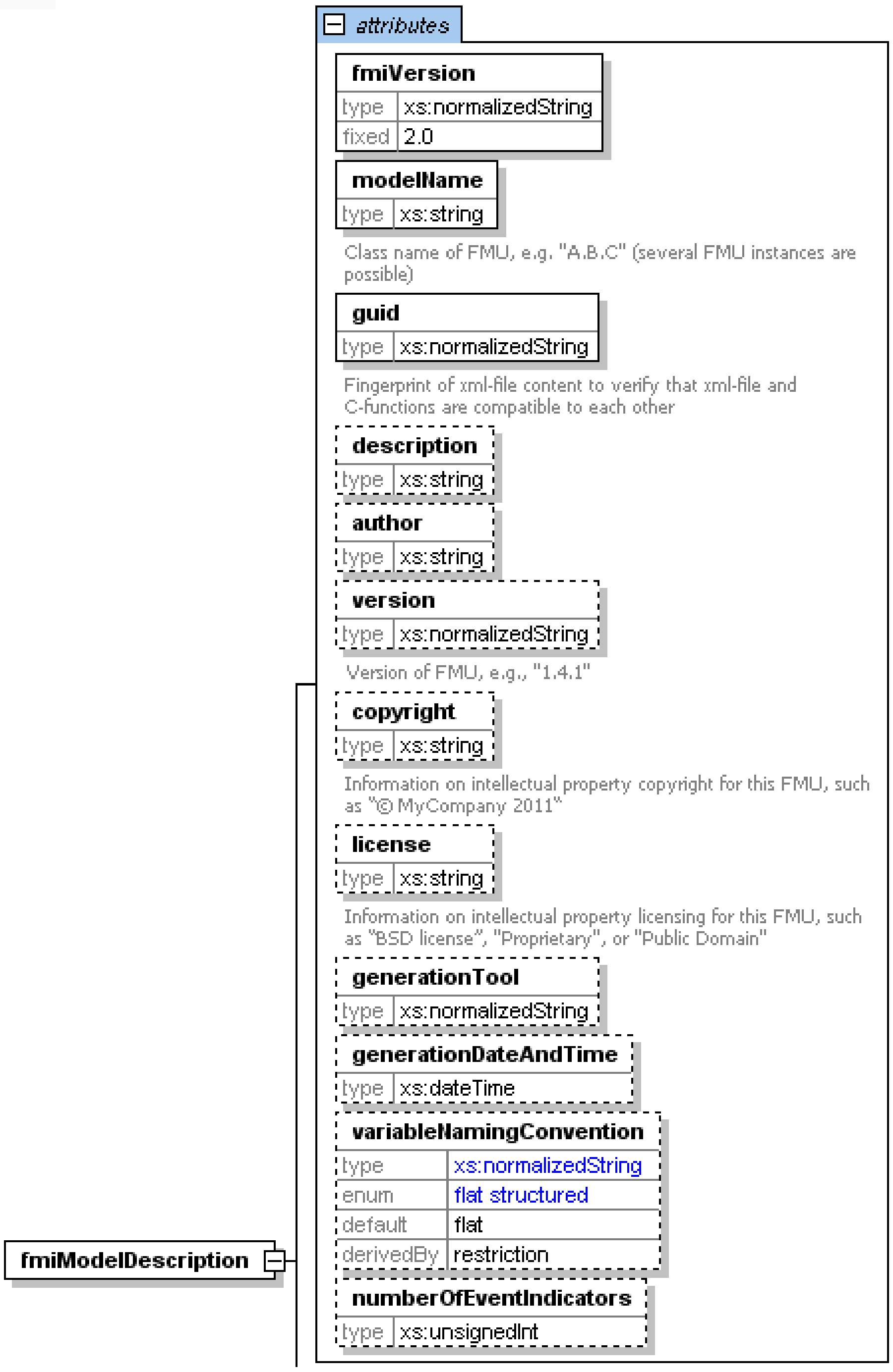
On the top level, the schema consists of the following elements (see figure above):

Element-Name	*Description*
`ModelExchange`	If present, the FMU is based on “FMI for Model Exchange” [(*in other words the FMU includes the model or the communication to a tool*

	that provides the model, and the environment provides the simulation engine)].
`CoSimulation`	If present, the FMU is based on "FMI for Co-Simulation" [*(in other words the FMU includes the model* <u>*and*</u> *the simulation engine, or a communication to a tool that provides the model and the simulation engine, and the environment provides the master algorithm to run coupled FMU co-simulation slaves together)*].
`UnitDefinitions`	A global list of unit and display unit definitions [*for example to convert display units into the units used in the model equations*]. These definitions are used in the XML element "`ModelVariables`".
`TypeDefinitions`	A global list of type definitions that are utilized in "`ModelVariables`".
`LogCategories`	A global list of log categories that can be set to define the log information that is supported from the FMU.
`DefaultExperiment`	Providing default settings for the integrator, such as stop time and relative tolerance.
`VendorAnnotations`	Additional data that a vendor might want to store and that other vendors might ignore.
`ModelVariables`	The central FMU data structure defining all variables of the FMU that are visible/accessible via the FMU functions.
`ModelStructure`	Defines the structure of the model. Especially, the ordered lists of inputs, states and outputs is defined here. Furthermore, the dependency of the unkowns from the knowns can be optionally defined. [*This information can be, for example used to compute efficiently a sparse Jacobian for simulation or to utilize the input/output dependency in order to detect that in some cases there are actually no algebraic loops when connecting FMUs together*].

At least one element of `ModelExchange` or `CoSimulation` must be present to identify the type of the FMU. If both elements are defined, different types of models are included in the FMU. The details of these elements are defined in section 3.3.1 and section 4.3.1.

The XML attributes of `fmiModelDescription` are:



Attribute-Name	*Description*
`fmiVersion`	Version of "FMI for Model Exchange or Co-Simulation" that was used to generate the XML file. The value for this version is "2.0".
`modelName`	The name of the model as used in the modeling environment that generated the XML file, such as "Modelica.Mechanics.Rotational.Examples.CoupledClutches".
`guid`	The "Globally Unique IDentifier" is a string that is used to check that the XML file is compatible with the C functions of the FMU. Typically when generating the XML file, a fingerprint of the "relevant" information is stored as guid and in the generated C-function.
`description`	Optional string with a brief description of the model.

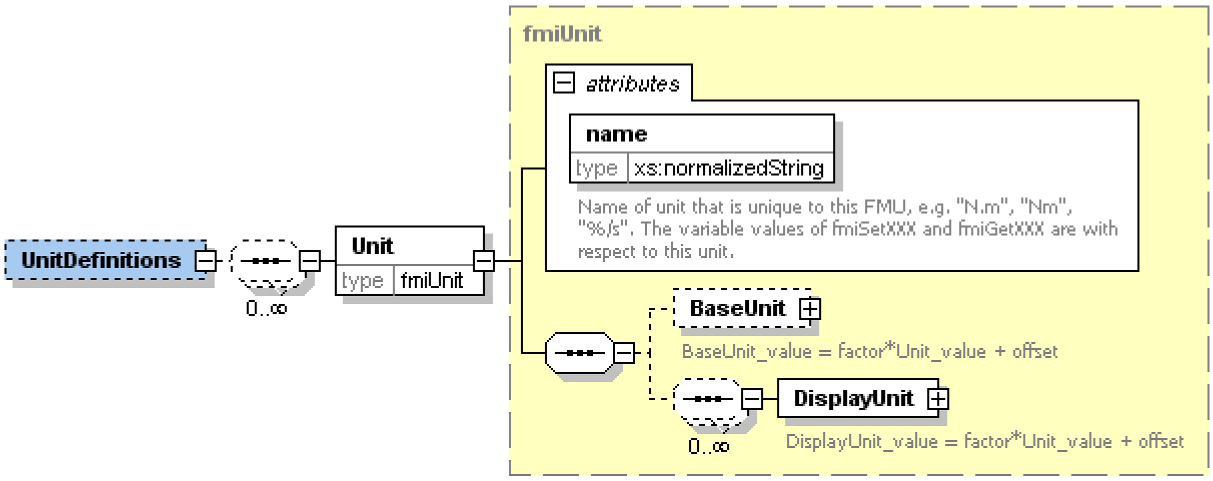
`author`	Optional string with the name and organization of the model author.
`version`	Optional version of the model, for example "1.0".
`copyright`	Optional information on the intellectual property copyright for this FMU. [*Example: copyright = "© My Company 2011"*].
`license`	Optional information on the intellectual property licensing for this FMU. [*Example: license = "BSD license <license text or link to license>"*].
`generationTool`	Optional name of the tool that generated the XML file.
`generationDateAndTime`	Optional date and time when the XML file was generated. The format is a subset of "xs:dateTime" and should be: "YYYY-MM-DDThh:mm:ssZ" (with one "T" between date and time; "Z" characterizes the Zulu time zone, in other words Greenwich meantime). [*Example: `"2009-12-08T14:33:22Z"`*].
`variableNamingConvention`	Defines whether the variable names in "`ModelVariables / ScalarVariable / name`" and in "`TypeDefinitions / Type / name`" follow a particular convention. For the details, see section 2.2.9. Currently standardized are: • "*`flat`*": A list of strings (the default). • "`structured`": Hierarchical names with "." as hierarchy separator, and with array elements and derivative characterization.
`numberOfEventIndicators`	The (fixed) number of event indicators for an FMU based on FMI for Model Exchange. For Co-Simulation, this value is ignored.

[*Attributes "numberOfScalarVariables", "numberOfContinuousStates", "numberOfInputs", "numberOfOutputs" available in FMI 1.0 have been removed for FMI 2.0, since this information can be deduced from the remaining data in the XML file]*

2.2.2 Definition of Units (fmiUnit)

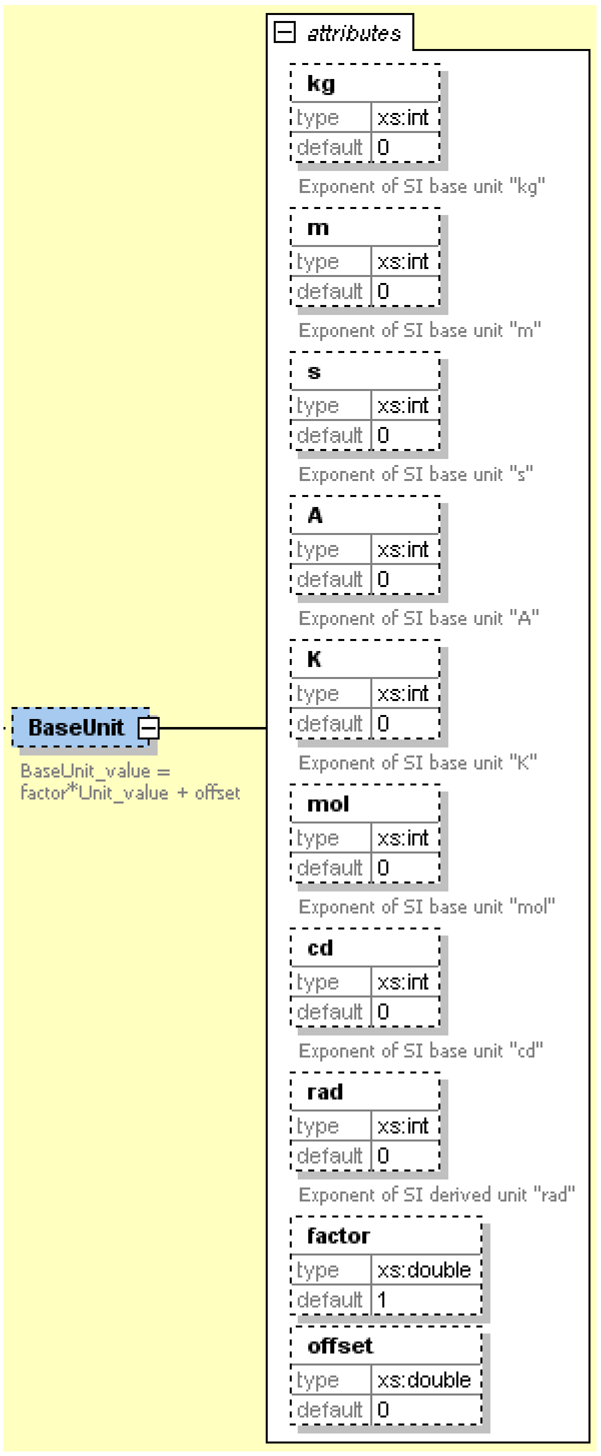
[*In this section the units of the variables are (optionally) defined. Unit support is important for technical systems since otherwise it is very easy for errors to occur. Unit handling is a difficult topic and there seems to be no method available that is really satisfactory for all applications such as unit check, unit conversion, unit propagation or dimensional analysis. In FMI a pragmatic approach is used that takes into account that every software system supporting units has potentially its own specific technique to describe and utilize units. The approach used here is slightly different to FMI 1.0 to reduce the need for standardized string representations.*]

Element "**`UnitDefinitions`**" of `fmiModelDescription` is defined as:


fmiUnit
attributes
name
type xs:normalizedString
Name of unit that is unique to this FMU, e.g. "N.m", "Nm", "%/s". The variable values of fmiSetXXX and fmiGetXXX are with respect to this unit.
UnitDefinitions
0..∞
Unit
type fmiUnit
BaseUnit
BaseUnit_value = factor*Unit_value + offset
DisplayUnit
0..∞
DisplayUnit_value = factor*Unit_value + offset

It consists of zero or more `Unit` definitions. A `Unit` is defined by its `name` attribute such as “N.m” or “N*m” or “Nm”, which must be unique with respect to all other defined `Units`. If a variable is associated with a `Unit`, then the value of the variable has to be provided with the `fmiSetXXX` functions or is returned by the `fmiGetXXX` functions with respect to this `Unit`. [*The purpose of the name is to uniquely identify a unit and, for example use it to display the unit in menus or in plots. Since there is no standard to represent units in strings, and there are different ways how this is performed in different tools, no specific string representation of the unit is required.*]

Optionally, a value given in unit `Unit` can be converted to a value with respect to unit `BaseUnit` utilizing the conversion `factor` and `offset` attributes:


attributes
kg
type xs:int
default 0
Exponent of SI base unit "kg"
m
type xs:int
default 0
Exponent of SI base unit "m"
s
type xs:int
default 0
Exponent of SI base unit "s"
A
type xs:int
default 0
Exponent of SI base unit "A"
K
type xs:int
default 0
Exponent of SI base unit "K"
mol
type xs:int
default 0
Exponent of SI base unit "mol"
cd
type xs:int
default 0
Exponent of SI base unit "cd"
rad
type xs:int
default 0
Exponent of SI derived unit "rad"
factor
type xs:double
default 1
offset
type xs:double
default 0
BaseUnit
BaseUnit_value = factor*Unit_value + offset

Besides `factor` and `offset`, the `BaseUnit` definition consists of the exponents of the 7 SI base units "kg", "m", "s", "A", "K", "mol", "cd", and of the exponent of the SI derived unit "rad". [*Depending on the analysis/operation carried out, the SI derived unit "rad" is or is not utilized, see discussion below. The additional "rad" base unit helps to handle the often occurring quantities in technical systems that depend on an angle.*]

A value with respect to `Unit` (abbreviated as "Unit_value") is converted with respect to `BaseUnit` (abbreviated as "BaseUnit_value") by the equation:

BaseUnit_value = `factor`*Unit_value + `offset`

[For example if p_{bar} is a pressure value in Unit "bar" and p_{Pa} is the pressure value in BaseUnit, then

$$p_{Pa} = 10^5\, p_{bar}$$

and therefore `factor = 1.0e5` *and* `offset = 0.0`.

In the following table several unit examples are given (note, if in column "`exponents`*" the definition "kgm^2/s^2"is present, then the attributes of* `BaseUnit` *are: "*`kg=1, m=2, s=-2`*"):*

Quantity	**Unit.name** (examples)	**Unit.BaseUnit**		
		exponents	**factor**	**offset**
Torque	"N.m"	$kg \cdot m^2 / s^2$	1.0	0.0
Energy	"J"	$kg \cdot m^2 / s^2$	1.0	0.0
Pressure	"bar"	$\frac{kg}{m \cdot s^2}$	1.0e5	0.0
Angle	"deg"	rad	0.01745329251994330 (= pi/180)	0.0
Angular velocity	"rad/s"	rad/s	1.0	0.0
Angular velocity	"rpm"	rad/s	0.1047197551196598 (=2*pi/60)	0.0
Frequency	"Hz"	rad/s	6.283185307179586 (= 2*pi)	0.0
Temperature	"°F"	K	0.5555555555555556 (= 5/9)	255.3722222222222 (= 273.15-32*5/9)
Per cent by length	"%/m"	1/m	0.01	0.0
Parts per million	"ppm"	1	1.0e-6	0.0
Length	"km"	m	1000	0.0
Length	"yd"	m	0.9144	0.0

Note, "Hz" is typically used as `Unit.name` *for a frequency quantity, but it can also be used as* `DisplayUnit` *for an angular velocity quantity (since "*`revolution/s`*").*

The `BaseUnit` *definitions can be utilized for different purposes (the following application examples are optional and a tool may also completely ignore the* `Unit` *definitions):*

Signal connection check:

When two signals v1 and v2 are connected together, and on at least one of the signals no `BaseUnit` *element is defined, then the connection equation "v2 = v1" holds (if v1 is an output of an FMU and v2 is an input of another FMU, with* `fmiGetXXX` *the value of v1 is inquired and used as value for v2 by calling* `fmiSetXXX`*).*

When two signals v1 and v2 are connected together, and for both of them `BaseUnit` *elements are defined, then they must have identical exponents of their* `BaseUnit`*. If* `factor` *and* `offset` *are also identical, again the connection equation "v2 = v1" holds. If* `factor` *and* `offset` *are not identical, the tool may either trigger an error or, if supported, perform a conversion, in other words use the connection equation (in this case the* `relativeQuantity` *of the* `TypeDefinition`*, see below, has to be taken into account in order to determine whether* `offset` *shall or shall not be utilized, since absolute or relative quantities):*

```
factor(v1)*v1 + offset(v1) = factor(v2)*v2 + offset(v2)
```

As a result, wrong connections can be detected (for example connecting a force with an angle signal would trigger an error) and conversions between, say, US and SI units can be either automatically performed or, if not supported, an error is triggered as well. Note, this approach is not satisfactory for variables belonging to different quantities that have, however, the same `BaseUnit`*, such as quantities "Energy" and "Torque", or "AngularVelocity" and "Frequency". To handle such cases quantity definitions have to be taken into account (see TypeDefinitions) and quantity names need to be standardized.*

*This approach allows a general treatment of units, without being forced to standardize the grammar and allowed values for units (for example in FMI 1.0, a unit could be defined as "N.m" in one FMU and as "N*m" in another FMU and a tool would have to reject a connection, since the units are not identical, In FMI 2.0 the connection would be accepted, provided both elements have the same BaseUnit definition).*

Dimensional analysis of equations:

In order to check the validity of equations in a modeling language, the defined units can be used for dimensional analysis, by using the `BaseUnit` *definition of the respective unit. For this purpose, the* `BaseUnit` *"rad" has to be treated as "1". Example:*

$J*\alpha = \tau \rightarrow [kg.m^2]*[rad/s^2] = [kg.m^2/s^2]$). // o.k. ("rad" is treated as "1")
$J*\alpha = f \rightarrow [kg.m^2]*[rad/s^2] = [kg.m/s^2]$). // error, since dimensions do not agree

Unit propagation:

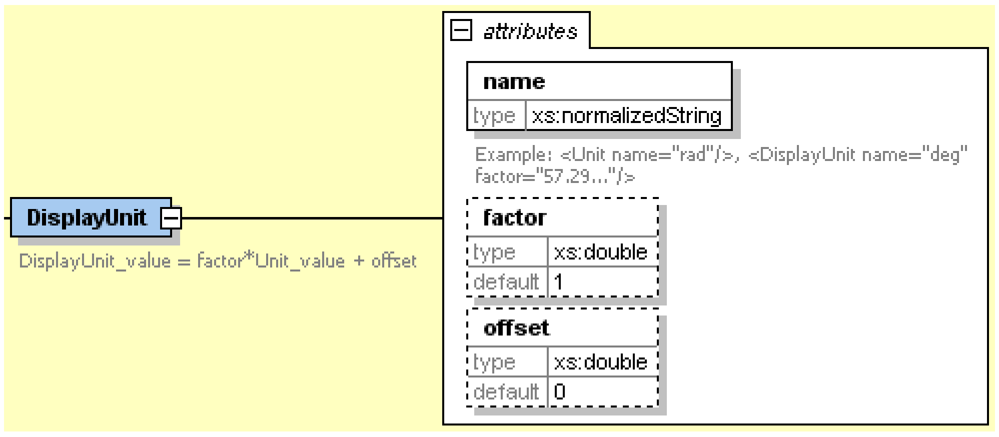
If unit definitions are missing for signals, they might be deduced from the equations where the signals are used. If no unit computation is needed, "rad" is propagated. If a unit computation is needed and one of the involved units has "rad" as a `BaseUnit`*, then unit propagation is not possible. Examples:*

- *a = b + c, and* `Unit` *of c is provided, but not* `Unit` *of a and b:*
  The `Unit` *definition of c (in other words* `Unit.name`*,* `BaseUnit`*,* `DisplayUnit`*) is also used for a and b. For example if BaseUnit(c) = "rad/s", then BaseUnit(a) = BaseUnit(b) = "rad/s".*

- *a = b*c, and* `Unit` *of a and of c is provided, but not* `Unit` *of b:*
  If "rad" is either part of the BaseUnit of "a" and/or of "c", then the BaseUnit of b cannot be deduced (otherwise it can be deduced). Example: If BaseUnit(a)="kg.m/s^2" and BaseUnit(c)="m/s^2", then the BaseUnit(b) can be deduced to be "kg". In such a case `Unit.name` *of b cannot be deduced from the* `Unit.name` *of a and c, and a tool would typically construct the* `Unit.name` *of b from the deduced* `BaseUnit`*.*

]

Additionally to the unit definition, optionally a set of display units can be defined that can be utilized for input/output of a value:


attributes
name
type xs:normalizedString
Example: <Unit name="rad"/>, <DisplayUnit name="deg" factor="57.29..."/>
DisplayUnit
DisplayUnit_value = factor*Unit_value + offset
factor
type xs:double
default 1
offset
type xs:double
default 0

A value with respect to Unit (abbreviated as “Unit_value”) is converted with respect to DisplayUnit (abbreviated as “DisplayUnit_value”) by the equation:

DisplayUnit_value = `factor*`Unit_value + `offset`

[*“offset” is, for example needed for temperature units.*]

[*For example if T_K is the temperature value of Unit.name (in “K”) and T_F is the temperature value of DisplayUnit (in “^{o}F”), then*

$$T_F = (9/5) * (T_K - 273.15) + 32$$

and therefore `factor = 1.8 (=9/5)` and `offset = -459.67 (= 32 – 273.15*9/5).`

Both the `DisplayUnit.name` definitions as well as the `Unit.name` definitions are used in the ScalarVariable elements. Example for a definition:

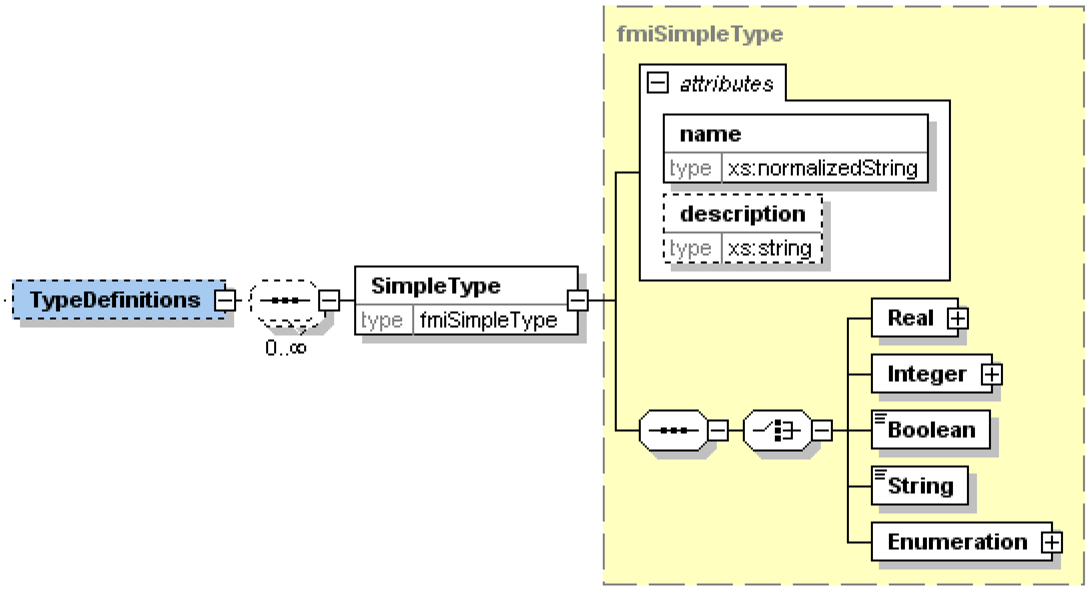
```
<Unit name="rad/s">
   <BaseUnit s="-1" rad="1"/>
   <DisplayUnit name="deg/s" factor= "57.29577951308232"/>
   <DisplayUnit name="rev/min" factor= "9.549296585513721"/>
</Unit>

<Unit name="bar">
   <BaseUnit kg="1", m="-1", s="-2", factor="1.0e5", offset="0"/>
</Unit>

<Unit name="Re">
   <BaseUnit/>    // unit = "1"
                  //(dimensionless, all exponents of BaseUnit are zero)
</Unit>

<Unit name="Euro/PersonYear"/>  // no mapping to BaseUnit defined
```

]

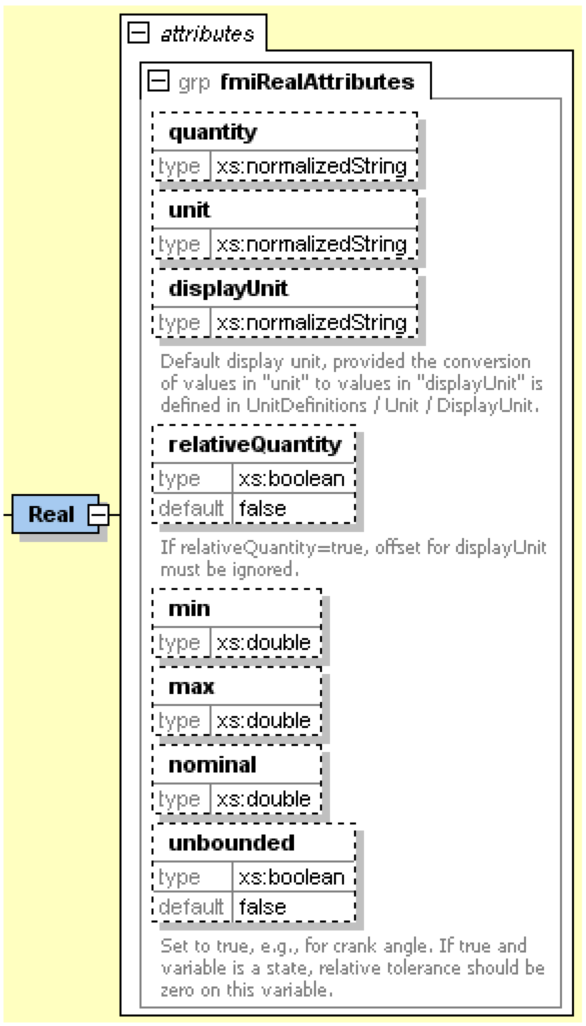
The schema definition is present in a separate file “`fmiUnit.xsd`”.

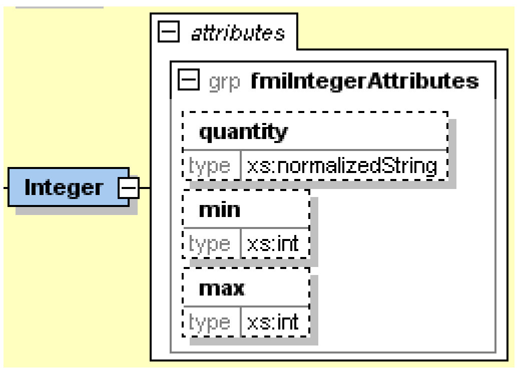
2.2.3 Definition of Types (fmiType)

Element “**`TypeDefinitions`**” of `fmiModelDescription` is defined as:

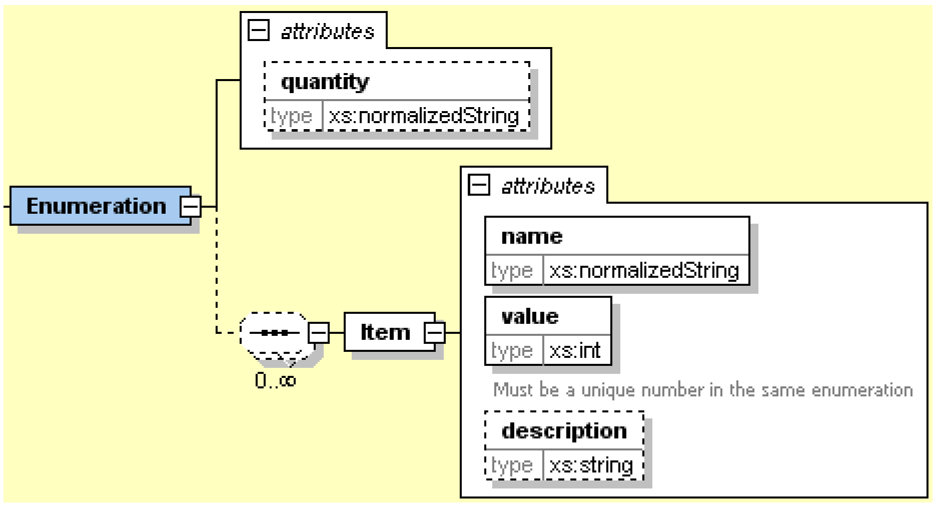


This element consists of a set of "`SimpleType`" definitions according to schema "`fmiSimpleType`" in file "`fmiType.xsd`". One "`SimpleType`" has a type "`name`" and "`description`" as attributes and one of `Real`, `Integer`, `Boolean`, `String`, or `Enumeration` must be present. The latter have the following definitions:


Real
attributes
grp fmiRealAttributes
quantity
type xs:normalizedString
unit
type xs:normalizedString
displayUnit
type xs:normalizedString
Default display unit, provided the conversion of values in "unit" to values in "displayUnit" is defined in UnitDefinitions / Unit / DisplayUnit.
relativeQuantity
type xs:boolean
default false
If relativeQuantity=true, offset for displayUnit must be ignored.
min
type xs:double
max
type xs:double
nominal
type xs:double
unbounded
type xs:boolean
default false
Set to true, e.g., for crank angle. If true and variable is a state, relative tolerance should be zero on this variable.



[The attributes of "Real" and "Integer" are collected in the attribute groups "fmiRealAttributes" and "fmiIntegerAttributes" in file "fmiAttributeGroups.xsd", since these attributes are reused in the ScalarVariable element definitions below.]



These definitions are used as default values in element `ScalarVariables`[, *in order that, say, the definition of a "Torque" type does not have to be repeated over and over again*]. The attributes and elements have the following meaning:

Name	*Description*
`quantity`	Physical quantity of the variable, for example "Angle", or "Energy". The quantity names are not standardized.
`unit`	Unit of the variable defined with `UnitDefinitions.Unit.name` that is used for the model equations *[, for example "N.m": in this case a `Unit.name = "N.m"` must be present under UnitDefinitions*].
`displayUnit`	Default display unit. The conversion to the "`unit`" is defined with the element "`fmiModelDescription / UnitDefinitions`". If the corresponding "`displayUnit`" is not defined here, then "`unit`" is used for input/output and `displayUnit` is ignored.
`relativeQuantity`	If this attribute is `true`, then the "`offset`" of "`displayUnit`" must be ignored (for example 10 degree Celsius = 10 Kelvin if "`relativeQuantity = true`" and not 283,15 Kelvin).
`min`	Minimum value of variable (variable Value ≥ `min`). If not defined, the minimum is the largest negative number that can be represented on the machine. The min definition is an information from the FMU to the environment defining the region in which the FMU is designed to operate, see also comment after this table.
`max`	Maximum value of variable (variableValue ≤ `max`). If not defined, the maximum is the largest positive number that can be represented on the machine. The max definition is an information from the FMU to the environment defining the region in which the FMU is designed to operate, see also comment after this table.
`nominal`	Nominal value of variable. If not defined and no other information about the nominal value is available, then nominal = 1 is assumed. *[The nominal value of a variable can be, for example used to determine the*

	absolute tolerance for this variable as needed by numerical algorithms: *absoluteTolerance* = `nominal`**relativeTolerance*0.01*]
`unbounded`	If true, indicates that the variable gets during time integration much larger than its nominal value `nominal`. [*Typical examples are the monotonically increasing rotation angles of crank shafts and the longitudinal position of a vehicle along the track in long distance simulations. This information can, for example, be used to increase numerical stability and accuracy by setting the corresponding bound for the relative error to zero (relative tolerance = 0.0), if the corresponding variable or an alias of it is a continuous state variable.*]
`Item`	Items of an enumeration as a sequence of "`name`" and "`value`" pairs. The values can be any integer number, but must be unique within the same enumeration (in order that the mapping between "`name`" and "`value`" is bijective).

[A*ttributes „min" and „max" can be set for variables of type Real, Integer or Enumeration. The question is how fmiSetReal, fmiSetInteger, fmiGetReal, fmiGetInteger shall utilize this definition. There are several conflicting requirements:*

- *Avoiding forbidden regions (e.g. if „u" is an input and „sqrt(u)" is computed in the FMU, min=0 on „u" shall guarantee that only values of „u" in the allowed regions are provided).*
- *Numerical algorithms (ODE-solver, optimizers. nonlinear solvers) do not guarantee constraints. If a variable is outside of the bounds, the solver tries to bring it back into the bounds. As a consequence, calling fmiGetReal during an iteration of such a solver might return values that are not in the defined min/max region. After the iteration is finalized, it is only guaranteed that a value is within its bounds upto a certain numerical precision.*
- *In debug mode checks on min/max should be performed. For maximum performance on a real-time system the checks might not be performed.*

The approach in FMI is therefore that min/max definitions are an information from the FMU to the environment defining the region in which the FMU is designed to operate. The environment is free to utilize this information (typically, in debug mode of the environment the min/max is checked in the cases as stated above). In any case, it is expected that the FMU handles variables appropriately where the region definition is critical. For example, dividing by an input (so the input should not be in a small range of zero) or taking the square root of an input (so the input should not be negative) may either result in fmiError, or the FMU is able to handle this situation in other ways.

If the FMU is generated so that min/max shall be checked whenever meaningful (e.g. for debug purposes) then the following strategy should be used:

If `fmiSetReal` *or* `fmiSetInteger` *is called violating the min/max attribute settings of the corresponding variable, the following actions are performed:*

- *On a fixed or tunable parameter* `fmiStatus = fmiDiscard` *is returned.*
- *On an input, the FMU decides what to return (If no computation is possible, it could return* `fmiStatus = fmiDiscard`*, in other situations it may return* `fmiWarning` *or* `fmiError`*, or* `fmiOK`*, if it is uncritical).*

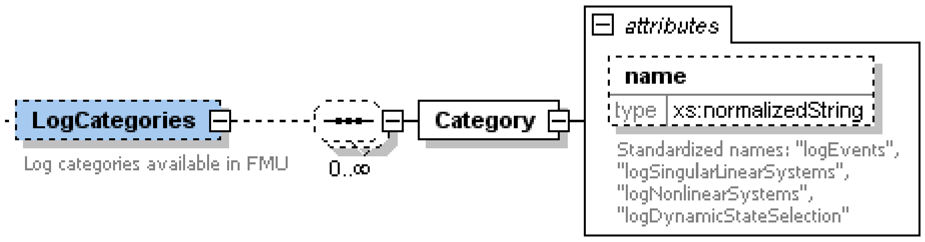
If an FMU defines min/max values for Integer and Enumerations (local and output variables), then the expected behavior of the FMU is that `fmiGetInteger` *returns values in the defined range.*

If an FMU defines min/max values for Reals, then the expected behavior of the FMU is that `fmiGetReal` *returns values at the solution (accepted steps of the integrators) in the defined range with a certain uncertainty related to the tolerances of the numerical algorithms.*

]

2.2.4 Definition of Log Categories (fmiModelDescription.LogCategories)

Element "**`LogCategories`**" of "`fmiModelDescription` is defined as:


LogCategories
Log categories available in FMU
0..∞
Category
attributes
name
type xs:normalizedString
Standardized names: "logEvents", "logSingularLinearSystems", "logNonlinearSystems", "logDynamicStateSelection"

LogCategories defines an unordered set of category strings that can be utilized to define the log output via function "`logger`", see section 2.1.5. A tool is free to use any normalizedString for a category value. There are, however, the following standardized names and these names should be used if a tool supports the corresponding log category:

Category name	*Description*
`logEvents`	Log all events (during initialization and simulation).
`logSingularLinearSystems`	Log the solution of linear systems of equations if the solution is singular (and the tool picked one solution of the infinitely many solutions).
`logNonlinearSystems`	Log the solution of nonlinear systems of equations.
`logDynamicStateSelection`	Log the dynamic selection of states.

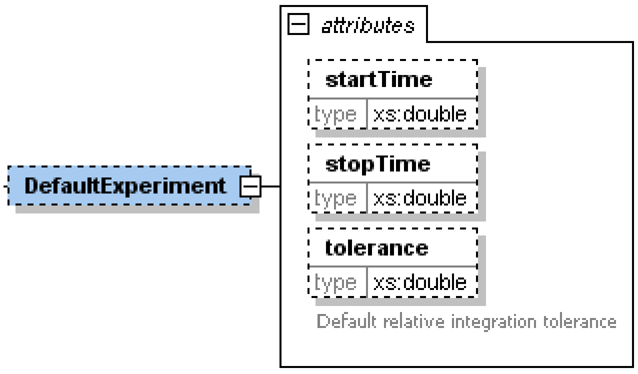
[*This approach to define LogCategories has the following advantages:*

1. *A simulation environment can present the possible log categories in a menu and the user can select the desired one (in the FMI 1.0 approach, there was no easy way for a user to figure out from a given FMU what log categories could be provided). Note, since element LogCategories is optional, an FMU does not need to expose its log categories.*
2. *The log output is drastically reduced, because via* `fmiSetDebugLogging` *exactly the categories are set that shall be logged and therefore the FMU only has to print the messages with the corresponding categories to the* `logger` *function. In FMI 1.0 it was necessary to provide all log output of the FMU to the* `logger` *and then a filter in the* `logger` *could select what to show to the end-user. The approach introduced in FMI 2.0 is therefore much more efficient.*

]

2.2.5 Definition of a Default Experiment (fmiModelDescription.DefaultExperiment)

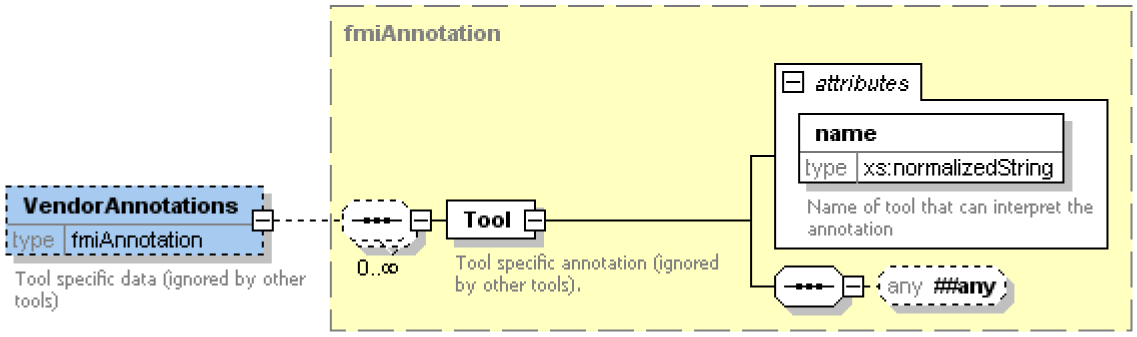
Element "**`DefaultExperiment`**" of `fmiModelDescription` is defined as:



`DefaultExperiment` consists of the optional default start time, stop time and relative tolerance for the first simulation run. A tool may ignore this information. However, it is convenient for a user that `startTime`, `stopTime` and `tolerance` have already a meaningful default value for the model at hand.

2.2.6 Definition of Vendor Annotations (fmiModelDescription.VendorAnnotations)

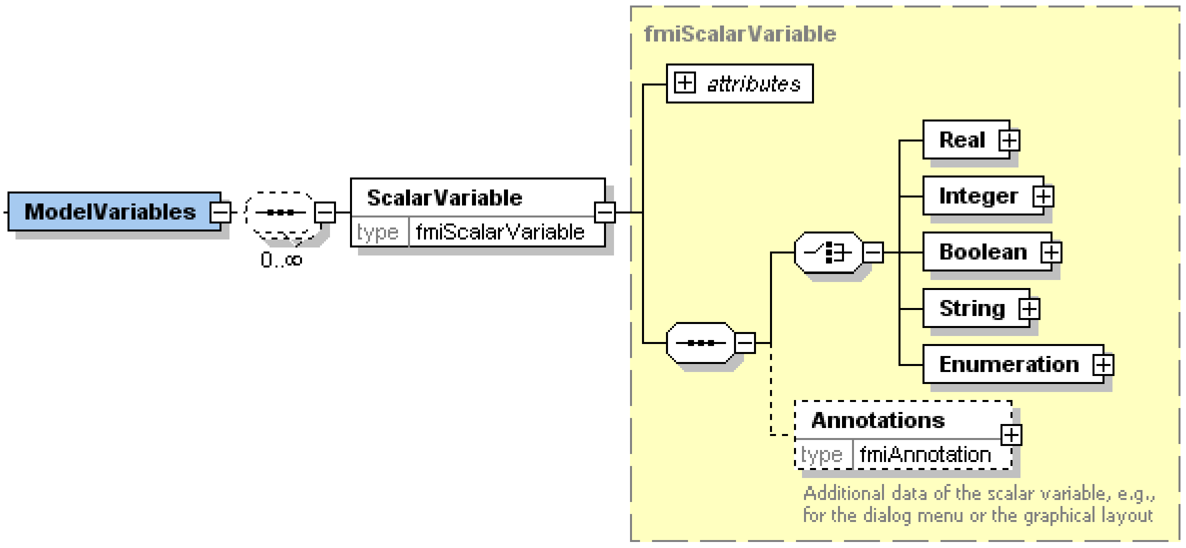
Element “**`VendorAnnotations`**” of `fmiModelDescription` is defined as:


fmiAnnotation
attributes
name
type xs:normalizedString
Name of tool that can interpret the annotation
VendorAnnotations
type fmiAnnotation
Tool specific data (ignored by other tools)
0..∞
Tool
Tool specific annotation (ignored by other tools).
any ##any

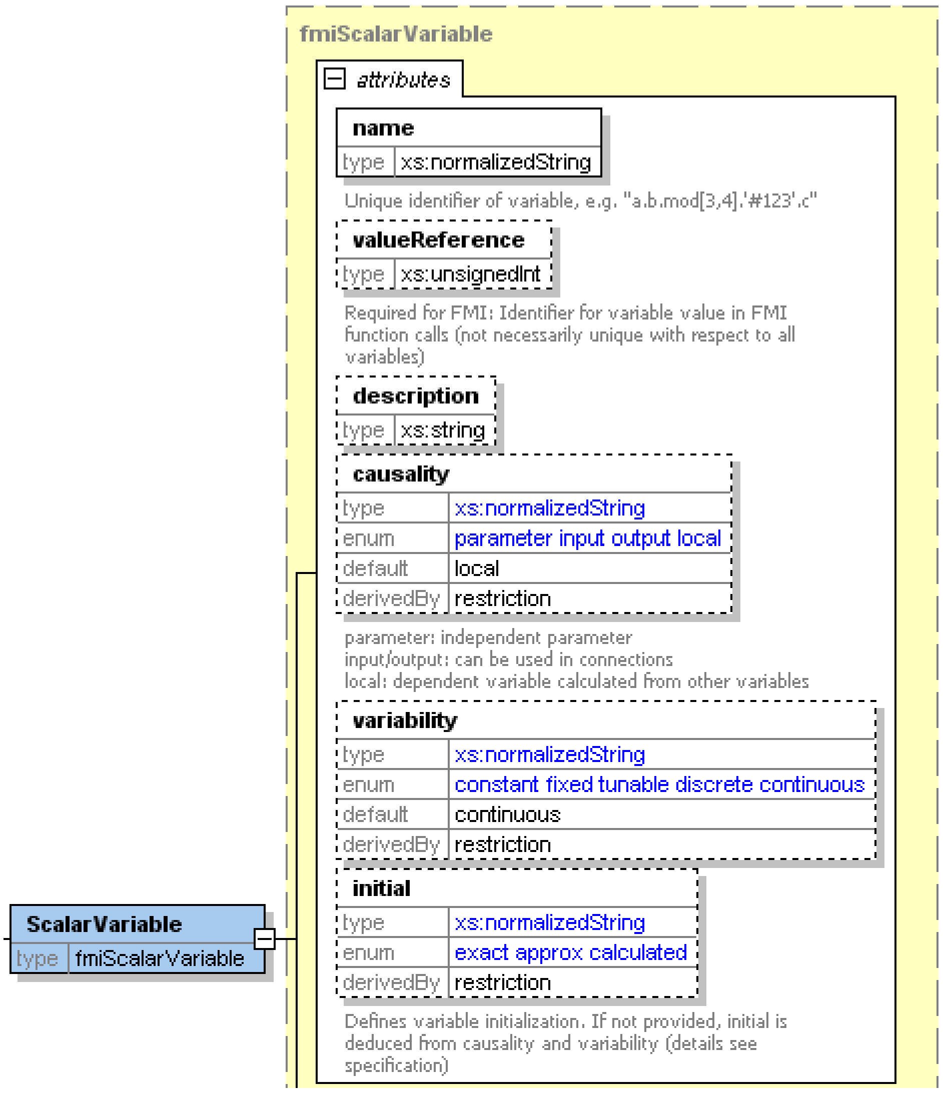
VendorAnnotations consist of an ordered set of annotations that are identified by the name of the tool that can interpret the “`any`” element. The “`any`” element can be an arbitrary XML data structure defined by the tool.

2.2.7 Definition of Model Variables (fmiModelDescription.ModelVariables)

The “**`ModelVariables`**” element of `fmiModelDescription` is the central part of the model description. It provides the static information of all exposed variables and is defined as:



The “`ModelVariables`” element consists of an ordered set of “`ScalarVariable`” elements (see figure above). A “`ScalarVariable`” represents a variable of primitive type, like a real or integer variable. For simplicity, only scalar variables are supported in the schema file in this version and structured entities (like arrays or records) have to be mapped to scalars. The schema definition is present in a separate file “`fmiScalarVariable.xsd`”. The attributes of “`ScalarVariable`” are:


fmiScalarVariable
attributes
name
type xs:normalizedString
Unique identifier of variable, e.g. "a.b.mod[3,4].'#123'.c"
valueReference
type xs:unsignedInt
Required for FMI: Identifier for variable value in FMI function calls (not necessarily unique with respect to all variables)
description
type xs:string
causality
type xs:normalizedString
enum parameter input output local
default local
derivedBy restriction
parameter: independent parameter
input/output: can be used in connections
local: dependent variable calculated from other variables
variability
type xs:normalizedString
enum constant fixed tunable discrete continuous
default continuous
derivedBy restriction
initial
type xs:normalizedString
enum exact approx calculated
derivedBy restriction
Defines variable initialization. If not provided, initial is deduced from causality and variability (details see specification)
ScalarVariable
type fmiScalarVariable

Attribute-Name	*Description*
`name`	The full, unique name of the variable. Every variable is uniquely identified within an FMU instance by this name.
`valueReference`	A handle of the variable to efficiently identify the variable value in the model interface. This handle is a secret of the tool that generated the C functions. It is not required to be unique. The only guarantee is that `valueReference` is sufficient to identify the respective variable value in the call of the C functions. This implies that it is unique for a particular base data type (`Real, Integer/Enumeration, Boolean, String`) with exception of variables that have identical values (such variables are also called “alias” variables). This attribute is “required”. However, since the `fmiScalarVariable` schema definition shall also be used for other (non-FMI) purposes, it is defined as “optional”.
`description`	An optional description string describing the meaning of the variable.
`causality`	Enumeration that defines the causality of the variable. Allowed values of this enumeration: • `"parameter"`: Independent parameter (an independent data value that is constant during the simulation). `variability` must be `fixed` or `tunable`. `initial` must be `exact` or not present (meaning `exact`).

	• `"input"`: The variable value can be provided from another model. `initial` must be `exact` or not present (meaning `exact`). Initially, the value of the variable is set to its `start` value (see below). • `"output"`: The variable value can be used by another model. The algebraic relationship to the inputs is defined via the `inputDependency` attribute of `fmiModelDescription.ModelStructure.Outputs.Output`. • `"`*`local`*`"`: Local variable that is calculated from other variables. It is not allowed to use the variable value in another model. The default is “`local`”.
`variability`	Enumeration that defines the time dependency of the variable, in other words it defines the time instants when a variable can change its value. [*The purpose of this attribute is to define when a result value needs to be inquired and to be stored. For example discrete variables change their values only at event instants and it is therefore only necessary to inquire them with fmiGetXXX and store them at event times*]. Allowed values of this enumeration: • `"constant"`: The value of the variable never changes. • `"fixed"`: The value of the variable is fixed after initialization, in other words after `fmiInitializeXXX` was called the variable value does not change anymore. • `"tunable"`: The value of the variable is constant between externally triggered events due to changing variables with `causality = "parameter"` or `"input"` and `variability = "tunable"`. Whenever a `parameter` or `input` signal with `variability = "tunable"` changes, then an event is triggered externally and the `output` and `local` variables with `variability = "tunable"` must be newly computed. • `"discrete"`: The value of the variable is constant between internal events (= time, state, step events defined implicitly in the FMU). • `"`*`continuous`*`"`: No restrictions on value changes. Only a variable of type = “`Real`” can be “`continuous`”. The default is “`continuous`”. [*Note, the information about continuous states is defined with element* `fmiModelDescription.ModelStructure.Derivatives`]
`initial`	Enumeration that defines how the variable is initialized: • = `"exact"`: The variable is initialized with the `start` value (provided under `Real`, `Integer`, `Boolean`, `String` or `Enumeration`). • = `"approx"`: The variable is an iteration variable of an algebraic loop and the iteration at initialization starts with the `start` value • = `"calculated"`: The variable is calculated from other variables during initialization. It is not allowed to provide a “`start`” value. If `initial` is not present, it is defined by the table below based on `causality` and `variability`. If `initial` = `exact` or `approx`, a `start` value must be provided. If `initial` = `calculated`, it is not allowed to provide a `start` value.

fmiSetXXX can be called on any variable **before initialization** if

• `initial` = `exact` or `approx` [*in order to set the corresponding* `start` *value*].

fmiSetXXX can be called on any variable **after initialization** if

- `variability = tunable`
  [*in order to change the value of the tunable parameter at an event instant*].
- `causality = input`, provided `variability = discrete` or `continuous`
  [*in order to provide new values for inputs*]

If `initial` is not present, its value is defined by the following tables based on the values of `causality` and `variability`. [*Note, "B" means that the variable is a dependent parameter which is computed from independent parameters and/or constants.*]:

			causality			
			parameter	input	output	local
variability	data	constant	--	--	(A)	(A)
		fixed	(A)	(A)	(B)	(B)
		tunable	(A)	(A)	(B)	(B)
	signals	discrete	--	(A)	(C)	(C)
		continuous	--	(A)	(C)	(C)

	initial	
	defaut	possible values
(A)	`exact`	`exact`
(B)	`calculated`	`approx` `calculated`
(C)	`calculated`	`exact`, `approx`, `calculated`

The following combinations of `variability/causality` settings are allowed:

			causality			
			parameter	input	output	local
variability	data	constant	-- (a)	-- (a)	(7)	(12)
		fixed	(1)	(3)	(8)	(13)
		tunable	(2)	(4)	(9)	(14)
	signals	discrete	-- (b)	(5)	(10)	(15)
		continuous	-- (b)	(6)	(11)	(16)

[*Discussion of the combinations that are not allowed:*

	Explanation why this combination is not allowed
(a)	*The combination "constant / parameter" and "constant / input" does not make sense, since parameters and inputs are set from the environment, whereas a constant has always a value.*
(b)	*The combination "discrete / parameter" and "continuous / parameter" does not make sense, since causality = "parameter" defines an independent parameter that does not depend on time (if it is "tunable", the value change is interpreted as a new start of the simulation), whereas "discrete" and "continuous" defines a variable where the value can change during simulation.*

Discussion of the combinations that are allowed:

	Setting	***Example***
(1)	*fixed parameter*	*Non-tunable independent parameter.*
(2)	*tunable parameter*	*Tunable independent parameter (changing such a parameter triggers an external event, and tunable output/local variables might change their values).*
(3)	*fixed input*	*Non-tunable independent parameter from another model.*
(4)	*tunable input*	*Tunable independent parameter from another model (distribution of parameters through model connections).*
(5)	*discrete input*	*Discrete input variable from another model.*

(6)	*continuous input*	*Continuous input variable from another model.*
(7)	*constant output*	*Variable where the value never changes and that can be used in another model,*
(8)	*fixed output*	*Parameter that depends on fixed parameters and can be used in another model (for example there is an equation y = p, where the output is set to a parameter p, that in turn depends on other parameters).*
(9)	*tunable output*	*Parameter that depends on tunable parameters and is computed in the FMU. Can be used in another model.*
(10)	*discrete output*	*Discrete variable that is computed in the FMU. Can be used in another model.*
(11)	*continuous output*	*Continuous variable that is computed in the FMU and can be used in another model.*
(12)	*constant local*	*Variable where the value never changes. Cannot be used in another model.*
(13)	*fixed local*	*Parameter that depends on fixed parameters and is computed in the FMU. Cannot be used in another model.*
(14)	*tunable local*	*Parameter that depends on tunable parameters and is computed in the FMU. Cannot be used in another model.*
(15)	*discrete local*	*Discrete variable that is computed in the FMU and cannot be used in another model.*
(16)	*continuous local*	*Continuous variable that is computed in the FMU and cannot be used in another model.*

How to treat tunable variables:

A parameter p is a variable that does not change its value during simulation, in other words dp/dt = 0. If the parameter "p" is changing, then Dirac impulses are introduced since dp/dt of a discontinuous constant variable "p" is a Dirac impulse. Even if this Dirac impulse would be modeled correctly by the modeling environment, it would introduce vibrations. Furthermore, in many cases the model equations are derived under the assumption of a constant value (like mass or capacity), and the model equations would be different if "p" would be time varying.
Therefore, "tuning a parameter" during simulation does not mean to "change the parameter online" during simulation. Instead, this is a short hand notation for:

1. *Stop the simulation at an event instant*
   (usually, a step event, in other words after a successful integration step).
2. *Change the values of the tunable parameters.*
3. *Compute all parameters that depend on the tunable parameters.*
4. *Newly start the simulation using as initial values the current values of all previous variables and the new values of the parameters.*

With this interpretation, changing parameters online is "clean", as long as these changes appear at an event instant.
]

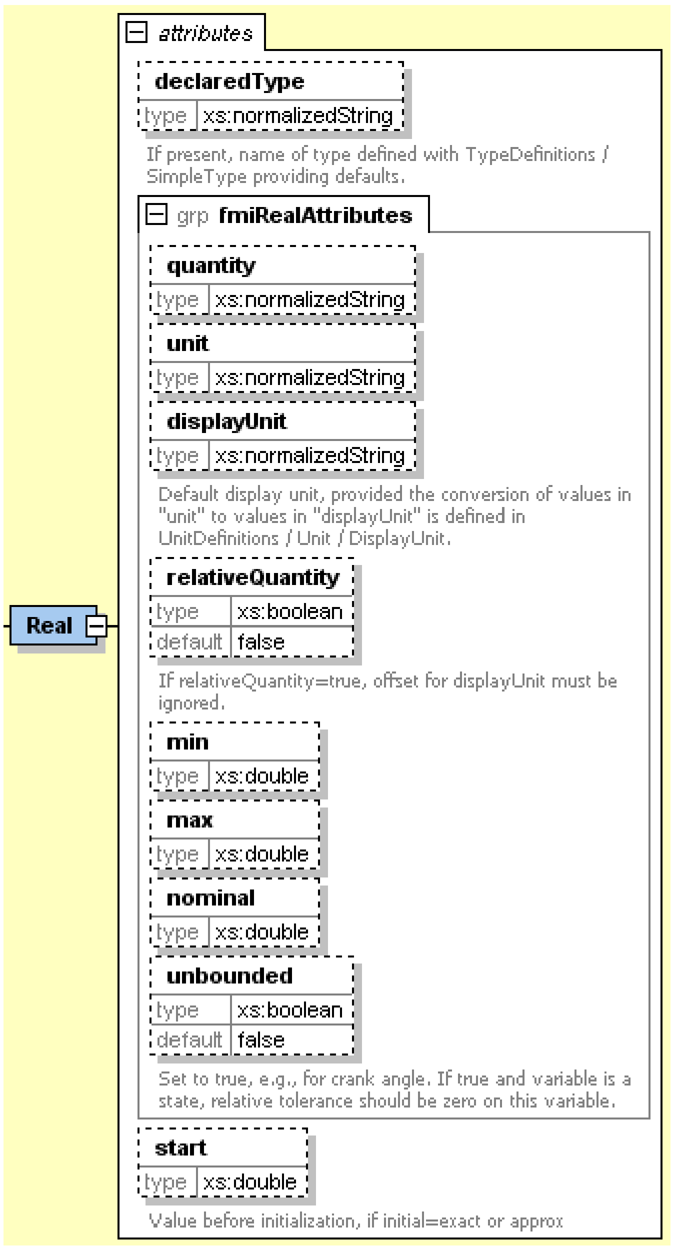
Variables of the same base type (like `fmiReal`) that have identical `valueReference` definitions are called "alias" variables. For "alias" variables several natural restrictions hold:

1. Variables with `causality = "parameter"` or `"input"` cannot be alias variables [*since these variables are "independent" variables and alias means that there is a constraint equation between variables (= the values are the same), and then the variables are no longer "independent"*].
2. At most one variable of the same alias set of variables can have a `start` attribute. [*since "*`start`*" variables are independent initial values.*]

The aliasing of variables only means that the "value" of the variables is always identical. However, aliased variables may have different attributes, like min/max/nominal values or description texts. [*For example if v1, v2 are two alias variables with v1=v2 and v1.max=10 and v2.max=5, then the FMU will trigger an error if either v1 or v2 becomes larger than 5.*]

[*The dependency definition in* `fmiModelDescription.ModelStructure` *is completely unrelated to the alias definition. In particular, the "direct dependency" definition can be a super set of the "real" direct dependency definition, even if the "alias" information shows that this is too conservative. For example if it is stated that the output y1 depends on input u1 and the output y2 depends on input u2, and y1 is an alias to y2, then this definition is fine, although it can be deduced that in reality neither y1 nor y2 depend on any input.*].

Type specific properties are defined in the required choice element, where exactly one of "`Real`", "`Integer`", "`Boolean`", "`String`", "`Enumeration`" must be present in the XML file:



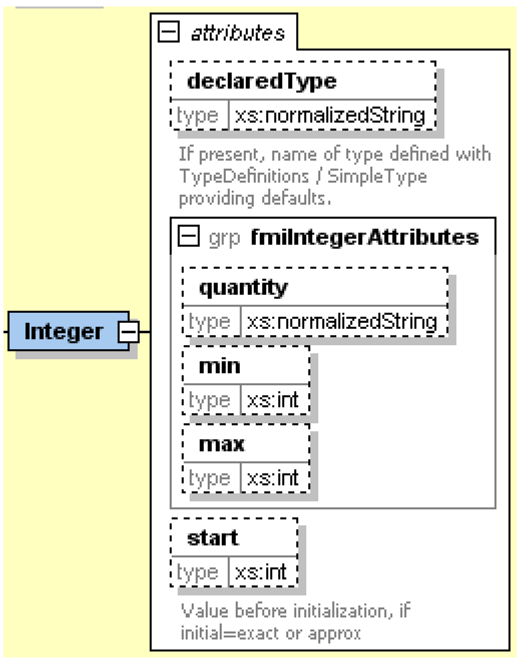
If present, name of type defined with TypeDefinitions / SimpleType providing defaults.

Default display unit, provided the conversion of values in "unit" to values in "displayUnit" is defined in UnitDefinitions / Unit / DisplayUnit.

If relativeQuantity=true, offset for displayUnit must be ignored.

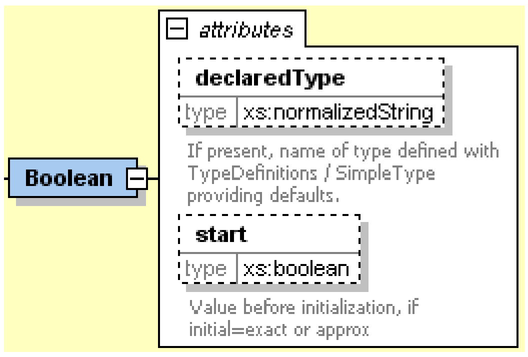
Set to true, e.g., for crank angle. If true and variable is a state, relative tolerance should be zero on this variable.

Value before initialization, if initial=exact or approx



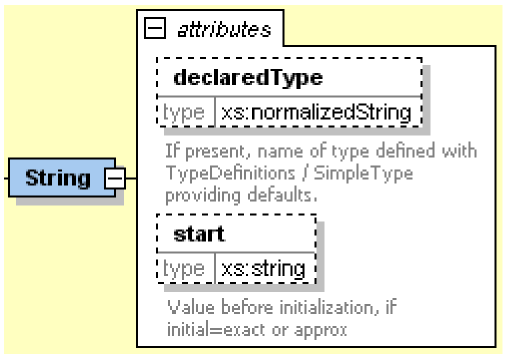
If present, name of type defined with TypeDefinitions / SimpleType providing defaults.

Value before initialization, if initial=exact or approx



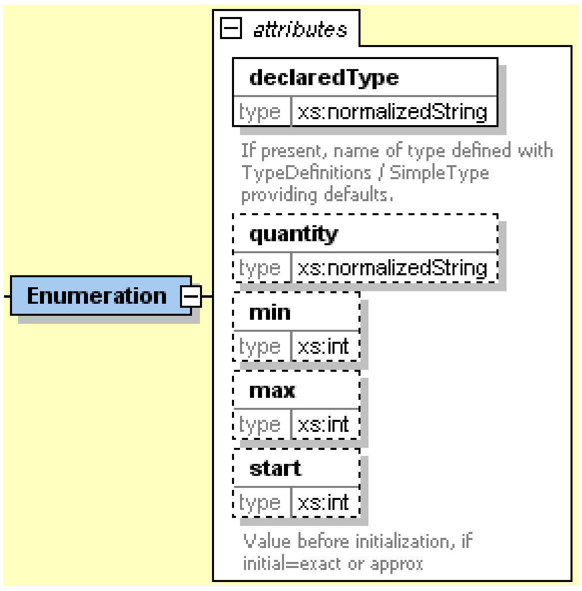
If present, name of type defined with TypeDefinitions / SimpleType providing defaults.

Value before initialization, if initial=exact or approx



If present, name of type defined with TypeDefinitions / SimpleType providing defaults.

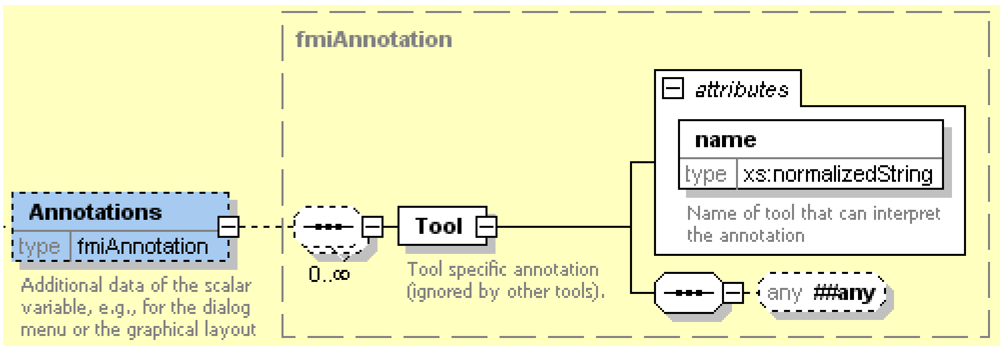
Value before initialization, if initial=exact or approx


attributes
declaredType
type xs:normalizedString
If present, name of type defined with TypeDefinitions / SimpleType providing defaults.
quantity
type xs:normalizedString
Enumeration
min
type xs:int
max
type xs:int
start
type xs:int
Value before initialization, if initial=exact or approx

The attributes are defined in section 2.2.3 (“`fmiType`”), except:

Attribute-Name	*Description*
`declaredType`	If present, name of type defined with `TypeDefinitions` / `SimpleType` (`fmiType`). The value defined in the corresponding `TypeDefinition` (see section 2.2.3) is used as default. [*If, for example “`min`” is present both in `Real` (of `TypeDefinition`) and in “`Real`” (of `ScalarVariable`), then the “`min`” of `ScalarVariable` is actually used.*] For `Real, Integer, Boolean, String,` this attribute is optional. For `Enumeration` it is required, because the `Enumeration` items are defined in `TypeDefinitions` / `SimpleType`.
`start`	Initial or guess value of variable. **This value is also stored in the C functions.** The interpretation of `start` is defined by `ScalarVariable` / `initial`. A different start value can be provided with an `fmiSetXXX` function before `fmiInitialize` is called (but not for “`constant`” variables). Variables with `causality = "parameter"` or `"input"`, as well as variables with `variability = "constant"`, must have a `"start"` value. • If `causality = "parameter"`, the `start`-value is the value of this independent parameter. • If `causality = "input"`, the `start` value is used by the model as value of the input, if the input is not set by the environment. • If `variability = "constant"`, the `start` value is the value of the constant. • If `causality = "output"`or or `"local"` then the `start` value is either an “initial” or a “guess” value, depending on the setting of attribute `"initial"`.
`min / max`	The optional attributes “`min`” and “`max`” in element “`Enumeration`” restrict the allowed values of the enumeration. The min/max definitions are an information from the FMU to the environment defining the region in which the FMU is designed to operate, see also comment in section 2.2.3. [*If, for example an `Enumeration` is defined with “name1 = -4”, “name2 = 1”, “name3 = 5”, “name4 = 11” and min=-2, max = 5, then only “name2” and “name3” are allowed*].

With element "`Annotations`" additional, tool specific data can be defined:


fmiAnnotation
attributes
name
type xs:normalizedString
Name of tool that can interpret the annotation
Annotations
type fmiAnnotation
Additional data of the scalar variable, e.g., for the dialog menu or the graphical layout
0..∞
Tool
Tool specific annotation (ignored by other tools).
any ##any

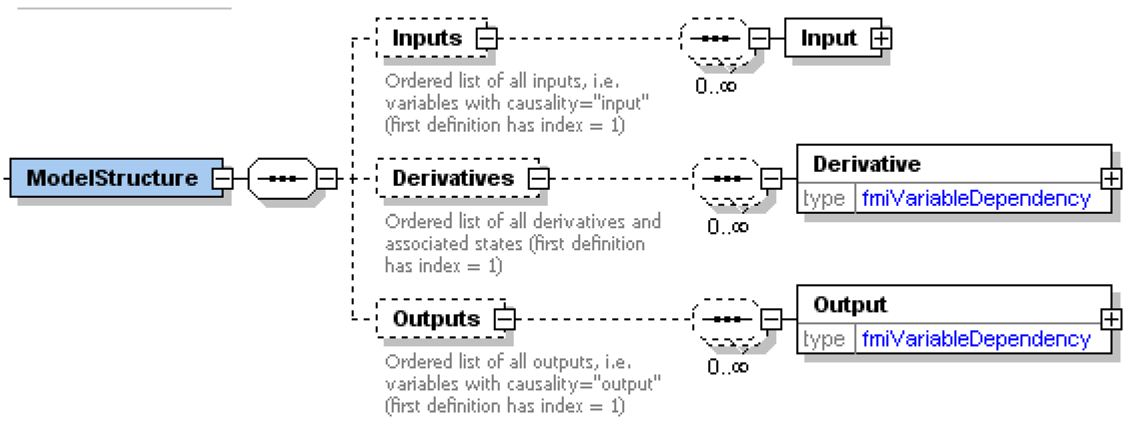
With `Tool.name` the name of the tool is defined that can interpret the "`any`" element. The "`any`" element can be an arbitrary XML data structure defined by the tool. [*Typically, additional data is defined here how to build up the menu for the variable, including the graphical layout and enabling/disabling an input field based on the values of other parameters.*]

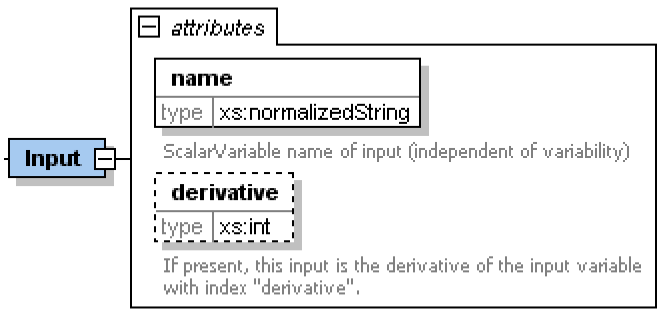
2.2.8 Definition of the Model Structure (fmiModelDescrption.ModelStructure)

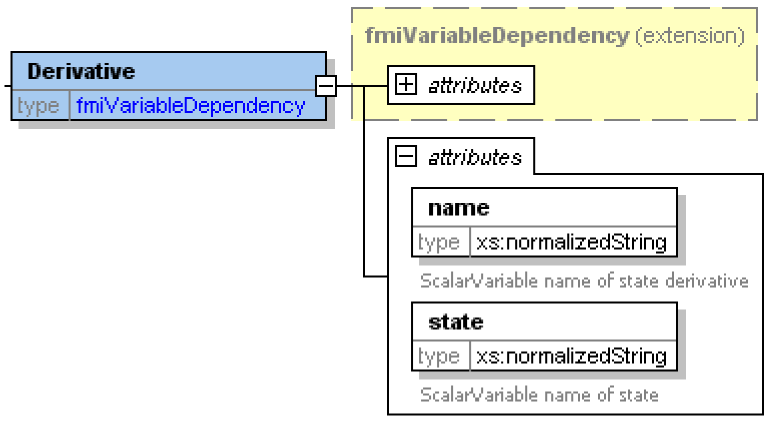
The structure of the model is defined in element "**`ModelStructure`**" within "`fmiModelDescription`". The required part defines an ordering of the inputs, outputs, and derivatives, and the association of every (continuous) state with its derivative. [*Therefore, when linearizing an FMU, every tool will use the same ordering for the inputs, outputs, states, and derivatives for the linearized model.*]

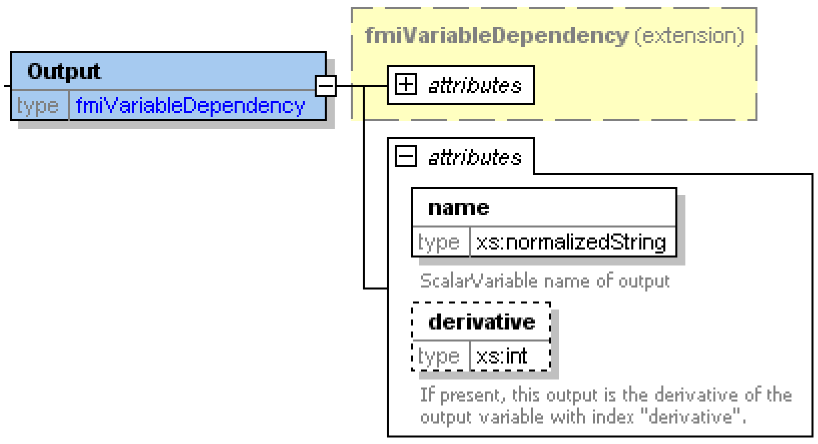
The optional part defines in which way derivatives and outputs depend on inputs and states. [*A simulation environment can utilize this information to improve the efficiency, e.g., when connecting FMUs together, or when computing the partial derivative of the derivatives with respect to the states in the simulation engine.*]

The required part of `ModelStructure` has the following definition (attribute `derivative` is not required, but gives optional information):


ModelStructure
Inputs
Ordered list of all inputs, i.e. variables with causality="input" (first definition has index = 1)
0..∞
Input
Derivatives
Ordered list of all derivatives and associated states (first definition has index = 1)
0..∞
Derivative
type fmiVariableDependency
Outputs
Ordered list of all outputs, i.e. variables with causality="output" (first definition has index = 1)
0..∞
Output
type fmiVariableDependency


Input
attributes
name
type xs:normalizedString
ScalarVariable name of input (independent of variability)
derivative
type xs:int
If present, this input is the derivative of the input variable with index "derivative".


Derivative
type fmiVariableDependency
fmiVariableDependency (extension)
attributes
attributes
name
type xs:normalizedString
ScalarVariable name of state derivative
state
type xs:normalizedString
ScalarVariable name of state


Output
type fmiVariableDependency
fmiVariableDependency (extension)
attributes
attributes
name
type xs:normalizedString
ScalarVariable name of output
derivative
type xs:int
If present, this output is the derivative of the output variable with index "derivative".

ModelStructure consists of the following elements (see also figures above):

Element-Name	*Description*
`Inputs`	Defines an ordered list of all inputs, in other words a list of `ScalarVariable` names where every `ScalarVariable` must have `causality = "input"` (and **every variable with `causality="input"` must be listed here**). The first definition has index=1, the second, index=2, etc. [*Note, all input variables are listed here, especially discrete and continuous inputs*]. If the continuous input u_k has attribute "`derivative=i`" then u_k is the time derivative of the continuous input u_i: $$u_k = \frac{du_i}{dt}$$
`Derivatives`	Defines an ordered list of the state derivative and associated state vector, in other words a list of elements, where every element has a reference to the `ScalarVariable` `name` of the state derivative and the `ScalarVariable` `name` of its associated state. The first definition has index=1, the second index=2, etc. *[Note, only continuous Real*

	variables are listed here. If a state or a derivative of a state shall not be exposed from the FMU, or if states are not statically associated with a variable (due to dynamic state selection), then dummy ScalarVariables have to be introduced, for example x[4], or xDynamicStateSet2[5].] For Co-Simulation, element "`Derivatives`" is ignored if all capability flags `providesPartialDerivativesOf_DerivativeFunction_wrt_States`, `providesPartialDerivativesOf_DerivativeFunction_wrt_Inputs`, `providesPartialDerivativesOf_OutputFunction_wrt_States`. have a value of `false`, in other words cannot be computed [*which is the default. If an FMU supports both ModelExchange and CoSimulation, then the "Derivatives" element might be present, since it is needed for ModelExchange. If the above flags are set to false for the CoSimulation case, then the "Derivatives" element is ignored for CoSimulation*].
`Outputs`	Defines the ordered list of all outputs, in other words a list of `ScalarVariable` `names` where every `ScalarVariable` must have `causality = "output"` (and **every variable with `causality="output"` must be listed here**). The first definition has index=1, the second, index=2, etc. [*Note, all output variables are listed here, especially discrete and continuous outputs*]. If the continuous output y_k has attribute "`derivative=i`" then y_k is the time derivative of the continuous output y_i: $$y_k = \frac{dy_i}{dt}$$

The "`Derivatives`" and "`Outputs`" element optionally define their dependency on (continuous) states and on (all) inputs and also provide more information if the independent variable enters the unknown linearly with a discrete-time factor. [*If present, this information can be used to compute the sparse Jacobian for the integrator, or to determine the direct dependency of outputs from inputs and use this information when connecting FMUs.*]. In general, it is assumed that the following equation structure is present (see also section 3.1):

$$\begin{aligned}\frac{d\mathbf{x}}{dt}(t) &= \mathbf{f}(\mathbf{x}(t),\mathbf{u}(t),t,\mathbf{m}(t_e)) \\ \mathbf{y}(t) &= \mathbf{g}(\mathbf{x}(t),\mathbf{u}(t),t,\mathbf{m}(t_e))\end{aligned}$$

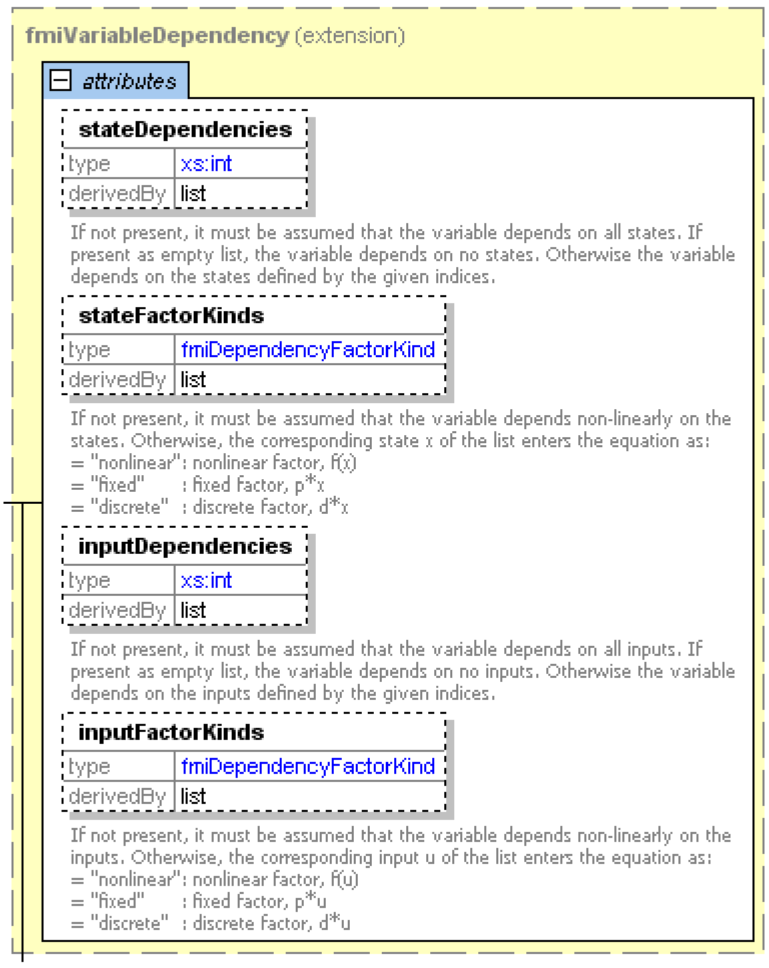
where **x** are (continuous) states, **u** are inputs, **y** are outputs, **m** are discrete-time variables, and an element of **f**(..) or **g**(..) may depend on only a subset of the states and/or the inputs. Furthermore, some states and some *continuous-time* inputs may enter the equation linearly with a factor that changes its value not during continuous integration (but at event instants):

$$\begin{aligned}\frac{dx_i}{dt}(t) &= f_i(\mathbf{x}(t),\mathbf{u}(t),t,\mathbf{m}(t_e)) \\ &= f_{i0}(\mathbf{x}_{nl}(t),\mathbf{u}_{nl}(t),t,\mathbf{m}(t_e)) + \sum_k f_{ik}^{disc,x}(\mathbf{u}_{disc}(t_e),t_e,\mathbf{m}(t_e))\cdot x_k + \sum_k f_{ik}^{disc,u}(\mathbf{u}_{disc}(t_e),t_e,\mathbf{m}(t_e))\cdot u_{cont,k} \\ y_j(t) &= g_j(\mathbf{x}(t),\mathbf{u}(t),t,\mathbf{m}(t_e)) \\ &= g_{j0}(\mathbf{x}_{nl}(t),\mathbf{u}_{nl}(t),t,\mathbf{m}(t_e)) + \sum_k g_{jk}^{disc,x}(\mathbf{u}_{disc}(t_e),t_e,\mathbf{m}(t_e))\cdot x_k + \sum_k g_{jk}^{disc,u}(\mathbf{u}_{disc}(t_e),t_e,\mathbf{m}(t_e))\cdot u_{cont,k}\end{aligned} \tag{1}$$

where

- f_{i0}, g_{j0} are functions of time
- $f_{ik}^{disc,x}, f_{ik}^{disc,u}, g_{ik}^{disc,x}, g_{ik}^{disc,u}$ are functions of event time t_e, in other words, their return values change only at event times.
- $\mathbf{x}_{nl}, \mathbf{u}_{nl}$ are states and inputs that enter the equations in a non-linear or unspecified way.
- x_k is a state that enters the equation as linear factor of a product where one term is x_k and the other term is a function that changes its value only at event times.
- $u_{cont,k}$ is a continuous-time input (variability = “continuous”) that enters the equation as linear factor of a product where one term is $u_{cont,k}$ and the other term is a function that changes its value only at event times.

These relationships are defined in the `Derivative` and `Output` elements with optional list attributes using the `fmiVariableDependency` schema definition:



fmiVariableDependency (extension)

attributes

stateDependencies	
type	xs:int
derivedBy	list

If not present, it must be assumed that the variable depends on all states. If present as empty list, the variable depends on no states. Otherwise the variable depends on the states defined by the given indices.

stateFactorKinds	
type	fmiDependencyFactorKind
derivedBy	list

If not present, it must be assumed that the variable depends non-linearly on the states. Otherwise, the corresponding state x of the list enters the equation as:
= "nonlinear": nonlinear factor, f(x)
= "fixed"   : fixed factor, p*x
= "discrete" : discrete factor, d*x

inputDependencies	
type	xs:int
derivedBy	list

If not present, it must be assumed that the variable depends on all inputs. If present as empty list, the variable depends on no inputs. Otherwise the variable depends on the inputs defined by the given indices.

inputFactorKinds	
type	fmiDependencyFactorKind
derivedBy	list

If not present, it must be assumed that the variable depends non-linearly on the inputs. Otherwise, the corresponding input u of the list enters the equation as:
= "nonlinear": nonlinear factor, f(u)
= "fixed"   : fixed factor, p*u
= "discrete" : discrete factor, d*u

The optional `stateDependencies` list has integer elements that refer to the index of the corresponding ordered set of derivatives/states (element “`Derivative`”). [*For example if the variable is a function of the second and fourth state, then the list is defined as “2 4”.*].

For Co-Simulation, attribute “`stateDependencies`” is ignored if the capability flags
`providesPartialDerivativesOf_DerivativeFunction_wrt_States`,
`providesPartialDerivativesOf_OutputFunction_wrt_States`.
have a value of `false`, in other words the respective partial derivatives cannot be computed.

The optional `inputDependencies` lists have integer elements that refer to the index of the corresponding ordered set of inputs (element “`Input`”). [*For example if the variable is a function of the second and fifth input, then the list is defined as “2 5”.*]

The “`stateFactorKinds`” and “`inputFactorKinds`” lists define optionally in which way the variable depends on the states/inputs, see also equation (1) above:

- List is not present: It must be assumed that the variable depends non-linearly on all states and/or inputs defined with the `stateDependencies` and `inputDependencies` lists.
- List is present and a list element is `"nonlinear"`: The variable depends non-linearly on the respective state or input.
- List is present and a list element is `"fixed"` or `"discrete"`: The variable depends on the respective state or continuous-time input (variability = “continuous”) that enters the equation as linear factor of a product where one term is the respective state or input and the other term is a function that depends on the value of the list element (in other words, the relationships with the terms $f_{ik}^{disc,x}, f_{ik}^{disc,u}, g_{ik}^{disc,x}, g_{ik}^{disc,u}$ in equation (1) are defined. If the list element is `"fixed"` then the term does not change its return value after initialization. If the list element is `"discrete"` then the terrm changes its value only at event instants.

[*Example:*

The model structure of an FMU with the equations:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} f_1(x_2) \\ f_2(x_1) + 3 \cdot p^2 \cdot x_2 + 2 \cdot u_1 + 3 \cdot u_3 \\ f_3(x_1, x_3, u_1, u_2, u_3) \end{bmatrix}$$
$$y = g_1(x_2, x_3)$$

where u_1 is a continuous-time input (variability=”continuous”), u_2 is any type of input, u_3 is a discrete-time input (variability=”discrete”), and p is a fixed parameter (variability=”fixed”), can be defined as:

```
<ModelStructure>
   <Inputs>
      <Input name="u1"/>
      <Input name="u2"/>
      <Input name="u3"/>
   </Inputs>

   <Derivatives>
     <Derivative name="der(x1)" state="x1" stateDependencies="2"
                                           inputDependencies="" />
     <Derivative name="der(x2)" state="x2" stateDependencies="1 2"
                                           stateFactorTypes ="nonlinear fixed"
                                           inputDependencies="1 3"
                                           inputFactorTypes ="fixed fixed" />
     <Derivative name="der(x3)" state="x3" stateDependencies="1 3" />
   </Derivatives>

   <Outputs>
     <Output name="y" stateDependencies="2 3" inputDependencies="" />
   </Outputs>
</ModelStructure>
```

]

2.2.9 Variable Naming Conventions (fmiModelDescription.variableNamingConvention)

With attribute "variableNamingConvention" of element "fmiModelDescription", the convention is defined how the ScalarVariable.names have been constructed. If this information is known, the environment may be able to represent the names in a better way (for example as tree and not as a linear list).

In the following definitions, the EBNF is used:

=	production rule
[ ]	optional
{ }	repeat zero or more times
|	or

The following conventions for scalar names are defined:

`variableNamingConvention = "flat"`

```
name = any member of the source character set // no hierarchy
```

The names are an ordered set that might be represented in a drop down menu as a list of strings.

`variableNamingConvention = "structured"`

Structured names are hierarchical using "." as a separator between hierarchies. A name consists of "_", letters and digits or may consist of any characters enclosed in single apostrophes. A name may identify an array element on every hierarchical level using "[...]" to identify the respective array index. A derivative of a variable is defined with "`der(name)`" for the first time derivative and "`der(name,N)`" for the N-th derivative. Examples:

```
vehicle.engine.speed
resistor12.u
v_min
robot.axis.'motor #234'
der(pipe[3,4].T[14],2)    // second time derivative of pipe[3,4].T[14]
```

The precise syntax is:

```
name            = identifier | "der(" identifier ["," unsignedInteger ] ")"
identifier      = B-name [ arrayIndices  ] {"." B-name [ arrayIndices ] }
B-name          = nondigit { digit | nondigit } | Q-name
nondigit        = "_" | letters "a" to "z" | letters "A" to "Z"
digit           = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Q-name          = "'" ( Q-char | escape ) { Q-char | escape } "'"
Q-char          = any member of the source character set except
                  single-quote "'", and backslash "\"
escape          = "\'" | "\"" | "\?" | "\\" | "\a" | "\b" |
                  "\f" | "\n" | "\r" | "\t" | "\v"
arrayIndices    = "[" unsignedInteger {"," unsignedInteger} "]"
unsignedInteger = digit { digit }
```

The tree of names is mapped to an ordered list of ScalarVariable.name's in depth-first order. Example:

```
vehicle
  transmission
    ratio
    outputSpeed
  engine
    inputSpeed
    temperature
```

is mapped to the following list of ScalarVariable.name's:

```
vehicle.transmission.ratio
vehicle.transmission.outputSpeed
vehicle.engine.inputSpeed
vehicle.engine.temperature
```

All array elements are given in a consecutive sequence of `ScalarVariables`. For example the vector "centerOfMass" in body "arm1" is mapped to the following ScalarVariables:

```
robot.arm1.centerOfMass[1]
robot.arm1.centerOfMass[2]
robot.arm2.centerOfMass[3]
```

It might be that not all elements of an array are present. If they are present, they are given in consecutive order in the XML file.

2.3 FMU Distribution

An FMU description consists of several files. An FMU implementation may be distributed in source code and/or in binary format. All relevant files are stored in a zip file with a pre-defined structure. The implementation must either implement all the functions of FMI for Model Exchange or all the functions of FMI for Co-Simulation or both. The extension of the zip file must be "**.fmu**", for example "HybridVehicle.fmu". The compression method used for the zip file must be "deflate" [*(most free tools, for example zlib, offer only the common compression method "deflate")*].

Every FMU is distributed with its own zip file. This zip file has the following structure:

```
// Structure of zip file of an FMU
modelDescription.XML          // Description of FMU (required file)
model.png                     // Optional image file of FMU icon
documentation                 // Optional directory containing the FMU documentation
   _main.html                 // Entry point of the documentation
   <other documentation files>
sources                       // Optional directory containing all C sources
   // all needed C sources and C header files to compile and link the FMU
   // with exception of: fmiTypesPlatform.h , fmiFunctionTypes.h and fmiFunctions.h
binaries                      // Optional directory containing the binaries
   win32                      // Optional binaries for 32-bit Windows
      <modelIdentifier>.dll   // DLL of the FMI implementation
                              // (build with option "MT" to include run-time environment)
      <other DLLs>            // The DLL can include other DLLs
      // Optional object Libraries for a particular compiler
      VisualStudio8         // Binaries for 32-bit Windows generated with
                            // Microsoft Visual Studio 8 (2005)
         <modelIdentifier>.lib   // Binary libraries
      gcc3.1                // Binaries for gcc 3.1.
        ...
   win64    // Optional binaries for 64-bit Windows
      ...
   linux32  // Optional binaries for 32-bit Linux
      <modelIdentifier>.so  // Shared library of the FMI implementation
      ...
   linux64  // Optional binaries for 64-bit Linux
      ...
resources  // Optional resources needed by the FMU
   < data in FMU specific files which will be read during initialization;
     also more folders can be added under resources (tool/model specific).
     In order for the FMU to access these resource files, the resource directory
```

```
      must be available in unzipped form and the absolute path to this directory
      must be reported via argument "fmuResourceLocation" via fmiInstantiateXXX.
    >
```

The FMU must be distributed with at least one implementation, in other words either sources or one of the binaries for a particular machine. It is also possible to provide the sources and binaries for different target machines together in one zip file. The following names are standardized: For Windows: "win32", "win64"- For Linux: "linux32", "linux64". For Mac: "darwin32", "darwin64" Futhermore, also the names "VisualStudioX" and "gccX" are standardize and define the compiler with which the binary has been generated [*, for example VisualStudio8*]. Further names can be introduced by vendors. Dynamic link libraries must include all referenced resources that are not available on a standard target machine [*for example DLLs on Windows machines must be built with option "MT" to include the run-time environment of VisualStudio in the DLL, and not use the option "MD" where this is not the case*]. When compiling a shared object on Linux, RPATH="$ORIGIN" has to be set when generating the shared object in order that shared objects used from it, can be dynamically loaded.

Typical scenarios are to provide binaries only for one machine type (for example on the machine where the target simulator is running and for which licenses of run-time libraries are available) or to provide only sources (for example for translation and download for a particular micro-processor). If run-time libraries cannot be shipped due to licensing, special handling is needed, for example by providing the run-time libraries at appropriate places by the receiver.

FMI provides the means for two kinds of implementation: `needsExecutionTool=true` and `needsExecutionTool=false`. In the first case a tool specific wrapper DLL/SharedObject has to be provided as the binary, in the second a compiled or source code version of the model with its solver is stored (see section 4.3.1 for details).

In an FMU both a version for ModelExchange and for CoSimulation might be present. If in both cases the executable part is provided as DLL/SharedObject, then different library names must be present that are defined in the `modelIndentifier` attribute of elements "`fmiModelDescription.ModelExchange`" and "`fmiModelDescription.CoSimulation`":

```
 [Example:
   binaries
      win32
         MyModel_ModelExchange.dll    // ModelExchange.modelIdentifier =
                                      //    "MyModel_ModelExchange"
         MyModel_CoSimulation.dll     // CoSimulation.modelIdentifier =
                                      //    "MyModel_CoSimulation"
 ]
```

The usual distribution of an FMU will be with DLLs/SharedObjects because then further automatic processing [*(for example importing into another tool)*] is possible.

If run-time libraries are needed by the FMU that have to be present on the target machine, then automatic processing is likely impossible. The requirements and the expected processing should be documented in the "`documentation`" directory in this case.

A source-based distribution will usually require manual interaction in order that it can be utilized. The intention is to support platforms that are not known in advanced (such as HIL-platforms or micro-controllers). Typically, in such a case the complete source code in ANSI-C is provided (for example one C source file that includes all other needed C files with the "#include" directive). An exporting tool should give documentation how to build an executable, either via a documentation file and/or a template makefile for a particular platform, from which a user can construct the makefile for his/her target platform. This documentation should be stored in the "`documentation`" directory, possibly with a link to the template makefile (stored in the "`sources`" directory). [*As template makefile, CMake (www.cmake.org), a cross-platform, open-source build system might be used.*]

In directory "`resources`", additional data can be provided in FMU specific formats, typically for tables and maps used in the FMU. This data must be read into the model at latest during initialization ("`fmiInitializeModel, fmiInitializeSlave`"). The actual file names in the zip file to access the data files can either be hard-coded in the generated FMU functions, or the file names can be provided as string parameters via the "`fmiSetString`" function. [*Note, the absolute file name of the resource directory is provided by the initialization functions*]. In the case of a co-simulation implementation of `needsExecutionTool=true` type, the "resources" directory can contain the model file in the tool specific file format.

[*Note, the header files `fmiTypesPlatform.h` and `fmiFunctionTypes.h/fmiFunctions.h` are not included in the FMU due to the following reasons:*

- *`fmiTypesPlatform.h` makes no sense in the "**`sources`**" directory, because if sources are provided, then the target simulator defines this header file and not the FMU.*
  *This header file is not included in the "**`binaries`**" directory, because it is implicitly defined by the platform directory (for example win32 for 32-bit machine or linux64 for 64-bit machine). Furthermore, the version that was used to construct the FMU can also be inquired via function `fmiGetTypesPlatform()`.*
- *`fmiFunctionTypes.h/fmiFunctions.h` are not needed in the "**`sources`**" directory, because they are implicitly defined by attribute `fmiVersion` in file **`modelDescription.XML`**. Furthermore, in order that the C-compiler can check for consistent function arguments, the header file from the target simulator should be used when compiling the C sources. It would therefore be counter productive (unsafe), if this header file would be present.*
  *These header files are not included in the "**`binaries`**" directory, since they are already utilized to build the target simulator executable. Via attribute `fmiVersion` in file **`modelDescription.XML`** or via function call `fmiGetVersion()` the version number of the header file used to construct the FMU can be deduced.*

]

2.4 Hierarchical FMUs

An FMU may use other FMUs which may use other FMUs. So an FMU may consist of a hierarchy of FMUs (also called external FMUs). All variables in an external FMU that shall be visible and/or accessible from the environment need to be "exposed", in other words in the root-level FMU a corresponding variable needs to be defined and in the generated code this variable must be assigned to the corresponding variable of the external FMU. As a result, only variables from the top most FMU are visible/accessible from the environment where the FMU is called. Note, in case of FMI for model exchange, continuous states of an external FMU must always be exposed. The hierarchical FMU structure is not exposed in the FMU distribution, so in the model zip file only one FMU is contained.

3. FMI for Model Exchange

This chapter contains the interface description to access the equations of a dynamic system from a C program. A schematic view of a model in "FMI for Model Exchange" format is shown in the next figure:

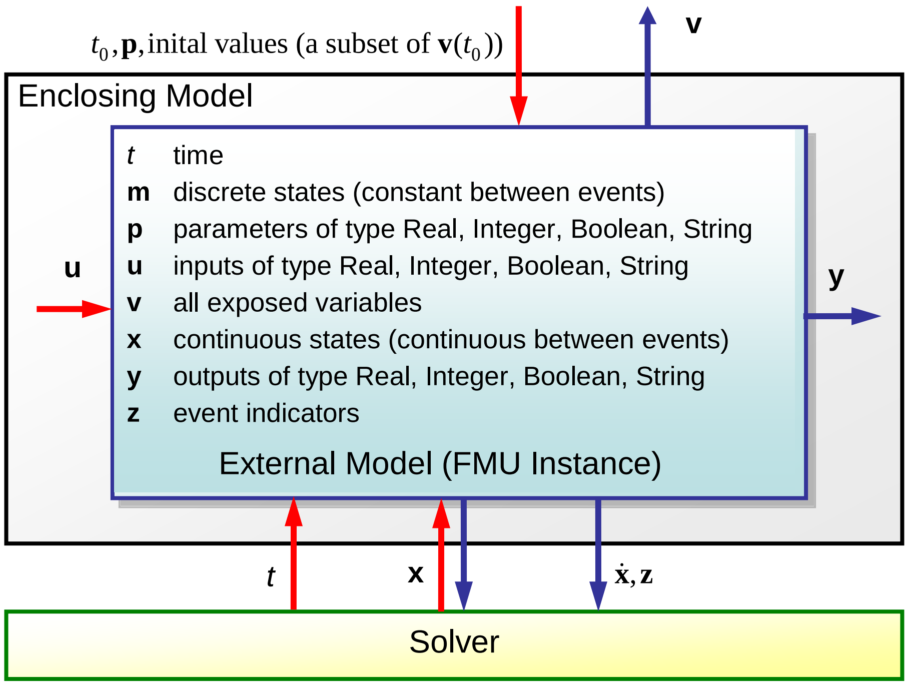

$t_0, \mathbf{p}$, inital values (a subset of $\mathbf{v}(t_0)$)
v
Enclosing Model
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
External Model (FMU Instance)
u
y
t
x
$\dot{\mathbf{x}}, \mathbf{z}$
Solver

Figure 2: Data flow between the environment and an FMU (for Model Exchange).
Blue arrows: Information provided by the FMU.
Red arrows : Information provided to the FMU.

3.1 Mathematical Description

The goal of the Model Exchange interface is to numerically solve a system of differential, algebraic and discrete equations. In this version of the interface, ordinary differential equations in state space form with events are handled (abbreviated as "hybrid ODE"). Algebraic equation systems might be contained inside the FMU.

This type of system is described as a piecewise continuous system. Discontinuities can occur at time instants t_0, t_1, …, t_n, where $t_i < t_{i+1}$. These time instants are called "events". Events can be known before hand (= time event), or are defined implicitly (= state and step events).

The "state" of a hybrid ODE is represented by a continuous state $\mathbf{x}(t)$ and by a time-discrete state $\mathbf{m}(t)$ that have the following properties:

- $\mathbf{x}(t)$ is a vector of real numbers (= time-continuous states) and is a continuous function of time inside each interval $t_i \le t < t_{i+1}$, where $t_i = \lim_{\varepsilon \to 0}(t_i + \varepsilon)$, in other words the right limit to t_i (note, $\mathbf{x}(t)$ is continuous between the right limit to t_i and the left limit to t_{i+1} respectively).
- $\mathbf{m}(t)$ is a set of real, integer, logical, and string variables (= time-discrete states) that is constant inside each interval $t_i \le t < t_{i+1}$. In other words, $\mathbf{m}(t)$ changes value only at events. This means, $\mathbf{m}(t) = \mathbf{m}(t_i)$, for $t_i \le t < t_{i+1}$.

At every event instant t_i, variables might be discontinuous and therefore have two values at this time instant, the "left" and the "right" limit. $\mathbf{x}(t_i)$, $\mathbf{m}(t_i)$ are always defined to be the right limit at t_i, whereas

$\mathbf{x}^{-}(t_i)$, $\mathbf{m}^{-}(t_i)$ are defined to be the "left" limit at t_i. [Example: $\mathbf{m}^{-}(t_i) = \mathbf{m}(t_{i-1})$.] In the following figure, the two variable types are visualized:

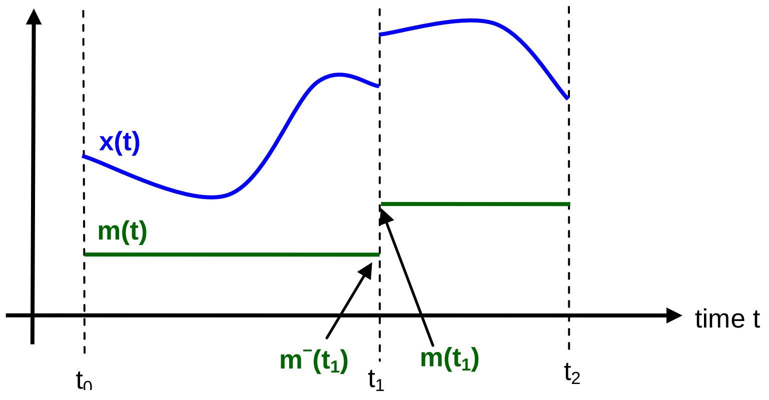


Figure 3: Piecewise-continuous states of an FMU: time-continuous (x) and time-discrete (m).

An <u>event instant</u> t_i is defined by one of the following conditions that give the smallest time instant:

1. At a predefined time instant $t_i = T_{next}(t_{i-1})$ that was defined at the previous event instant t_{i-1} either by the FMU, or by the environment of the FMU due to a discontinuous change of an input signal[5] u_j at t_i. Such an event is called <u>time event</u>.
2. At a time instant, where an <u>event indicator</u> $z_j(t)$ changes its domain from $z_j > 0$ to $z_j \le 0$ or vice versa (see Figure 4 below). More precisely: An event $t = t_i$ occurs at the smallest time instant "min t" with $t > t_{i-1}$ where "$(z_j(t) > 0) \neq (z_j(t_{i-1}) > 0)$". Such an event is called <u>state event</u>[6]. All event indicators are piecewise continuous and are collected together in one vector of real numbers $\mathbf{z}(t)$.

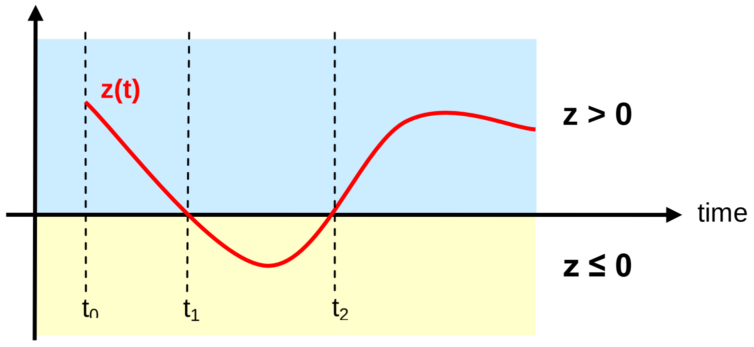


Figure 4: An event occurs when the event indicator changes its domain from z > 0 to z ≤ 0 or vice versa.

3. At every completed step of an integrator, `fmiCompletedIntegratorStep` must be called (provided the capability flag `ModelDescription.completedIntegratorStepNotNeeded = false`). An event occurs at this time instant, if indicated by the return argument `callEventUpdate`. Such an event is called <u>step event</u>. [*Step events are, for example used to dynamically change the (continuous) states of a model, because the previous states are no longer suited numerically.*]

The event handling at t_i is initiated by the environment by calling `fmiEventUpdate`.

A model (FMU) may have additional variables **p**, **u**, **y**, **v** as defined below. These symbols characterize sets of real integer, logical, and string variables that are piecewise continuous over time, respectively. The non-real variables change their values only at events. For example this means that $u_j(t) = u_j(t_i)$, for $t_i \le t < t_{i+1}$, if u_j is an integer, logical or string variable. If u_j is a real variable, it is either a continuous function of time

[5] Input signals are defined below.

[6] This definition is slightly different from the standard definition of state events: "$z_j(t) \cdot z_j(t_{i-1}) \le 0$". This often used definition has the severe drawback that $z_j(t_{i-1}) \neq 0$ is required in order to be well-defined and this condition cannot be guaranteed.

inside this interval or it is constant in this interval (= time-discrete). This property is defined in the Model Description File (see section 2.2.7). The variables have the following meaning:

- Independent parameters $\mathbf{p}(t) = \mathbf{p}(t_0)$ for $t \geq t_0$. The values of these variables are constant after initialization. Variables of this type are defined with attribute `causality = "parameter"` (if `variability="tunable"`, the parameter values can be changed at event instants, and this is conceptually treated as starting a new simulation run), see section 2.2.7.
- Input variables $\mathbf{u}(t)$. The values of these variables are defined outside of the model. Variables of this type are defined with attribute `causality = "input"`, see section 2.2.7.
- Output variables $\mathbf{y}(t)$. These variables are designed to be used in a model connection. So output variables might be used in the calling function as input values to other FMUs or other submodels. Variables of this type are defined with attribute `causality = "output"`, see section 2.2.7.
- All exposed variables $\mathbf{v}$(t). These are parameter, input, output, state and state derivative variables, as well as local variables that are not used in connections and are only exposed by the model to inspect results.

Based on the above prerequisites, the mathematical description of an FMU is defined as:

description	**range of t**	**equation**	**function names**

initialization	$t = t_0$	$(\mathbf{x},\mathbf{m},\mathbf{p},T_{next}) = \mathbf{f}_0(\mathbf{u},t_0,\text{subset of } \mathbf{v}_0)$	`fmiInitializeModel`
			`fmiGetReal/Integer/Boolean/String`
			`fmiGetContinuousStates`
			`fmiGetNominalContinuousStates`

derivatives $\dot{\mathbf{x}}(t)$	$t_i \leq t < t_{i+1}$	$\dot{\mathbf{x}} = \mathbf{f}_x(\mathbf{x},\mathbf{m},\mathbf{u},\mathbf{p},t)$	`fmiGetDerivatives`
outputs $\mathbf{y}(t)$	$t_i \leq t < t_{i+1}$	$\mathbf{y} = \mathbf{f}_y(\mathbf{x},\mathbf{m},\mathbf{u},\mathbf{p},t)$	`fmiGetReal/Integer/Boolean/String`
exposed variables $\mathbf{v}(t)$	$t_i \leq t < t_{i+1}$	$\mathbf{v} = \mathbf{f}_v(\mathbf{x},\mathbf{m},\mathbf{u},\mathbf{p},t)$	`fmiGetReal/Integer/Boolean/String`
event indicators $\mathbf{z}(t)$	$t_i \leq t < t_{i+1}$	$\mathbf{z} = \mathbf{f}_z(\mathbf{x},\mathbf{m},\mathbf{u},\mathbf{p},t)$	`fmiGetEventIndicators`

event update	$t = t_{i+1}$	$(\mathbf{x},\mathbf{m},T_{next}) = \mathbf{f}_m(\mathbf{x}^-,\mathbf{m}^-,\mathbf{u},\mathbf{p},t_{i+1})$	`fmiEventUpdate`
			`fmiCompletedEventIteration`
			`fmiGetReal/Integer/Boolean/String`
			`fmiGetContinuousStates`
			`fmiGetNominalStates`
			`fmiGetStateValueReferences`

event $t = t_{i+1}$ is triggered **if**		$t = T_{next}(t_i) \ \textbf{or} \ \min_{t>t_i} t : \left(z_j(t) > 0\right) \neq \left(z_j(t_i) > 0\right) \ \textbf{or} \ \text{step event}$	

$t \in \mathbb{R}$, $\mathbf{p} \in \mathbb{P}^{np}$, $\mathbf{u}(t) \in \mathbb{P}^{nu}$, $\mathbf{m}(t) \in \mathbb{P}^{nm}$, $\mathbf{x}(t) \in \mathbb{R}^{nx}$, $\mathbf{y}(t) \in \mathbb{P}^{ny}$, $\mathbf{v}(t) \in \mathbb{P}^{nv}$, $\mathbf{z}(t) \in \mathbb{R}^{nz}$

($\mathbb{R}$: real variable; $\mathbb{P}$: real **or** integer **or** Boolean **or** string variable)

$\mathbf{f}_x, \mathbf{f}_y, \mathbf{f}_v, \mathbf{f}_z \in C^0$ (= continuous functions with respect to all input arguments) inside $t_i \leq t < t_{i+1}$

where $t_i = \lim_{\varepsilon \to 0}(t_i + \varepsilon)$ and for all variables $\mathbf{v}$: $\mathbf{v}(t_i)$ is the right limit of $\mathbf{v}$ at t_i.

Table 1: Mathematical description of an FMU for Model Exchange.

An FMU is initialized with $\mathbf{f}_0(...)$. In order to remain flexible and allow using special initialization algorithms inside the model, the input arguments to this function are defined in the description schema (see section 2.2.7). This includes initial variable values, as well as guess values for iteration variables of algebraic equation systems, in order to compute the continuous and discrete states at the initial time t_0.

In the above table, this situation is described by stating that part of the input arguments to $\mathbf{f}_0$(...) are a subset of the initial values of all time varying variables appearing in the model equations. For example initialization might be defined by the initial states, $\mathbf{x}_0$, or by stating that the state derivatives are zero ($\dot{\mathbf{x}} = \mathbf{0}$). Initialization is a difficult topic by itself and it is assumed that the modeling environment generating the model code provides the initialization.

After initialization, integration is started. Basically, in this phase the derivatives of the continuous states are computed with $\mathbf{f}_x$(...). If FMUs and/or submodels are connected together, then the inputs of these models are the outputs of other models and therefore $\mathbf{f}_y$(...) must be called to compute outputs. Whenever result values shall be stored, usually at output points defined before the start of the simulation, function $\mathbf{f}_v$(...) must be called.

Continuous integration is stopped at an event instant. An event instant is determined by a time, state, or step event. In order to determine a state event, function $\mathbf{f}_z$(...) has to be called at every completed integrator step. Once the event indicators signal a change of their domain, an iteration over time is performed between the previous and the actual completed integrator step, in order to determine the time instant of the domain change up to a certain precision.

After an event is triggered, function $\mathbf{f}_m$(...) is called. This function returns with the new values of the (time-) continuous and (time-) discrete states. As input arguments the values of the states are used, just before the event was triggered. Inside function $\mathbf{f}_m$(...), an event iteration may take place until the new state values are determined. This might be a simple fixed point iteration, or the solution of a mixed equation system, with real, integer, logical and string unknowns.

The function calls in the table above describe precisely, which input arguments are needed to compute the desired output argument(s). There is no 1:1 mapping of these mathematical functions to C functions. Instead, all input arguments are set with `fmiSetXXX(..)` C-function calls and then the result argument(s) can be determined with the C functions defined in the right column of the above table. This technique is discussed in detail in section 3.2.1. [*In short: For efficiency reasons, all equations from the table above will usually be available in one (internal) C-function. With the C functions described in the next sections, input arguments are copied into the internal model data structure only when their value has changed in the environment. With the C functions in the right column of the table above, the internal function is called in such a way, that only the minimum needed equations are evaluated. Hereby, variable values calculated from previous calls can be reused. This technique is called "caching" and can significantly enhance the simulation efficiency of real-world models.*]

3.2 FMI Application Programming Interface

This section contains the interface description to evaluate different model parts from a C program.

3.2.1 Creation and Destruction of FMU instances

This section documents functions that deal with instantiation and destruction of FMUs.

```
fmiComponent fmiInstantiateModel(fmiString  instanceName, fmiString fmuGUID,
                                 fmiString  fmuResourceLocation,
                                 const fmiCallbackFunctions* functions,
                                 fmiBoolean                  visible,
                                 fmiBoolean                  loggingOn);
```

Returns a new instance of an FMU. If a null pointer is returned, then instantiation failed. In that case, function "`functions->logger`" was called with detailed information about the reason. An FMU can be instantiated many times, if capability flag `canBeInstantiatedOnlyOncePerProcess = false`. This function must be called successfully, before any of the following functions can be called.

The arguments of this function are defined by `fmiInstantiateXXX` in section 2.1.5.

```
void fmiFreeModelInstance(fmiComponent c);
```
Disposes the given instance, unloads the loaded model, and frees all the allocated memory and other resources that have been allocated by the functions of the FMU interface. If a null pointer is provided for "c", the function call is ignored (does not have an effect).

3.2.2 Providing Independent Variables and Re-initialization of Caching

Depending on the situation, different variables need to be computed. In order to be efficient, it is important that the interface requires only the computation of variables that are needed in the present context. For example during the iteration of an integrator step, only the state derivatives need to be computed, provided the output of a model is not connected. It might be that at the same time instant other variables are needed. For example if an integrator step is completed, the event indicator functions need to be computed as well. For efficiency it is then important that in the call to compute the event indicator functions, the state derivatives are not newly computed, if they have been computed already at the present time instant. This means, the state derivatives shall be reused from the previous call. This feature is called "caching of variables" in the sequel.

Caching requires that the model evaluation can detect when the input arguments, like time or states, have changed. This is achieved by setting them explicitly with a function call, since every such function call signals precisely a change of the corresponding variables. For this reason, this section contains functions to set the input arguments of the equation evaluation functions. This is unproblematic for time and states, but is more involved for parameters and inputs, since the latter may have different data types.

```
fmiStatus fmiSetTime(fmiComponent c, fmiReal time);
```
Set a new time instant and re-initialize caching of variables that depend on time (variables that depend solely on constants or parameters need not to be newly computed in the sequel, but the previously computed values can be reused).

```
fmiStatus fmiSetContinuousStates(fmiComponent c, const fmiReal x[], size_t nx);
```
Set a new (continuous) state vector and re-initialize caching of variables that depend on the states. Argument `nx` is the length of vector `x` and is provided for checking purposes (variables that depend solely on constants, parameters, time, and inputs do not need to be newly computed in the sequel, but the previously computed values can be reused). Note, `fmiEventUpdate` might change the continuous states as well.
Note: `fmiStatus = fmiDiscard` is possible.

```
fmiStatus fmiSetXXX(..);
```
Set new values for (independent) parameters and inputs and re-initialize caching of variables that depend on these variables. The details of these functions are defined in section 2.1.6.

```
fmiStatus fmiCompletedIntegratorStep(fmiComponent c,
                                     fmiBoolean* callEventUpdate);
```
This function must be called by the environment after every completed step of the integrator (provided the capability flag `completedIntegratorStepNotNeeded = false`). If the function returns with `callEventUpdate = fmiTrue`, then the environment has to call `fmiEventUpdate(..)`, otherwise, no action is needed.

When the integrator step is completed and the states are modified by the integrator afterwards (for example correction by a BDF method), then `fmiSetContinuousStates(..)` has to be called with the updated states before `fmiCompletedIntegratorStep(..)` is called.

After `fmiCompletedIntegratorStep` is called, it is still allowed to go back in time (calling `fmiSetTime`) and inquire values of variables at previous time instants with `fmiGetXXX` [*for* *example to determine values of non-state variables at output points*]: However, it is not allowed to go back in time over the previous `completedIntegratorStep` or the previous `fmiEventUpdate` call.

[*This function might be used, for example for the following purposes:*

1. *Delays:*
   All variables that are used in a "delay(..)" operator are stored in an appropriate buffer and the function returns with `callEventUpdate = fmiFalse`.
2. *Dynamic state selection:*
   It is checked whether the dynamically selected states are still numerically appropriate. If yes, the function returns with `callEventUpdate = fmiFalse` otherwise with `fmiTrue`. In the latter case, `fmiEventUpdate(..)` has to be called and changes the states dynamically.

]

The functions above have the slight drawback that values must always be copied, for example a call to "`fmiSetContinuousStates`" will provide the actual states in a vector and this function has to copy the values in to the internal model data structure "`c`" so that subsequent evaluation calls can utilize these values. If this turns out to be an efficiency issue, a future release of FMI might provide additional functions to provide the address of a memory area where the variable values are present.

3.2.3 Evaluation of Model Equations

This section contains the core functions to evaluate the model equations. Before one of these functions can be called, the appropriate functions from the previous section have to be used, to set the input arguments to the current model evaluation.

```
fmiStatus fmiInitializeModel(fmiComponent c, fmiBoolean toleranceControlled,
                             fmiReal relativeTolerance,
                             fmiEventInfo* eventInfo);

typedef struct{
  // only meaningful for fmiEventUpdate
  // (fmiInitializeModel returns with fmiTrue):
  fmiBoolean iterationConverged;
  fmiBoolean stateValueReferencesChanged; // valueReferences of states x changed
  fmiBoolean stateValuesChanged;          // values of states x changed

  // meaningful for fmiInitializeModel and for fmiEventUpdate:
  fmiBoolean terminateSimulation;
  fmiBoolean upcomingTimeEvent;  // if fmiTrue, nextEventTime is next time event
  fmiReal    nextEventTime;
} fmiEventInfo;
```

Initializes the model, in other words computes initial values for all variables. Before calling this function, `fmiSetTime` must be called, and all variables with a "ScalarVariable / <type> / start" attribute or a setting of `ScalarVariable.causality` = "`input`" can be set with the "`fmiSetXXX`" functions (the ScalarVariable attributes are defined in the Model Description File, see section 0). Setting other variables is not allowed.

If "`toleranceControlled = fmiTrue`" then the model is called with a numerical integration scheme where the step size is controlled by using "`relativeTolerance`" for error estimation. In such a case, all numerical algorithms used inside the model (for example to solve non-linear algebraic equations) should also operate with an error estimation of an appropriate smaller relative tolerance.

Once initialization is completed the function returns (or when used in `fmiEventUpdate`, when a new consistent state has been found) and the integration can be restarted. The function returns with `eventInfo`. This structure is also used as return value of `fmiEventUpdate`. The variables of the structure have the following meaning:

Arguments `iterationConverged`, `stateValueReferencesChanged`, and `stateValuesChanged` are only meaningful when returning from `fmiEventUpdate`. When returning from `fmiInitializeModel`, all three flags are always `fmiTrue`.

If `stateValuesChanged = fmiTrue` when `iterationConverged = fmiTrue`, then at least one element of the continuous state vector has changed its value, for example since at initial time, or due to an impulse. The new values of the states must be inquired with `fmiGetContinuousStates`.

If `stateValueReferencesChanged = fmiTrue` when `iterationConverged = fmiTrue`, then the meaning of the states has changed. The valueReferences of the new states can be inquired with `fmiGetStateValueReferences` and the nominal values of the new states can be inquired with `fmiGetNominalContinuousStates`.

If `terminateSimulation = fmiTrue`, the simulation shall be terminated (successfully). It is assumed that an appropriate message is printed by logger function (see section 2.1.5) to explain the reason for the termination.

If `upcomingTimeEvent = fmiTrue`, then the simulation shall integrate at most until time = `nextEventTime`, and shall call `fmiEventUpdate` at this time instant. If integration is stopped before `nextEventTime`, for example due to a state event, the definition of `nextEventTime` becomes obsolete.

[*Currently, this function can only be called <u>once</u> for <u>one</u> instance. Note, even if it can only be called once, an event can be triggered and then event iteration via fmiEventUpdate is possible at the initial time.*]

```
fmiStatus fmiGetDerivatives    (fmiComponent c, fmiReal derivatives[],size_t nx);
fmiStatus fmiGetEventIndicators(fmiComponent c, fmiReal eventIndicators[],
                                                size_t ni);
```

Compute state derivatives and event indicators at the current time instant and for the current states. The derivatives are returned as a vector with “`nx`” elements. A state event is triggered when the domain of an event indicator changes from $z_j > 0$ to $z_j \leq 0$ or vice versa (see section 3.1). The FMU must guarantee that at an event restart $z_j \neq 0$, for example by shifting z_j with a small value. Furthermore, z_j should be scaled in the FMU with its nominal value (so all elements of the returned vector “eventIndicators” should be in the order of “one”). The event indicators are returned as a vector with “`ni`” elements.

The ordering of the elements of the derivatives vector is identical to the ordering of the state vector (for example `derivatives[2]` is the derivative of `x[2]`). Event indicators are not necessarily related to variables on the Model Description File.

Note: `fmiStatus = fmiDiscard` is possible for both functions.

```
fmiStatus fmiEventUpdate(fmiComponent c, fmiBoolean intermediateResults,
                         fmiEventInfo* eventInfo);
typedef struct{..} fmiEventInfo;   // see fmiInitializeModel(..)
```

This function is called after a time, state or step event occurred. The function returns with `eventInfo` (for details see `fmiInitializeModel`). If “`intermediateResults = fmiFalse`”, the function returns once a new consistent state has been found. If the argument is `fmiTrue`, then the function returns for every event iteration that is performed internally, in order to allow getting result variables after every iteration with the `fmiGetXXX` functions above.

The function has to be called successively until "`eventInfo->iterationConverged =fmiTrue`" and the event iteration caused by the enclosing environment has converged.

`fmiStatus fmiCompletedEventIteration(fmiComponent c);`

This function has to be called after the global event iteration over all involved FMUs and other models has converged.

[*This function might be used for the following purposes:*

- *If the FMU stores results internally on file, then the results after the event has been processed can be stored.*
- *If the FMU contains dynamically changing states, then a new state selection might be performed before the simulation restart.*

]

`fmiStatus fmiGetContinuousStates(fmiComponent c, fmiReal x[], size_t nx);`

Return the new (continuous) state vector `x` after an event iteration has finished (including initialization). This function has to be called after initialization and if the (continuous) state vector has changed at an event instant after calling `fmiEventUpdate`(..) with `eventInfo->iterationConverged =fmiTrue`.

```
fmiStatus fmiGetNominalContinuousStates(fmiComponent c, fmiReal x_nominal[],
                                        size_t nx);
```

Return the nominal values of the continuous states. This function should always be called after `fmiInitialize`, and if `eventInfo->stateValueReferencesChanged = fmiTrue` in `fmiEventUpdate`, since then the association of the continuous states to variables has changed and therefore also their nominal values. If the FMU does not have information about the nominal value of a continuous state `i`, a nominal value `x_nominal[i] = 1.0` should be returned.

[*Typically, the nominal values of the continuous states are used to compute the absolute tolerance required by the integrator. Example:*

```
absoluteTolerance[i] = 0.01*relativeTolerance*x_nominal[i];]
```

```
fmiStatus fmiGetStateValueReferences(fmiComponent c, fmiValueReference vrx[],
                                     size_t nx);
```

Return the value references of the state vector. This function is only needed in case of dynamic state selection: In this case some of the exposed states are "dummy" states. Depending on the state selection algorithm, different actual variables might be selected as states. This function returns the value references to the actually used states, if they correspond to defined variables that are exposed. These value references may change after calling `fmiEventUpdate`(..). In this case `fmiEventUpdate` returns with `eventInfo->stateValueReferencesChanged = fmiTrue`.

If `vrx[i] = fmiUndefinedValueReference` (see section 2.1.2), either no actual variable is used as a state (but maybe a linear combination of variables), or the model is hiding the meaning of the state and no value reference for this state is returned.

`fmiStatus fmiTerminate(fmiComponent c);`

Terminate the model evaluation at the end of a simulation or after a desired stop of the integration before the simulation end. Release all resources that have been allocated since `fmiInitializeModel` has been called. After calling this function, the final values of all variables can be inquired with the `fmiGetXXX(..)` functions above. It is not allowed to call this function after one of the functions returned with a status flag of `fmiError` or `fmiFatal`.

3.2.4 State Machine of Calling Sequence

Every implementation of the FMI must support calling sequences of the functions according to the following state chart:

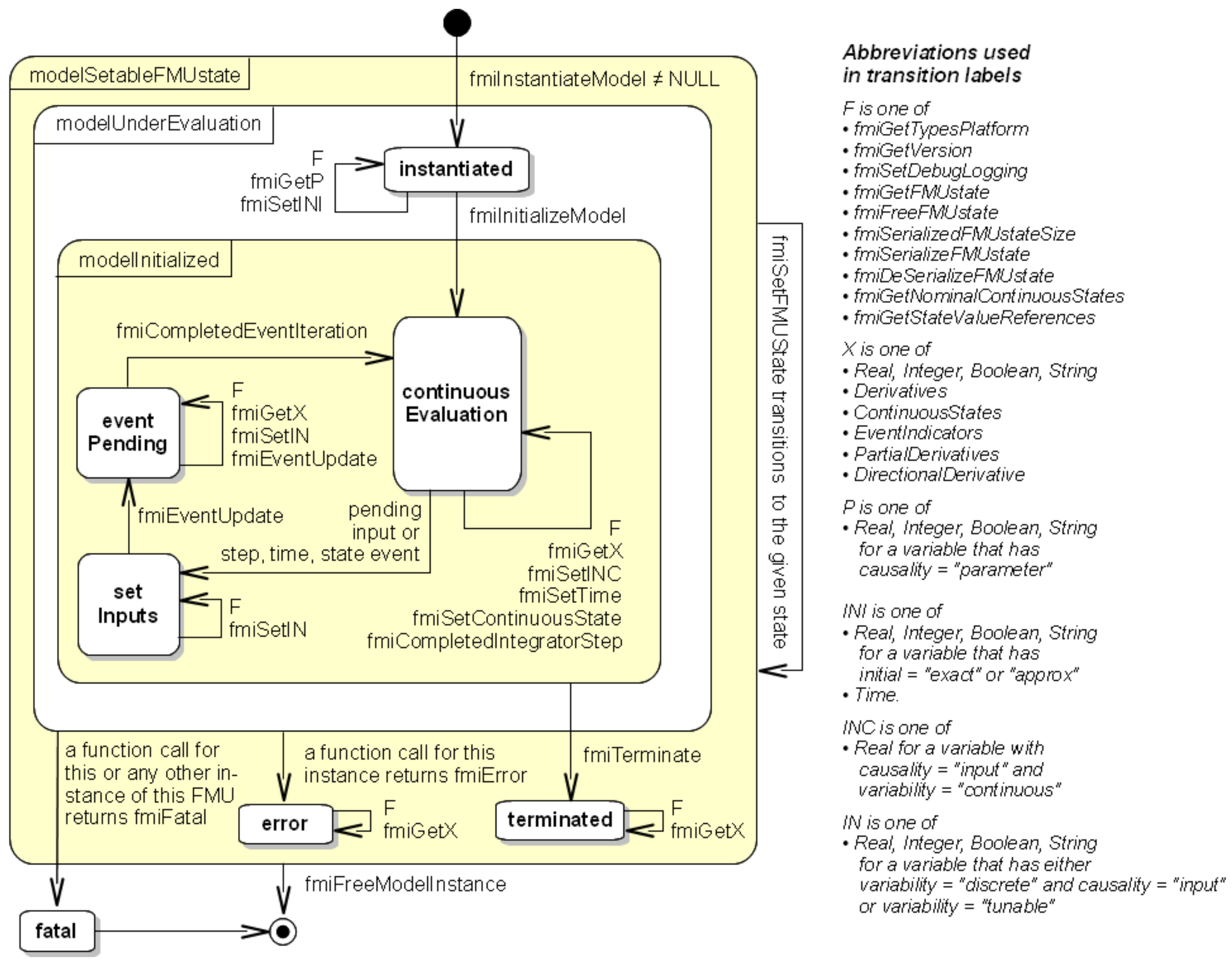

modelSetableFMUstate
fmiInstantiateModel ≠ NULL
modelUnderEvaluation
F
fmiGetP
fmiSetINI
instantiated
fmiInitializeModel
modelInitialized
fmiSetFMUState transitions to the given state
fmiCompletedEventIteration
continuous Evaluation
event Pending
F
fmiGetX
fmiSetIN
fmiEventUpdate
fmiEventUpdate
pending input or step, time, state event
set Inputs
F
fmiSetIN
F
fmiGetX
fmiSetINC
fmiSetTime
fmiSetContinuousState
fmiCompletedIntegratorStep
a function call for this or any other instance of this FMU returns fmiFatal
a function call for this instance returns fmiError
fmiTerminate
error
F
fmiGetX
terminated
F
fmiGetX
fmiFreeModelInstance
fatal
Abbreviations used in transition labels
F is one of
• fmiGetTypesPlatform
• fmiGetVersion
• fmiSetDebugLogging
• fmiGetFMUstate
• fmiFreeFMUstate
• fmiSerializedFMUstateSize
• fmiSerializeFMUstate
• fmiDeSerializeFMUstate
• fmiGetNominalContinuousStates
• fmiGetStateValueReferences
X is one of
• Real, Integer, Boolean, String
• Derivatives
• ContinuousStates
• EventIndicators
• PartialDerivatives
• DirectionalDerivative
P is one of
• Real, Integer, Boolean, String for a variable that has causality = "parameter"
INI is one of
• Real, Integer, Boolean, String for a variable that has initial = "exact" or "approx"
• Time.
INC is one of
• Real for a variable with causality = "input" and variability = "continuous"
IN is one of
• Real, Integer, Boolean, String for a variable that has either variability = "discrete" and causality = "input" or variability = "tunable"

Figure 5: Calling sequence of Model Exchange C functions in form of an UML 2.0 state machine.

The objective of the start chart is to define the allowed calling sequences for functions of the FMI: calling sequences not accepted by the state chart are not supported by the FMI. The behaviour of an FMU is undefined for such a calling sequence. For example, the state chart indicates that when an FMU for Model Exchange is in state “modelInitialized”, a call to fmiSetReal for a discrete input directly followed by a call to fmiGetReal is not supported. The state chart is given here as UML 2.0 state machine. If a transition is labelled with one or more function names (for example fmiGetReal, fmiGetInteger) this means that the transition is taken if any of these functions is successfully called. Note that the FMU cannot determine in general in which state it is by just keeping track of the function calls it receives. This is because the environment decides when to proceed to the next phase of the solution process without always indicating this choice by a dedicated function call to the FMU. For example, the environment might choose to transition from state “continuousEvaluation” to state “setInputs” when it detects that a discrete input of the FMU has changed its value. In this case, the transition condition “pending input” fires and triggers transition to state '”setInputs”. But the FMU can not infer the state transition at this time, because the environment does not inform the FMU about this state change.

The transition conditions "step event", "time event", and "state event" are defined in section 3.1. Each state of the state machine corresponds to a certain phase of a simulation as follows:

- **instantiated**:
  In this state, inputs, start and guess values can be set.
- **continuousEvaluation**:
  In this state, the solution at initial time, after a completed integrator step, or after event iteration can be retrieved. Also an integrator step is performed and the event time of a state event may be determined here after a domain change of at least one event indicator was detected at the end of a completed integrator step. If `fmiInitialize` or `fmiEventUpdate` return with `eventInfo.terminated = fmiTrue`, a transition to state "terminated" occurs.
- **setInputs**:
  Before starting with the event handling, changed (continuous or discrete) inputs have to be set. The state machine changes from state "continuousEvaluation" to state "setInputs" when an event is triggered. In order to avoid inconsistencies and/or unnecessary model evaluations, it is not allowed to call `fmiGetX` in state "setInputs" until `fmiEventUpdate` is called [*otherwise, if for example a new value of u1 is set and an output y1 is inquired then y1 has to be newly computed based on u1 although not yet all inputs have been set before, so the value of y1 does not make sense; in other words, setting the inputs has to be treated as one atomic operation*].
- **eventPending**:
  In this state, at least one event is waiting to be processed by a call to `fmiEventUpdate`. Intermediate results of the event iteration can be retrieved. If `fmiEventUpdate` returns with `eventInfo.iterationConverged = fmiTrue`, and the global event iteration over all connected FMUs terminated, then `fmiCompletedEventIteration(..)` must be called and then this state is left and the state machine continues in state "`continuousEvaluation`".
- **terminated**:
  In this state, the solution at the final time of a simulation can be retrieved.

Note, that simulation backward in time is only allowed over continuous time intervals. As soon as an event occured (`fmiEventUpdate` was called) going back in time is forbidden, because `fmiEventUpdate` can only compute the next discrete state, not the previous one.

The allowed function calls in the respective states are summarized in the following table (functions marked in "yellow" are only available for "Model Exchange", the other functions are available both for "Model Exchange" and "Co-Simulation"):

	FMI 2.0 for Model Exchange							
Function	start, end	instantiated	continuousEvaluation	setInputs	eventPending	terminated	error	fatal
fmiGetTypesPlatform	x	x	x	x	x	x	x	
fmiGetVersion	x	x	x	x	x	x	x	
fmiSetDebugLogging		x	x	x	x	x	x	
fmiGetReal		5	x		x	x	7	
fmiGetInteger		5	x		x	x	7	
fmiGetBoolean		5	x		x	x	7	
fmiGetString		5	x		x	x	7	
fmiSetReal		1	3	4	4			
fmiSetInteger		1		4	4			
fmiSetBoolean		1		4	4			
fmiSetString		1		4	4			
fmiGetFMUstate		x	x	x	x	x	7	
fmiSetFMUstate		x	x	x	x	x	x	
fmiFreeFMUstate		x	x	x	x	x	x	
fmiSerializedFMUstateSize		x	x	x	x	x	x	
fmiSerializeFMUstate		x	x	x	x	x	x	
fmiDeSerializeFMUstate		x	x	x	x	x	x	
fmiGetPartialDerivatives			x		x	x	7	
fmiGetDirectionalDerivative			x		x	x	7	
fmiInstantiateModel	x							
fmiFreeModelInstance		x	x	x	x	x	x	
fmiSetTime		x	x					
fmiSetContinuousStates			x					
fmiCompletedEventIteration					x			
fmiCompletedIntegratorStep			x					
fmiInitializeModel		x						
fmiEventUpdate				x	x			
fmiTerminate			x	x	x			
fmiGetDerivatives			x		x	x	7	
fmiGetEventIndicators			x		x	x	7	
fmiGetContinuousStates			x		x	x	7	
fmiGetNominalContinuousStates		x	x	x	x	x	7	
fmiGetStateValueReferences		x	x	x	x	x	7	

x means: call is allowed in the corresponding state
number means: call is allowed if the indicated condition holds:
1 for a variable that has initial = "exact" or approx"
3 for a variable with causality = "input" and variability = "continuous"
4 for a variable with variability = "discrete" and causality = "input", or variability = tunable"
5 for a variable with causality = "parameter"
7 always, but retrieved values are usable for debugging only

3.2.5 Pseudo Code Example

In the following example, the usage of the fmiXXX functions is sketched in order to clarify the typical calling sequence of the functions in a simulation environment. The example is given in a mix of pseudo-code and “C”, in order to keep it small and understandable. Furthermore, it is assumed that one FMU is directly integrated in a simulation environment. If the FMU would be used inside another model, additional code is needed, especially initialization and event iteration has to be adapted.

```
m = M_fmiInstantiateModel("m", ...)  // "m" is the instance name
                                     // "M_" is the MODEL_IDENTIFIER
nx     = ...   // number of states, from XML file
nz     = ...   // number of event indicators, from XML file
Tstart = 0     // could also be retrieved from XML file
Tend   = 10    // could also be retrieved from XML file
dt     = 0.01  // fixed step size of 10 milli-seconds

// set the start time
Tnext = Tend
time  = Tstart
M_fmiSetTime(m, time)

// set all variable start values (of "ScalarVariable / <type> / start") and
// set the input values at time = Tstart
M_fmiSetReal/Integer/Boolean/String(m, ...)

// initialize
M_fmiInitializeModel(m, fmiFalse, 0.0, &eventInfo)

// retrieve initial state x and
// nominal values of x (if absolute tolerance is needed)
M_fmiGetContinuousStates(m, x, nx)
M_fmiGetNominalContinuousStates(m, x_nominal, nx)

// retrieve solution at t=Tstart, for example for outputs
M_fmiGetReal/Integer/Boolean/String(m, ...)

while time < Tend and not eventInfo.terminateSimulation loop
  // compute derivatives
  M_fmiGetDerivatives(m, der_x, nx)

  // advance time
  h    = min(dt, Tnext-time)
  time = time + h
  M_fmiSetTime(m, time)

  // set inputs at t = time
  M_fmiSetReal/Integer/Boolean/String(m, ...)

  // set states at t = time (perform one step)
  x = x + h*der_x  // forward Euler method
  M_fmiSetContinuousStates(m, x, nx)
  // get event indicators at t = time
  M_fmiGetEventIndicators(m, z, nz)

  // inform the model about an accepted step
  M_fmiCompletedIntegratorStep(m, &callEventUpdate)

  // handle events, if any
  time_event  = abs(time - Tnext) <= eps
  state_event = ...        // compare sign of z with previous z
  if callEventUpdate or time_event or state_event then
    eventInfo.iterationConverged = fmiFalse

    while eventInfo.iterationConverged == fmiFalse loop //event iteration
      M_fmiEventUpdate(m, fmiTrue, &eventInfo)

      // retrieve solution at every event iteration
      if eventInfo.iterationConverged == fmiFalse then
         M_fmiGetReal/Integer/Boolean/String(m, ...)
      end if
    end while
    M_fmiCompletedEventIteration(m)

    if eventInfo.stateValuesChanged == fmiTrue then
      //the model signals a value change of states, retrieve them
      M_fmiGetContinuousStates(m, x, nx)
    end if

    if eventInfo.stateValueReferencesChanged = fmiTrue then
      //the meaning of states has changed; retrieve new nominal values
      M_fmiGetNominalContinuousStates(m, x_nominal, nx)
    end if


    if eventInfo.upcomingTimeEvent then
       Tnext = min(eventInfo.nextEventTime, Tend)
    else
       Tnext = Tend
    end if
  end if

  // Retrieve solution at t=time, for example for outputs
  M_fmiGetReal/Integer/Boolean/String(m, ...)
end while

// terminate simulation and retrieve final values
M_fmiTerminate(m)
M_fmiGetReal/Integer/Boolean/String(m, ...)

// cleanup
M_fmiFreeModelInstance(m)
```

Above, errors are not handled. Typically, `fmiXXX` function calls are performed in the following way:

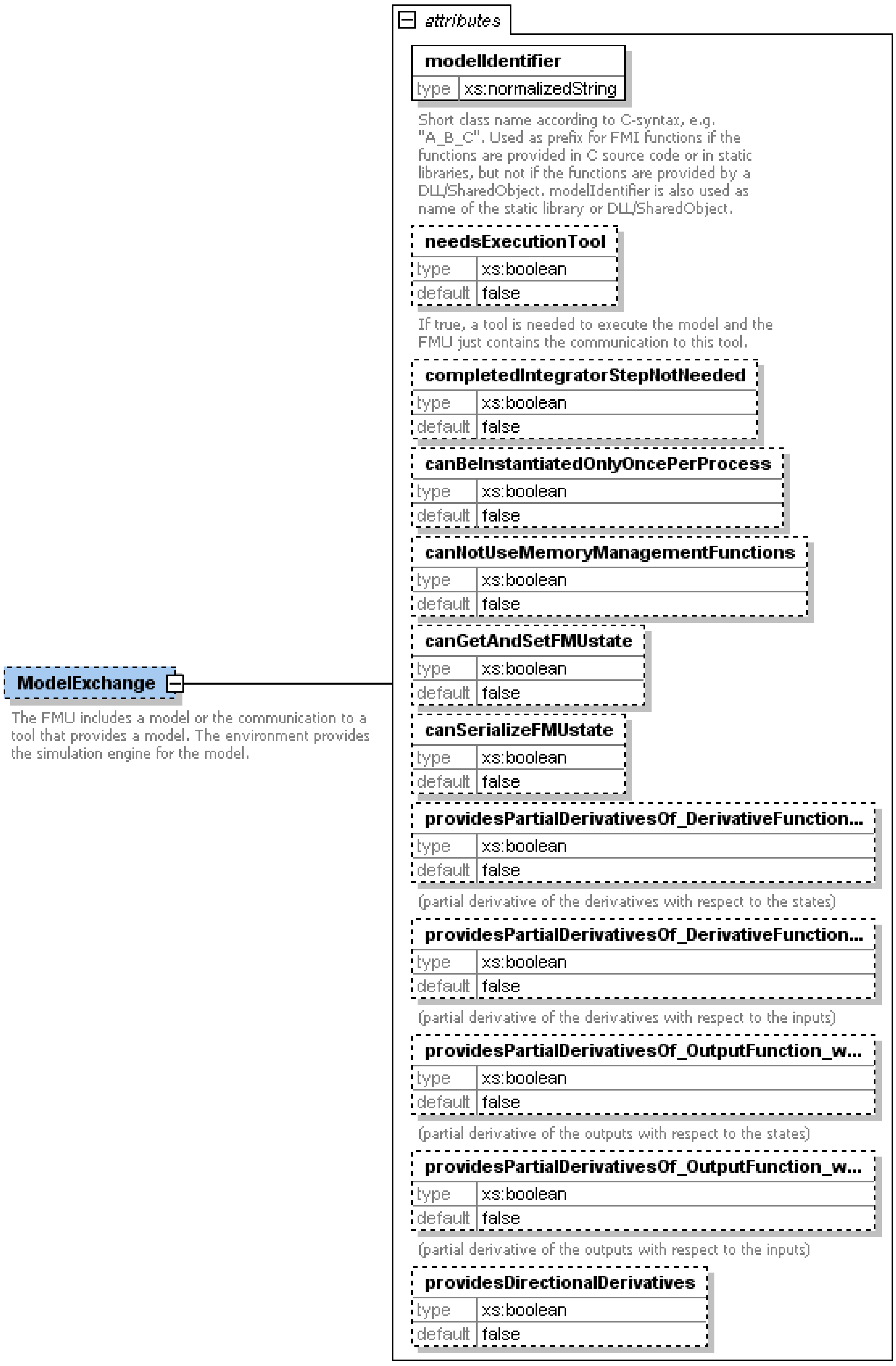
```
status = M_fmiGetDerivatives(m, der_x, nx);
switch ( status ) { case fmiDiscard: ....; break; // reduce step size and try again
                    case fmiError  : ....; break; // cleanup and stop simulation
                    case fmiFatal  : ....; }      // stop using the model
The switch statement could also be stored in a macro to simplify the code.
```

3.3 FMI Description Schema

This is defined in section 2.2. Additionally, the “Model Exchange” specific element “ModelExchange” is defined in the next section.

3.3.1 Model Exchange FMU (fmiModelDescription.ModelExchange)

If the XML file defines an FMU for Model Exchange, element “ModelExchange” must be present. It is defined as:



Short class name according to C-syntax, e.g. "A_B_C". Used as prefix for FMI functions if the functions are provided in C source code or in static libraries, but not if the functions are provided by a DLL/SharedObject. modelIdentifier is also used as name of the static library or DLL/SharedObject.

If true, a tool is needed to execute the model and the FMU just contains the communication to this tool.

(partial derivative of the derivatives with respect to the states)

(partial derivative of the derivatives with respect to the inputs)

(partial derivative of the outputs with respect to the states)

(partial derivative of the outputs with respect to the inputs)

The FMU includes a model or the communication to a tool that provides a model. The environment provides the simulation engine for the model.

The following attributes are defined (all of them are optional, with exception of "`modelIdentifier`"):

Attribute Name	*Description*
`modelIdentifier`	Short class name according to C syntax, for example "A_B_C". Used as prefix for FMI functions if the functions are provided in C source code or in static libraries, but not if the functions are provided by a DLL/SharedObject. `modelIdentifier` is also used as name of the static library or DLL/SharedObject . See also section 2.1.1.
`needsExecutionTool`	If true, a tool is needed to execute the model and the FMU just contains the communication to this tool. [*Typically, this information is only utilized for information purposes. For example when loading an FMU with* `needsExecutionTool = true`, *the environment can inform the user that a tool has to be available on the computer where the model is instantiated. The name of the tool can be taken from attribute* `generationTool` *of* `fmiModelDescription`.]
`completedIntegratorStepNotNeeded`	If `true`, function `fmiCompletedIntegratorStep` need not to be called (which gives a slightly more efficient integration). If it is called, it has no effect. If `false` (the default), the function must be called after every completed integrator step, see section 3.2.3.
`canBeInstantiatedOnlyOncePerProcess`	This flag indicates cases (especially for embedded code), where only one instance per FMU is possible (multiple instantiation is default = `false`; if multiple instances are needed and this flag = `true`, the FMUs must be instantiated in different processes).
`canNotUseMemoryManagementFunctions`	If `true`, the FMU uses its own functions for memory allocation and freeing only. The callback functions `allocateMemory` and `freeMemory` given in `fmiInstantiateModel` are ignored.
`canGetAndSetFMUstate`	If `true`, the environment can inquire the internal FMU state and can restore it. That is, functions `fmiGetFMUstate`, `fmiSetFMUstate`, and `fmiFreeFMUstate` are supported by the FMU.
`canSerializeFMUstate`	If `true`, the environment can serialize the internal FMU state, in other words functions `fmiSerializedFMUstateSize`,

	`fmiSerializeFMUstate`, `fmiDeSerializeFMUstate` are supported by the FMU. If this is the case, then flag `canGetAndSetFMUstate` must be `true` as well.
The following capability attributes define whether “`fmiGetPartialDerivatives`” and “`fmiGetDirectionalDerivatives`” can be called to compute the partial derivatives defined below. Independently of these functions, the dependency of variables is provided in the XML file under “`ModelVariables.Derivatives/Outputs`”. It is assumed that models have the following structure (**u** are inputs, **x** are states, **y** are outputs): $$\frac{d\mathbf{x}}{dt}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$ $$\mathbf{y}_{cont}(t) = \mathbf{g}_{cont}(\mathbf{x}(t), \mathbf{u}_{cont}(t), \mathbf{u}_{disc}(t), t)$$	
`providesPartialDerivativesOf_DerivativeFunction_wrt_States`	
	If true, the partial derivative of the state derivatives function **f** with respect to the states is provided, in other words $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$
`providesPartialDerivativesOf_DerivativeFunction_wrt_Inputs`	
	If true, the partial derivative of the state derivatives function **f** with respect to the (continuous Real) inputs is provided, in other words $\frac{\partial \mathbf{f}}{\partial \mathbf{u}_{cont}}$
`providesPartialDerivativesOf_OutputFunction_wrt_States`	
	If true, the partial derivative of the (continuous Real) output functions **g** with respect to the states is provided, in other words $\frac{\partial \mathbf{g}_{cont}}{\partial \mathbf{x}}$
`providesPartialDerivativesOf_OutputFunction_wrt_Inputs`	
	If true, the partial derivative of the (continuous Real) output functions **g** with respect to the (continuous Real) inputs is provided, in other words $\frac{\partial \mathbf{g}_{cont}}{\partial \mathbf{u}_{cont}}$
`providesDirectionalDerivatives`	
	If true, the directional derivatives of the equations are provided by `fmiGetDirectionalDerivatives(..)`

The flags have the following default values.

- boolean: false
- unsignedInt: 0

3.3.2 Example XML Description File

When generating an FMU from the hypothetical model “MyLibrary.SpringMassDamper”, the XML file may have the following content:

```
<?XML version="1.0" encoding="UTF8"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="MyLibrary.SpringMassDamper"
  guid="{8c4e810f-3df3-4a00-8276-176fa3c9f9e0}"
  description="Rotational Spring Mass Damper System"
```

```
version="1.0"
generationDateAndTime="2011-09-23T16:57:33Z"
variableNamingConvention="structured"
numberOfEventIndicators="2">

<ModelExchange
  modelIdentifier="MyLibrary_SpringMassDamper"/>

<UnitDefinitions>
  <Unit name="rad">
    <BaseUnit rad="1"/>
    <DisplayUnit name="deg" factor="57.2957795130823"/>
  </Unit>
  <Unit name="rad/s">
    <BaseUnit s="-1" rad="1"/>
  </Unit>
  <Unit name="kg.m2">
    <BaseUnit kg="1" m="2"/>
  </Unit>
</UnitDefinitions>

<TypeDefinitions>    <SimpleType name="Modelica.SIunits.Inertia">
    <Real quantity="MomentOfInertia" unit="kg.m2" min="0.0"/>
  </SimpleType>
  <SimpleType name="Modelica.SIunits.Torque">
    <Real quantity="Torque" unit="N.m"/>
  </SimpleType>
  <SimpleType name="Modelica.SIunits.AngularVelocity">
    <Real quantity="AngularVelocity" unit="rad/s"/>
  </SimpleType>
  <SimpleType name="Modelica.SIunits.Angle">
    <Real quantity="Angle" unit="rad"/>
  </SimpleType>
</TypeDefinitions>

<DefaultExperiment startTime="0.0" stopTime="3.0" tolerance="0.0001"/>

<ModelVariables>
  <ScalarVariable
    name="inertia1.J"
    valueReference="1073741824"
    description="Moment of load inertia"
    causality="parameter"
    variability="fixed">
    <Real declaredType="Modelica.SIunits.Inertia" start="1"/>
  </ScalarVariable>

  <ScalarVariable
    name="torque.tau"
    valueReference="536870912"
    description="Accelerating torque acting at flange (= -flange.tau)"
    causality="input">
    <Real declaredType="Modelica.SIunits.Torque" />    </ScalarVariable>

  <ScalarVariable
    name="inertia1.phi"
```

```
      valueReference="805306368"
      description="Absolute rotation angle of component"
      causality="output">
      <Real declaredType="Modelica.SIunits.Angle" />
    </ScalarVariable>

    <ScalarVariable
      name="inertia1.w"
      valueReference="805306369"
      description="Absolute angular velocity of component (= der(phi))"
      causality="output">
      <Real declaredType="Modelica.SIunits.AngularVelocity" />
    </ScalarVariable>

    <ScalarVariable name="x[0]"      valueReference="0"> <Real/> </ScalarVariable>
    <ScalarVariable name="x[1]"      valueReference="1"> <Real/> </ScalarVariable>
    <ScalarVariable name="der(x[0])" valueReference="2"> <Real/> </ScalarVariable>
    <ScalarVariable name="der(x[1])" valueReference="3"> <Real/> </ScalarVariable>
  </ModelVariables>

  <ModelStructure>
    <Inputs>
      <Input name="torque.tau"/>
    </Inputs>
    <Derivatives>
      <Derivative name="der(x[0])" state="x[0]" />
      <Derivative name="der(x[1])" state="x[1]" />
    </Derivatives>
    <Outputs>
      <Output name="inertia1.phi" />
      <Output name="inertia1.w" />
    </Outputs>
  </ModelStructure>
</fmiModelDescription>
```

4. FMI for Co-Simulation

This chapter defines the Functional Mock-up Interface (FMI) for the coupling of two or more simulation models in a co-simulation environment (*FMI for Co-Simulation*). Co-simulation is a rather general approach to the simulation of coupled technical systems and coupled physical phenomena in engineering with focus on instationary (time-dependent) problems.

FMI for Co-Simulation is designed both for coupling with subsystem models, which have been exported by their simulators together with its solvers as runnable code (Figure 6), and for coupling of simulation tools (*simulator coupling*, *tool coupling* (Figure 7 and Figure 6)).

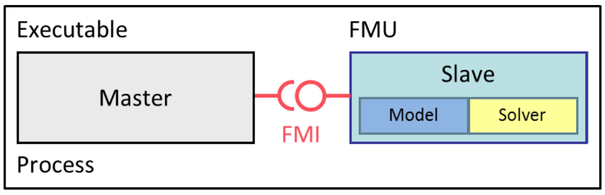


Figure 6: Co-simulation with generated code on a single computer
(for simplicity shown for one slave only).

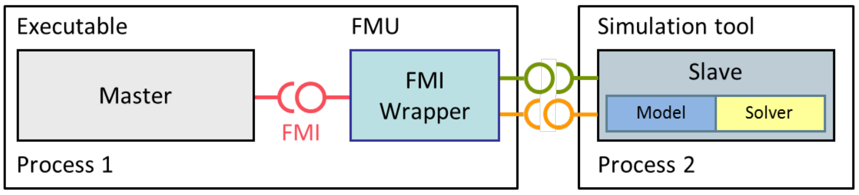


Figure 7: Co-simulation with tool coupling on a single computer
(for simplicity shown for one slave only).

In the tool coupling case the FMU implementation wraps the FMI function calls to API calls which are provided by the simulation tool (for example a COM or CORBA API). Additionally to the FMU the simulation tool is needed to run a co-simulation.

In its most general form, a tool coupling based co-simulation is implemented on distributed hardware with subsystems being handled by different computers with maybe different OS (cluster computer, computer farm, computers at different locations). The data exchange and communication between the subsystems is typically done using one of the network communication technologies (for example MPI, TCP/IP). The definition of this communication layer is not part of the FMI standard. However distributed co-simulation scenarios can be implemented using FMI as shown in Figure 8.

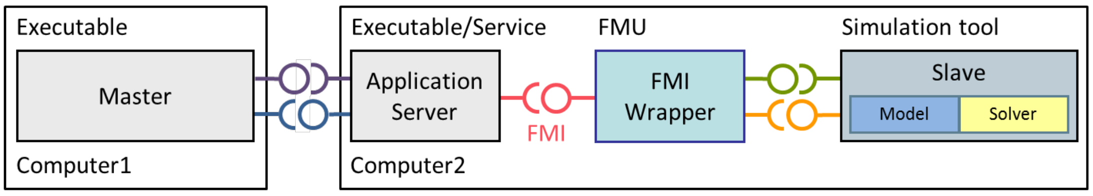


Figure 8: Distributed co-simulation infrastructure (for simplicity shown for one slave only).

The master has to implement the communication layer. Additional parameters for establishing the network communication (for example identification of the remote computer, port numbers, user account) are to be set via the GUI of the master. These data are not transferred via the FMI API.

4.1 Basics

Co-simulation exploits the modular structure of coupled problems in all stages of the simulation process beginning with the separate model setup and preprocessing for the individual *subsystems* in different *simulation tools* (which can be powerful simulators as well as simple C programs). During time integration, the simulation is again performed independently for all subsystems restricting the data exchange between subsystems to discrete *communication points* tc_i. For simulator coupling, also the visualization and post-processing of simulation data is done individually for each subsystem in its own native simulation tool. In different contexts, the communication points tc_i, the *communication steps* $tc_i \rightarrow tc_{i+1}$ and the *communication step sizes* $hc_i := tc_{i+1} - tc_i$ are also known as sampling points (synchronization points), macro steps and sampling rates, respectively. The term "communication point" in FMI for Co-Simulation refers to the communication between subsystems in a co-simulation environment and should not be mixed with the output points for saving simulation results to file.

FMI for Co-Simulation provides an interface standard for the solution of time dependent *coupled systems* consisting of subsystems that are continuous in time (model components that are described by instationary differential equations) or time-discrete (model components that are described by difference equations like, for example discrete controllers). In a block representation of the coupled system, the subsystems are represented by blocks with (internal) *state variables* $x(t)$ that are connected to other subsystems (blocks) of the coupled problem by *subsystem inputs* $u(t)$ and *subsystem outputs* $y(t)$. In this framework, the physical connections between subsystems are represented by mathematical coupling conditions between the inputs $u(t)$ and the outputs $y(t)$ of all subsystems, *Kübler and Schiehlen (2000)*.

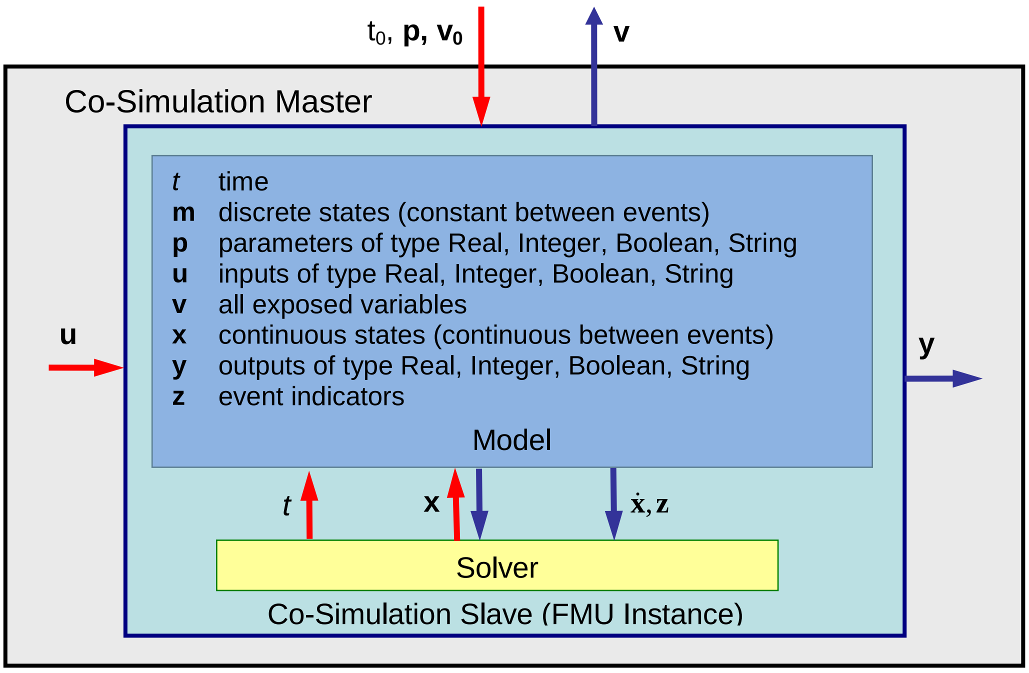

t0, p, v0
v
Co-Simulation Master
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
Model
u
y
t
x
ẋ, z
Solver
Co-Simulation Slave (FMU Instance)

Figure 9: Data flow at communication points.

For co-simulation two basic groups of functions have to be realized:

1. functions for the data exchange between subsystems and

2. functions for algorithmic issues to synchronize the simulation of *all* subsystems and to proceed in communication steps $tc_i \rightarrow tc_{i+1}$ from initial time $tc_0 := t_{start}$ to end time $tc_N := t_{stop}$.

In FMI for Co-Simulation both functions are implemented in one software component, the co-simulation *master*. The data exchange between the subsystems (*slaves*) is handled via the master only. There is no direct communication between the slaves. The master functionality can be implemented by a special software tool (a separate simulation backplane) or by one of the involved simulation tools. In its most general form, the coupled system may be simulated in *nested* co-simulation environments and FMI for Co-Simulation applies to each level of the hierarchy.

FMI for Co-Simulation defines interface routines for the communication between the master and all slaves (subsystems) in a co-simulation environment. The most common master algorithm stops at each communication point tc_i the simulation (time integration) of all slaves, collects the outputs $y(tc_i)$ from all subsystems, evaluates the subsystem inputs $u(tc_i)$, distributes these subsystem inputs to the slaves and continues the (co-)simulation with the next communication step $tc_i \rightarrow tc_{i+1} = tc_i + hc$ with fixed communication step size hc. In each slave, an appropriate solver is used to integrate one of the subsystems for a given communication step $tc_i \rightarrow tc_{i+1}$. The most simple co-simulation algorithms approximate the (unknown) subsystem inputs $u(t), (t > tc_i)$ by frozen data $u(tc_i)$ for $tc_i \le t < tc_{i+1}$. FMI for Co-Simulation supports this classical brute force approach as well as more sophisticated master algorithms. FMI for Co-Simulation is designed to support a very general class of master algorithms but it does *not* define the master algorithm itself.

The ability of slaves to support more sophisticated master algorithms is characterized by a set of *capability flags* inside the XML description of the slave (see section 4.3.1). Typical examples are:

- the ability to handle variable communication step sizes hc_i,
- the ability to repeat a rejected communication step $tc_i \rightarrow tc_{i+1}$ with reduced communication step size,
- the ability to provide derivatives w.r.t. time of outputs to allow interpolation (section 4.2.3),
- or the ability to provide Jacobians.

FMI for Co-Simulation is restricted to slaves with the following properties:

- All calculated values $v(t)$ are time dependent functions within an a priori defined time interval $t_{start} \le t \le t_{stop}$ (provided `StopTimeDefined = fmiTrue` when calling `fmiInstantiateSlave`).
- All calculations (simulations) are carried out with increasing time in general. The current time t is running step by step from t_{start} to t_{stop}. The algorithm of the slave may have the property to be able to repeat the simulation of parts of $[t_{start}, t_{stop}]$ or the whole time interval $[t_{start}, t_{stop}]$.
- The slave can be given a time value tc_i, $t_{start} \le tc_i \le t_{stop}$.
- The slave is able to interrupt the simulation when tc_i is reached.
- During the interrupted simulation the slave (and its individual solver) can receive values for inputs $u(tc_i)$ and send values of outputs $y(tc_i)$.
- Whenever the simulation in a slave is interrupted, a new time value tc_{i+1}, $tc_i \le tc_{i+1} \le t_{stop}$ can be given to simulate the time subinterval $tc_i < t \le tc_{i+1}$
- The subinterval length hc_i is the communication step size of the i^{th} communication step, $hc_i = tc_{i+1} - tc_i$. In general, the communication step size can be positive, zero, but not negative.

FMI for Co-Simulation allows a co-simulation flow which starts with instantiation and initialization (all slaves are prepared for computation, the communication links are established), followed by simulation (the slaves are forced to simulate a communication step), and finishes with shutdown. In detail the flow is defined in the state machine of the calling sequences from master to slave (section 0).

4.2 FMI Application Programming Interface

This section contains the interface description to access the in/output data and status information of a co-simulation slave from a C program.

4.2.1 Creation and Destruction of Co-Simulation Slaves

This section documents functions that deal with instantiation and destruction of FMUs.

```
fmiComponent fmiInstantiateSlave(fmiString instanceName, fmiString fmuGUID,
                                 fmiString fmuResourceLocation,
                                 const fmiCallbackFunctions* functions,
                                 fmiBoolean                  visible,
                                 fmiBoolean                  loggingOn)
```

Returns a new instance of a co-simulation FMU slave. If a null pointer is returned, then instantiation failed. In that case, function “`functions->logger`” was called with detailed information about the reason. An FMU can be instantiated many times, if capability flag `canBeInstantiatedOnlyOncePerProcess` is not set. This function must be called successfully, before any of the following functions can be called. It has to perform all actions of a slave which are necessary before a simulation run starts (for example loading the model file, compilation...).

The arguments of this function are defined by `fmiInstantiateXXX` in section 2.1.5.

```
void fmiFreeSlaveInstance(fmiComponent c);
```

Disposes the given instance, unloads the loaded model, and frees all the allocated memory and other resources that have been allocated by the functions of the FMU interface. If a null pointer is provided for “c”, the function call is ignored (does not have an effect).

4.2.2 Initialization and Termination of Co-Simulation Slaves

This section documents functions that deal with initialization and termination of co-simulation slaves.

```
fmiStatus fmiInitializeSlave(fmiComponent c,
                             fmiReal    relativeTolerance,
                             fmiReal    tStart,
                             fmiBoolean stopTimeDefined,
                             fmiReal    tStop);
```

Informs the slave that the simulation run starts now.

Argument “`relativeTolerance`” suggests a relative (local) tolerance in case the slave utilizes a numerical integrator with variable step size and error estimation.

The arguments `tStart` and `tStop` can be used to check whether the model is valid within the given boundaries or to allocate memory which is necessary for storing results. If `stopTimeDefined = fmiTrue`, then `tStop` is the defined stop time and if the master tries to compute past `tStop` the slave has to return `fmiStatus = fmiError`. If `stopTimeDefined = fmiFalse`, then no stop time is defined and argument `tStop` is meaningless.

```
fmiStatus fmiTerminateSlave(fmiComponent c);
```

Is called by the master to signal to the slave the end of the co-simulation run.

```
fmiStatus fmiResetSlave(fmiComponent c);
```

Is called by the master to reset the slave after a simulation run. Before starting a new run, `fmiInitializeSlave` has to be called.

4.2.3 Transfer of Input / Output Values and Parameters

Input and output variables and variables are transferred via the `fmiGetXXX` and `fmiSetXXX` functions, defined in section 2.1.6.

In order to enable the slave to interpolate the continuous real inputs between communication steps the derivatives of the inputs with respect to time can be provided. To allow higher order interpolation also higher derivatives can be set. Whether a slave is able to interpolate and therefore needs this information is provided by the capability attribute `canInterpolateInputs`.

```
fmiStatus fmiSetRealInputDerivatives(fmiComponent c,
                                     const fmiValueReference vr[],
                                     size_t nvr, const fmiInteger order[],
                                     const fmiReal value[]);
```

Sets the n-th time derivative of real input variables. Argument “`vr`” is a vector of value references that define the variables whose derivatives shall be set. The array “`order`” contains the orders of the respective derivative (1 means the first derivative, 0 is not allowed). Argument “`value`” is a vector with the values of the derivatives. “`nvr`” is the dimension of the vectors.
Restrictions on using the function are the same as for the `fmiSetReal` function.

Inputs and their derivatives are set with respect to the beginning of a communication time step.

To allow interpolation/approximation of the real output variables between communication steps (if they are used as inputs for other slaves) the derivatives of the outputs with respect to time can be read. Whether the slave is able to provide the derivatives of outputs is given by the unsigned integer capability flag `MaxOutputDerivativeOrder`. It delivers the maximum order of the output derivative. If the actual order is lower (because the order of integration algorithm is low), the retrieved value is 0.

Example: If the internal polynomial is of order 1 and the master inquires the second derivative of an output, the slave will return zero.

The derivatives can be retrieved by:

```
fmiStatus fmiGetRealOutputDerivatives (fmiComponent c,
                                       const fmiValueReference vr[],
                                       size_t nvr, const fmiInteger order[],
                                       fmiReal value[]);
```

Retrieves the n-th derivative of output values. Argument “`vr`” is a vector of “`nvr`” value references that define the variables whose derivatives shall be retrieved. The array “`order`” contains the order of the respective derivative (1 means the first derivative, 0 is not allowed). Argument “`value`” is a vector with the actual values of the derivatives.
Restrictions on using the function are the same as for the `fmiGetReal` function.

The returned outputs correspond to the current slave time. E. g. after a successful `fmiDoStep(...)` the returned values are related to the end of the communication time step.

This standard supports polynomial interpolation and extrapolation as well as more sophisticated signal extrapolation schemes like rational extrapolation, see the companion document "FunctionalMockupInterface-ImplementationHints.pdf".

4.2.4 Computation

The computation of time steps is controlled by the following function.

```
fmiStatus fmiDoStep(fmiComponent c, fmiReal currentCommunicationPoint,
                    fmiReal    communicationStepSize,
                    fmiBoolean noSetFMUStatePriorToCurrentPoint);
```

The computation of a time step is started.
The argument currentCommunicationPoint is the current communication point of the master (tci). [*Formally this argument is not needed. It is present in order to detect a mismatch between the master and the FMU state of the slave: The `currentCommunicationPoint` and the FMU state of the slaves defined by former `fmiDoStep` or `fmiSetFMUState` calls, have to be consistent with respect to each other. For first call to `fmiDoStep` after `fmiInitializeSlave`, it needs to be consistent with `tStart` given to the latter.*]
Argument `communicationStepSize` is the communication step size. If the master carries out an event iteration the parameter `communicationStepSize` is zero. Argument `noSetFMUStatePriorToCurrentPoint` is `fmiTrue` if fmiSetFMUState will no longer be called for time instants prior to currentCommunicationPoint [*the slave can use this flag to flush a result buffer*].
The function returns:

`fmiOK` - if the communication step was computed successfully until its end.
`fmiDiscard` – if the slave computed successfully only a subinterval of the communication step. The master can call the appropriate `fmiGetXXXStatus` functions to get further information. If possible, the master should retry the simulation with a shorter communication step size.
`fmiError` – the communication step could not be carried out at all. The master can try to repeat the step with other input values and/or a different communication step size.
`fmiPending` – is returned if the slave executes the function asynchronously. That means the slave starts the computation but returns immediately. The master has to call `fmiGetStatus(...,fmiDoStep,...)` to find out, if the slave is done. An alternative is to wait until the callback function `fmiStepFinished` is called by the slave. `fmiCancelStep` can be called to cancel the current computation. It is not allowed to call any other function during a pending `fmiDoStep`.

```
fmiStatus fmiCancelStep(fmiComponent c);
```

Can be called if `fmiDoStep` returned `fmiPending` in order to stop the current asynchronous execution. The master calls this function if for example the co-simulation run is stopped by the user or one of the slaves. Afterwards it is only allowed to call `fmiTerminateSlave`, `fmiResetSlave`, or `fmiFreeSlaveInstance`.

It depends on the capabilities of the slave which parameter constellations and calling sequences are allowed (see 4.3.1)

4.2.5 Retrieving Status Information from the Slave

Status information is retrieved from the slave by the following functions:

```
fmiStatus fmiGetStatus       (fmiComponent c, const fmiStatusKind s,
                              fmiStatus* value);
fmiStatus fmiGetRealStatus   (fmiComponent c, const fmiStatusKind s,
                              fmiReal* value);
fmiStatus fmiGetIntegerStatus(fmiComponent c, const fmiStatusKind s,
                              fmiInteger* value);
fmiStatus fmiGetBooleanStatus(fmiComponent c, const fmiStatusKind s,
                              fmiBoolean* value);
fmiStatus fmiGetStringStatus (fmiComponent c, const fmiStatusKind s,
                              fmiString* value);
```

Informs the master about the actual status of the simulation run. Which status information is to be returned is specified by the argument `fmiStatusKind`. It depends on the capabilities of the slave which status information can be given by the slave (see 4.3.1). If a status is required which cannot be retrieved by the slave it returns `fmiDiscard`.

```
typedef enum {fmiDoStepStatus,
              fmiPendingStatus,
              fmiLastSuccessfulTime,
              fmiTerminated
             } fmiStatusKind;
```

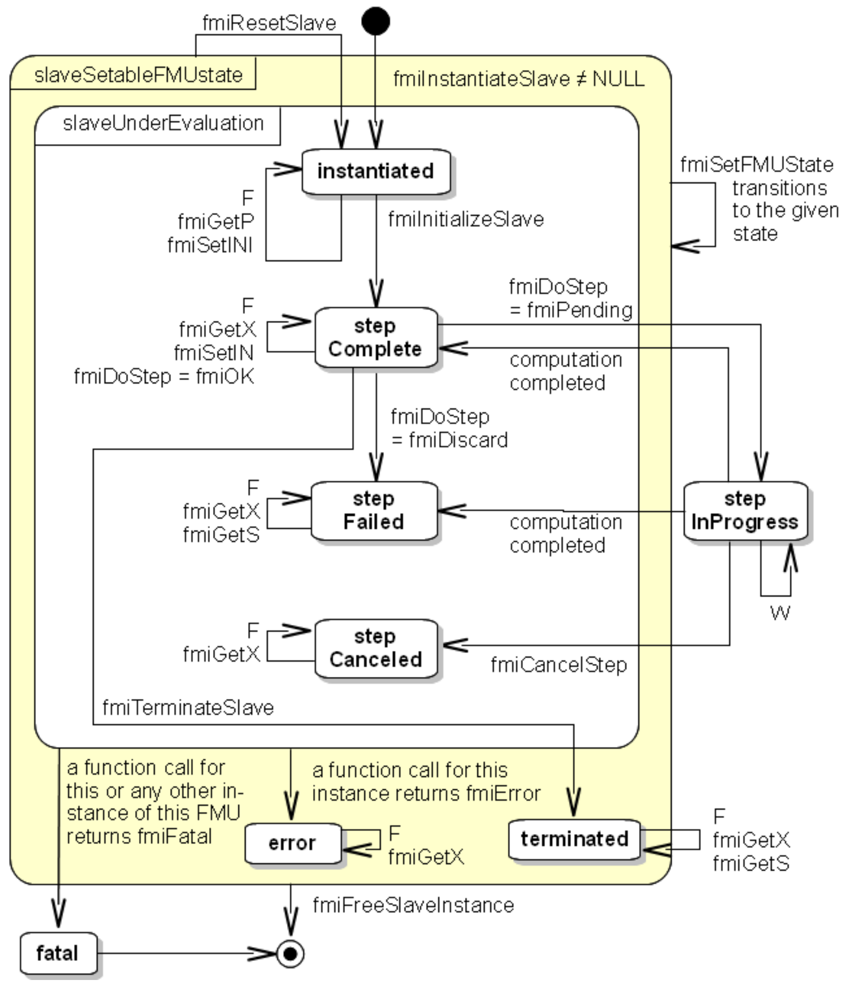
Defines which status is inquired.

The following status information can be retrieved from a slave:

Status	*Type of retrieved value*	*Description*
`fmiDoStepStatus`	`fmiStatus`	Can be called when the `fmiDoStep` function returned `fmiPending`. The function delivers `fmiPending` if the computation is not finished. Otherwise the function returns the result of the asynchronously executed `fmiDoStep` call.
`fmiPendingStatus`	`fmiString`	Can be called when the `fmiDoStep` function returned `fmiPending`. The function delivers a string which informs about the status of the currently running asynchronous `fmiDoStep` computation.
`fmiLastSuccessfulTime`	`fmiReal`	Returns the end time of the last successfully completed communication step. Can be called after `fmiDoStep(...)` returned `fmiDiscard`.
`fmiTerminated`	`fmiBoolean`	Returns true, if the slave wants to terminate the simulation. Can be called after `fmiDoStep(...)` returned `fmiDiscard`. Use `fmiLastSuccessfulTime` to determine the time instant at which the slave terminated. See sample code in section 4.2.7.

4.2.6 State Machine of Calling Sequence from Master to Slave

The following state machine defines the supported calling sequences See 3.2.4 for an explanation of the intended meaning of the state chart.



Abbreviations used in transition labels

F is one of
- *fmiGetTypesPlatform*
- *fmiGetVersion*
- *fmiSetDebugLogging*
- *fmiGetFMUstate*
- *fmiFreeFMUstate*
- *fmiSerializedFMUstateSize*
- *fmiSerializeFMUstate*
- *fmiDeSerializeFMUstate*

S is one of
- *Status, RealStatus, IntegerStatus, BooleanStatus, StringStatus*

X is one of
- *Real, Integer, Boolean, String,*
- *RealOutputDerivatives*
- *PartialDerivatives*
- *DirectionalDerivative*

P is one of
Real, Integer, Boolean, String for a variable with causality = "parameter"

INI is one of
- *Real, Integer, Boolean, String for a variable that has initial = "exact" or "approx"*
- *RealInputDerivatives*

IN is one of
- *Real, Integer, Boolean, String for a variable that has either causality = "input" and variablity = "discrete" or "continuous" or variability = "tunable"*
- *RealInputDerivatives*

W is one of
- *fmiGetTypesPlatform*
- *fmiGetVersion*
- *fmiSetDebugLogging*
- *fmiGetS*

Figure 10: Calling sequence of Co-Simulation C functions in form of an UML 2.0 state machine.

The allowed function calls in the respective states are summarized in the following table (functions marked in "light blue" are only available for "Co-Simulation", the other functions are available both for "Model Exchange" and "Co-Simulation"):

	FMI 2.0 forCo-Simulation								
Function	start, end	instantiated	stepComplete	stepInProgress	stepFailed	stepCanceled	terminated	error	fatal
fmiGetTypesPlatform	x	x	x	x	x	x	x	x	
fmiGetVersion	x	x	x	x	x	x	x	x	
fmiSetDebugLogging		x	x	x	x	x	x	x	
fmiGetReal		5	x		8	7	x	7	
fmiGetInteger		5	x		8	7	x	7	
fmiGetBoolean		5	x		8	7	x	7	
fmiGetString		5	x		8	7	x	7	
fmiSetReal		1	6						
fmiSetInteger		1	6						
fmiSetBoolean		1	6						
fmiSetString		1	6						
fmiGetFMUstate		x	x		8	7	x	7	
fmiSetFMUstate		x	x		x	x	x	x	
fmiFreeFMUstate		x	x		x	x	x	x	
fmiSerializedFMUstateSize		x	x		x	x	x	x	
fmiSerializeFMUstate		x	x		x	x	x	x	
fmiDeSerializeFMUstate		x	x		x	x	x	x	
fmiGetPartialDerivatives			x		8	7	x	7	
fmiGetDirectionalDerivative			x		8	7	x	7	
fmiInstantiateSlave	x								
fmiFreeSlaveInstance		x	x		x	x	x	x	
fmiInitializeSlave		x							
fmiTerminateSlave			x						
fmiResetSlave		x	x		x	x	x	x	
fmiSetRealInputDerivatives		x	x						
fmiGetRealOutputDerivatives			x		8	x	x	7	
fmiDoStep			x						
fmiCancelStep				x					
fmiGetStatus		x	x	x	x		x		
fmiGetRealStatus		x	x	x	x		x		
fmiGetIntegerStatus		x	x	x	x		x		
fmiGetBooleanStatus		x	x	x	x		x		
fmiGetStringStatus		x	x	x	x		x		

x means: call is allowed in the corresponding state
number means: call is allowed if the indicated condition holds:
1 for a variable that has initial = "exact" or "approx"
5 for a variable with causality = "parameter"
6 for a variable with causality = "input" and variability = "discrete" or "continuous",
or variability = "tunable"
7 always, but retrieved values are usable for debugging only
8 always, but if status is other than fmiTerminated, retrieved values are useable for debugging only

4.2.7 Pseudo Code Example

In the following example, the usage of the FMI functions is sketched in order to clarify the typical calling sequence of the functions in a simulation environment. The example is given in a mix of pseudo-code and “C”, in order to keep it small and understandable. We consider two slaves. Both have one continuous real input and one continuous real output which are connected in the following way:

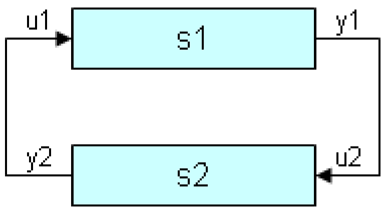


Figure 11: Connection graph of the slaves

We assume no algebraic dependency between input and output of each slave. The code demonstrates the simplest master algorithm as shown in section 4.1.

- Constant communication step size.
- No repeating of communication steps.
- The slaves do not support asynchronous execution of `fmiDoStep`.

The error handling is implemented in a very rudimentary way.

```
//////////////////////////
//Initialization sub-phase

//Set callback functions,
fmiCallbackFunctions cbf;
cbf.logger = loggerFunction;  //logger function
cbf.allocateMemory = calloc;
cbf.freeMemory = free;
cbf.stepFinished = NULL;      //synchronous execution
cbf.componentEnvironment = NULL;

//Instantiate both slaves
fmiComponent s1 = s1_fmiInstantiateSlave("Tool1", GUID1, "", fmiFalse,
      fmiFalse, &cbf, fmiTrue);
fmiComponent s2 = s2_fmiInstantiateSlave("Tool2", GUID2, "", fmiFalse,
      fmiFalse, &cbf, fmiTrue);

if ((s1 == NULL) || (s2 == NULL))
      return FAILURE;

//tStart needs to be between startTime and stopTime from the XML file
tStart = 0;
//tStop needs to be between startTime and stopTime from the XML file
tStop = 10;
//communication step size
h = 0.01;
```

```
// set all variable start values (of "ScalarVariable / <type> / start") and
// set the input values at time = tStart
s1_fmiSetReal/Integer/Boolean/String(s1, ...);
s1_fmiSetReal/Integer/Boolean/String(s2, ...);
```

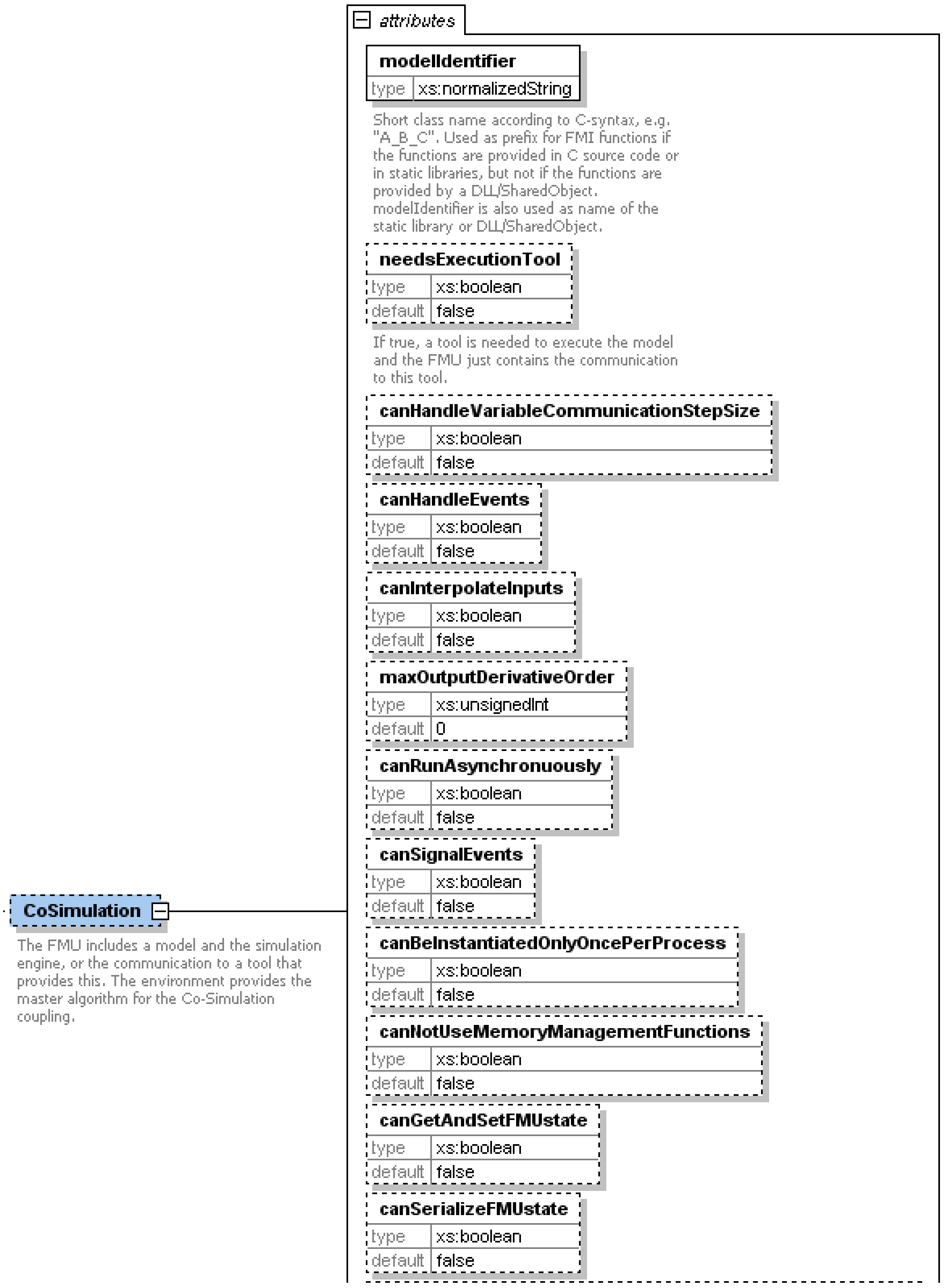
```
//Initialize slaves
status = s1_fmiInitializeSlave(s1, tStart, fmiTrue, tStop);
if (status == fmiOK)
      status = s2_fmiInitializeSlave(s2, tStart, fmiTrue, tStop);

if (status == fmiOK)
{
      //////////////////////////
      //Simulation sub-phase

      tc = tStart; //Current master time

      while ((tc < tStop) && (status == fmiOK))
      {
            //retrieve outputs
            s1_fmiGetReal(s1, ..., 1, &y1);
            s2_fmiGetReal(s2, ..., 1, &y2);
            //set inputs
            s1_fmiSetReal(s1, ..., 1, &y2);
            s2_fmiSetReal(s2, ..., 1, &y1);

            //call slave s1 and check status
            status = s1_fmiDoStep(s1, tc, h, fmiTrue);
            switch (status) {
            case fmiDiscard:
                  fmiGetBooleanStatus(s1, fmiTerminated, &boolVal);
                  if (boolVal == fmiTrue)
                        printf("Slave s1 wants to terminate simulation.");
            case fmiError:
            case fmiFatal:
                  terminateSimulation = true;
                  break;
            }
            if (terminateSimulation)
                  break;

            //call slave s2 and check status as above
            status = s2_fmiDoStep(s2, tc, h, fmiTrue);
            ...

            //increment master time
            tc += h;
      }
      //////////////////////////
      //Shutdown sub-phase
      if ((status != fmiError) && (status != fmiFatal))
      {
            s1_fmiTerminateSlave(s1);
            s2_fmiTerminateSlave(s2);
      }
}
if (status != fmiFatal)
{
      s1_fmiFreeSlaveInstance(s1);
      s2_fmiFreeSlaveInstance(s2);
}
```

4.3 FMI Description Schema

This is defined in section 2.2. Additionally, the co-simulation specific element “Implementation” is defined in the next section.

4.3.1 Co-Simulation FMU (CoSimulation)

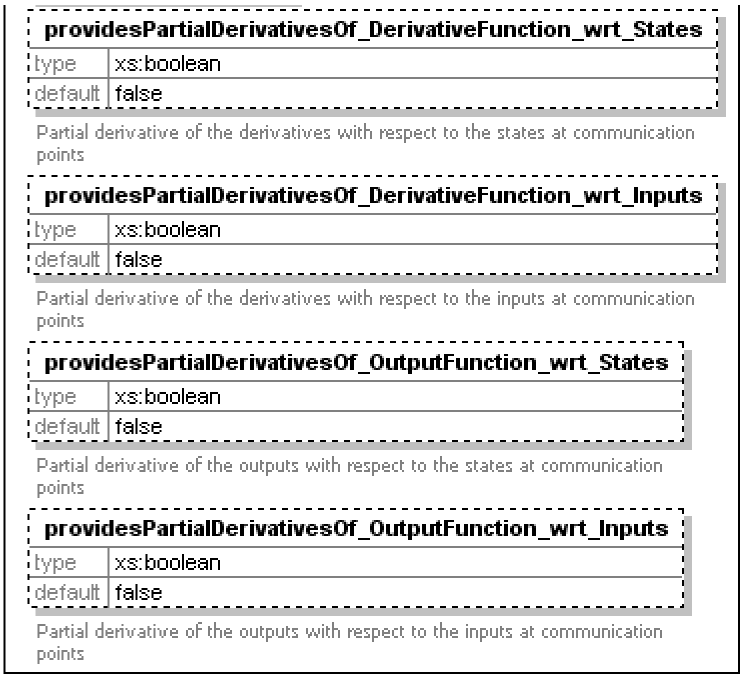
If the XML file defines an FMU for Co-Simulation, element “CoSimulation” must be present. It is defined as:



Short class name according to C-syntax, e.g. "A_B_C". Used as prefix for FMI functions if the functions are provided in C source code or in static libraries, but not if the functions are provided by a DLL/SharedObject. modelIdentifier is also used as name of the static library or DLL/SharedObject.

If true, a tool is needed to execute the model and the FMU just contains the communication to this tool.

The FMU includes a model and the simulation engine, or the communication to a tool that provides this. The environment provides the master algorithm for the Co-Simulation coupling.


providesPartialDerivativesOf_DerivativeFunction_wrt_States
type | xs:boolean
default | false
Partial derivative of the derivatives with respect to the states at communication points
providesPartialDerivativesOf_DerivativeFunction_wrt_Inputs
type | xs:boolean
default | false
Partial derivative of the derivatives with respect to the inputs at communication points
providesPartialDerivativesOf_OutputFunction_wrt_States
type | xs:boolean
default | false
Partial derivative of the outputs with respect to the states at communication points
providesPartialDerivativesOf_OutputFunction_wrt_Inputs
type | xs:boolean
default | false
Partial derivative of the outputs with respect to the inputs at communication points

These attributes have the following meaning (all attributes are optional with exception of "`modelIdentifier`"):

Attribute Name	*Description*
`modelIdentifier`	Short class name according to C syntax, for example "A_B_C". Used as prefix for FMI functions if the functions are provided in C source code or in static libraries, but not if the functions are provided by a DLL/SharedObject. `modelIdentifier` is also used as name of the static library or DLL/SharedObject . See also section 2.1.1.
`needsExecutionTool`	If true, a tool is needed to execute the model. The FMU just contains the communication to this tool (see Figure 7). [*Typically, this information is only utilized for information purposes. For example a co-simulation master can inform the user that a tool has to be available on the computer where the slave is instantiated. The name of the tool can be taken from attribute* `generationTool` *of* `fmiModelDescription.`]
`canHandleVariableCommunicationStepSize`	The slave can handle variable communication step size. The communication step size (parameter `communicationStepSize` of `fmiDoStep(...)`) has not to be constant for each call.

`canHandleEvents`	The slave supports event handling during co-simulation. The communication step size (parameter `communicationStepSize` of `fmiDoStep(...)`) can be zero.
`canInterpolateInputs`	The slave is able to interpolate continuous inputs. Calling of `fmiSetRealInputDerivatives(...)` has an effect for the slave.
`maxOutputDerivativeOrder`	The slave is able to provide derivatives of outputs with maximum order. Calling of `fmiGetRealOutputDerivatives(...)` is allowed up to the order defined by `maxOutputDerivativeOrder`.
`canRunAsynchronuously`	This flag describes the ability to carry out the `fmiDoStep(...)` call asynchronously.
`canSignalEvents`	If a slave is able to provide information about events during a communication step, this flag has to be set to `true`.
`canBeInstantiatedOnlyOncePerProcess`	This flag indicates cases (especially for embedded code), where only one instance per FMU is possible (multiple instantiation is default = `false`; if multiple instances are needed, the FMUs must be instantiated in different processes).
`canNotUseMemoryManagementFunctions`	If `true`, the slave uses its own functions for memory allocation and freeing only. The callback functions `allocateMemory` and `freeMemory` given in `fmiInstantiateSlave` are ignored.
`canGetAndSetFMUstate`	If `true`, the environment can inquire the internal FMU state and can restore it. That is, `fmiGetFMUstate`, `fmiSetFMUstate`, and `fmiFreeFMUstate` are supported by the FMU.
`canSerializeFMUstate`	If `true`, the environment can serialize the internal FMU state, in other words `fmiSerializedFMUstateSize`, `fmiSerializeFMUstate`, `fmiDeSerializeFMUstate` are supported by the FMU. If this is the case, then flag `canGetAndSetFMUstate` must be `true` as well.

`providesPartialDerivativesOf_DerivativeFunction_wrt_States`
`providesPartialDerivativesOf_DerivativeFunction_wrt_Inputs`
`providesPartialDerivativesOf_OutputFunction_wrt_States`	

`providesPartialDerivativesOf_OutputFunction_wrt_Inputs` These capability attributes define whether “`fmiGetPartialDerivatives`” and “`fmiGetDirectionalDerivatives`” can be called to compute the corresponding partial derivatives at communication points. The details are given in the corresponding capability flags of ModelExchange, see section 3.3.1.

The flags have the following default values.

- boolean: false
- unsignedInt: 0

Note, if `needsExecutionTool = true`, then it is required that the original tool is available to be executed in co-simulation mode. If `needsExecutionTool = false`, the slave is completely contained inside the FMU in source code or binary format (DLL/SharedObject).

4.3.2 Example XML Description File

The example below is the same one as shown in section 3.3.2 for a ModelExchange FMU. The only difference is the replacement of element ModelExchange by element CoSimulation (with additional attributes) and the removed local variables which are associated with continuous states and their derivatives. The XML file may have the following content:

```
<?XML version="1.0" encoding="UTF8"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="MyLibrary.SpringMassDamper"
  guid="{8c4e810f-3df3-4a00-8276-176fa3c9f9e0}"
  description="Rotational Spring Mass Damper System"
  version="1.0"
  generationDateAndTime="2011-09-23T16:57:33Z"
  variableNamingConvention="structured">

  <CoSimulation
    modelIdentifier="MyLibrary_SpringMassDamper"
    canHandleVariableCommunicationStepSize="true"
    canHandleEvents="true"
    canInterpolateInputs="true"/>

  <UnitDefinitions>
    <Unit name="rad">
      <BaseUnit rad="1"/>
      <DisplayUnit name="deg" factor="57.2957795130823"/>
    </Unit>
    <Unit name="rad/s">
      <BaseUnit s="-1" rad="1"/>
    </Unit>
    <Unit name="kg.m2">
      <BaseUnit kg="1" m="2"/>
    </Unit>
  </UnitDefinitions>

  <TypeDefinitions>
    <SimpleType name="Modelica.SIunits.Inertia">
      <Real quantity="MomentOfInertia" unit="kg.m2" min="0.0"/> </SimpleType>
    <SimpleType name="Modelica.SIunits.Torque">
```

```
      <Real quantity="Torque" unit="N.m"/> </SimpleType>
    <SimpleType name="Modelica.SIunits.AngularVelocity">
      <Real quantity="AngularVelocity" unit="rad/s"/> </SimpleType>
    <SimpleType name="Modelica.SIunits.Angle">
      <Real quantity="Angle" unit="rad"/> </SimpleType>
  </TypeDefinitions>

  <DefaultExperiment startTime="0.0" stopTime="3.0" tolerance="0.0001"/>

  <ModelVariables>
    <ScalarVariable
      name="inertia1.J"
      valueReference="1073741824"
      description="Moment of load inertia"
      causality="parameter"
      variability="fixed">
      <Real declaredType="Modelica.SIunits.Inertia" start="1"/>
    </ScalarVariable>

    <ScalarVariable
      name="torque.tau"
      valueReference="536870912"
      description="Accelerating torque acting at flange (= -flange.tau)"
      causality="input">
      <Real declaredType="Modelica.SIunits.Torque" />
    </ScalarVariable>

    <ScalarVariable
      name="inertia1.phi"
      valueReference="805306368"
      description="Absolute rotation angle of component"
      causality="output">
      <Real declaredType="Modelica.SIunits.Angle" />
    </ScalarVariable>

    <ScalarVariable
      name="inertia1.w"
      valueReference="805306369"
      description="Absolute angular velocity of component (= der(phi))"
      causality="output">
      <Real declaredType="Modelica.SIunits.AngularVelocity" />
    </ScalarVariable>
</ModelVariables>

  <ModelStructure>
    <Inputs>
      <Input name="torque.tau"/>
    </Inputs>
    <Outputs>
      <Output name="inertia1.phi" />
      <Output name="inertia1.w" />
    </Outputs>
  </ModelStructure>
</fmiModelDescription>
```

5. Literature

AMESim: www.lmsintl.com/

Andersson C., Akesson J., Führer C., Gäfvert M. (2011): **Import and Export of Functional Mock-up Units in JModelica.org**. Modelica Conference 2011. http://www.ep.liu.se/ecp/063/036/ecp11063036.pdf

Atego Ace: http://www.atego.com/products/atego-ace/

AUTOSAR: www.autosar.org.

Bastian J., Clauß C., Wolf S., Schneider P. (2011): **Master for Co-Simulation Using FMI**. 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/014/ecp11063014.pdf

Blochwitz T., Kurzbach G., Neidhold T. (2008): **An External Model Interface for Modelica**. 6-th International Modelica Conference, Bielefeld 2008. www.modelica.org/events/modelica2008/Proceedings/sessions/session5f.pdf

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. (2011): **The Functional Mockup Interface for Tool independent Exchange of Simulation Models**. 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf

Blochwitz T., Otter M., Akesson J., Arnold M., Clauß C., Elmqvist H., Friedrich M., Junghanns A., Mauss J,, Neumerkel D., Olsson H., Viel A. (2012): **Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models**. 9th International Modelica Conference, Munich, 2012.

Brembeck J., Otter M., Zimmer D. (2011): **Nonlinear Observers based on the Functional Mockup Interface with Applications to Electric Vehicles**. 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/053/ecp11063053.pdf

Dymola: www.dymola.com.

Kübler R., Schiehlen, W. (2000): **Two methods of simulator coupling**. Mathematical and Computer Modeling of Dynamical Systems **6** pp. 93-113.

Modelica (2010): **Modelica, A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, Version 3.2**. March 24, 2010. www.modelica.org/documents/ModelicaSpec32.pdf

MODELISAR Glossary (2009): **MODELISAR WP2 Glossary and Abbreviations**. Version 1.0, June 9, 2009.

Noll, C., Blochwitz T., Neidhold, T. Kehrer, C. (2011): **Implementation of Modelisar Functional Mock-up Interfaces in SimulationX.** 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/037/ecp11063037.pdf

Otter M. (1999): **Objektorientierte Modellierung Physikalischer Systeme, Teil 4**. at – Automatisierungstechnik, April, pp. A13-A16.

Otter M., Elmqvist H. (1995): **The DSblock model interface for exchanging model components**. Proceedings of EUROSIM '95 Simulation Congress, pp. 505-510, Sept. 11-15, Vienna.

Yongqi Sun Y., Vogel S., Steuer H. (2011): **Combining Advantages of Specialized Simulation Tools and Modelica Models using Functional Mock-up Interface (FMI).** 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/055/ecp11063055.pdf

Schubert, C., Neidhold, T, Kunze G. (2011): **Experiences with the new FMI Standard, Selected Applications at Dresden University.** 8th Internataional Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/038/ecp11063038.pdf

Thiele, B., Henriksson, D. (2011): **Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development**. 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/054/ecp11063054.pdf

Silver: www.qtronic.de/de/silver.html

Simpack: www.simpack.com

SimulationX: www.simulationx.com.

XML: www.w3.org/XML, en.wikipedia.org/wiki/XML

Appendix A FMI Revision History

This appendix describes the history of the FMI design and its contributors. The current version of this document is available from http://www.functional-mockup-interface.org/fmi.html.

The Functional Mock-up Interface development was initiated and organized by Daimler AG (from Bernd Relovsky and others) as subproject inside the ITEA2 MODELISAR project.

The development of versions 1.0 and 2.0 was performed within WP200 of MODELISAR, organized by the WP200 work package leader Dietmar Neumerkel from Daimler.

A.1 Version 1.0 – FMI for Model Exchange

Version 1.0 of FMI for Model Exchange was released on Jan. 26, 2010.

The subgroup "FMI for Model Exchange" was headed by Martin Otter (DLR-RM). The essential part of the design of this version was performed by (alphabetical list):

Torsten Blochwitz, ITI, Germany
Hilding Elmqvist, Dassault Systèmes (Dynasim), Sweden
Andreas Junghanns, QTronic, Germany
Jakob Mauss, QTronic, Germany
Hans Olsson, Dassault Systèmes (Dynasim), Sweden
Martin Otter, DLR-RM, Germany.

This version was evaluated with prototypes implemented for (alphabetical list):

Dymola by Peter Nilsson, Dan Henriksson, Carl Fredrik Abelson, and Sven Erik Mattson, Dassault Systèmes (Dynasim),
JModelica.org by Tove Bergdahl, Modelon AB,
Silver by Andreas Junghanns, and Jakob Mauss, QTronic.

These prototypes have been used to refine the design of "FMI for Model Exchange".

The following MODELISAR partners participated at FMI design meetings and contributed to the discussion (alphabetical list):

Ingrid Bausch-Gall, Bausch-Gall GmbH, Munich, Germany
Torsten Blochwitz, ITI GmbH, Dresden, Germany
Alex Eichberger, SIMPACK AG, Gilching, Germany
Hilding Elmqvist, Dassault Systèmes (Dynasim), Lund, Sweden
Andreas Junghanns, QTronic GmbH, Berlin, Germany
Rainer Keppler, SIMPACK AG, Gilching, Germany
Gerd Kurzbach, ITI GmbH, Dresden, Germany
Carsten Kübler, TWT, Germany
Jakob Mauss, QTronic GmbH, Berlin, Germany
Johannes Mezger, TWT, Germany
Thomas Neidhold, ITI GmbH, Dresden, Germany
Dietmar Neumerkel, Daimler AG, Stuttgart, Germany
Peter Nilsson, Dassault Systèmes (Dynasim), Lund, Sweden
Hans Olsson, Dassault Systèmes (Dynasim), Lund, Sweden
Martin Otter, German Aerospace Center (DLR), Oberpfaffenhofen, Germany
Antoine Viel, LMS International (Imagine), Roanne, France

Daniel Weil, Dassault Systèmes, Grenoble, France

The following people outside of the MODELISAR consortium contributed with comments:

Johan Akesson, Lund University, Lund, Sweden
Joel Andersson, KU Leuven, The Netherlands
Roberto Parrotto, Politecnico di Milano, Italy

A.2 Version 1.0 – FMI for Co-Simulation

Version 1.0 of FMI for Co-Simulation was released on Oct. 10, 2010.

FMI for Co-Simulation was developed in three subgroups: “Solver Coupling” headed by Martin Arnold (University Halle) and Torsten Blochwitz (ITI), “Tool Coupling” headed by Jörg-Volker Peetz (Fraunhofer SCAI), and “Control Logic” headed by Manuel Monteiro (Atego). The essential part of the design of this version was performed by (alphabetical list):

Martin Arnold, University Halle, Germany
Constanze Bausch, Atego Systems GmbH, Wolfsburg, Germany
Torsten Blochwitz, ITI GmbH, Dresden, Germany
Christoph Clauß, Fraunhofer IIS EAS, Dresden, Germany
Manuel Monteiro, Atego Systems GmbH, Wolfsburg, Germany
Thomas Neidhold, ITI GmbH, Dresden, Germany
Jörg-Volker Peetz, Fraunhofer SCAI, St. Augustin, Germany
Susann Wolf, Fraunhofer IIS EAS, Dresden, Germany

This version was evaluated with prototypes implemented for (alphabetical list):

SimulationX by Torsten Blochwitz and Thomas Neidhold (ITI GmbH),
Master algorithms by Christoph Clauß (Fraunhofer IIS EAS)

The following MODELISAR partners participated at FMI design meetings and contributed to the discussion (alphabetical list):

Martin Arnold, University Halle, Germany
Jens Bastian, Fraunhofer IIS EAS, Dresden, Germany
Constanze Bausch, Atego Systems GmbH, Wolfsburg, Germany
Torsten Blochwitz, ITI GmbH, Dresden, Germany
Christoph Clauß, Fraunhofer IIS EAS, Dresden, Germany
Manuel Monteiro, Atego Systems GmbH, Wolfsburg, Germany
Thomas Neidhold, ITI GmbH, Dresden, Germany
Dietmar Neumerkel, Daimler AG, Böblingen, Germany
Martin Otter, DLR, Oberpfaffenhofen, Germany
Jörg-Volker Peetz, Fraunhofer SCAI, St. Augustin, Germany
Tom Schierz, University Halle, Germany
Klaus Wolf, Fraunhofer SCAI, St. Augustin, Germany

A.3 Version 2.0 – FMI for Model Exchange and Co-Simulation

FMI 2.0 for Model Exchange and Co-Simulation was released on xxx, 2012.

A.3.1 Overview

This section gives an overview about the changes with respect to versions 1.0 for Model Exchange and 1.0 for Co-Simulation:

- FMI 2.0 is not backwards compatible to FMI 1.0.
- The documents, schema and header files for Model Exchange and for Co-Simulation have been merged. Due to the merging, some conflicts had to be resolved leading to some non-backwards compatible changes with respect to FMI 1.0.
- Parameters can be declared to be "tunable" in the FMU, in other words during simulation these parameters can be changed (if supported by the simulation environment).
- When enabling logging, log categories to be logged can be defined, so that the FMU needs to only generate logs of the defined categories (in FMI 1.0, logs had to be generated for all log categories and they had to be filtered afterwards). Log categories that are supported by an FMU can be optionally defined in the XML file so that a simulation environment can provide them to the user for selection.
- FMI function names are no longer prefixed with the "modelIdentifier" if used in a DLL/sharedObject. As a result, FMUs that need a tool, can use a generic communication DLL, and the loading of DLLs is no longer FMU dependent.
- The termination of every global event iteration over connected FMUs must be reported by a new function call.
- The unit definitions have been improved: The tool-specific unit-name can optionally be expressed as function of the 7 SI base units and the SI derived unit "rad". It is then possible to check units when FMUs are connected together (without standardizing unit names), or to convert variable values that are provided in different units (for the same physical quantity).
- Enumerations have an arbitrary (but unique) mapping to integers (in FMI 1.0, the mapping was automatically to 1,2,3,...).
- Explicit alias/antiAlias variable definitions have been removed, to simplify the interface: If variables of the same base type (like `fmiReal`) have the same `valueReference`, they have identical values. A simulation environment may ignore this completely (this was not possible in FMI 1.0), or can utilize this information to more efficiently store results on file.
- When instantiating an FMU, the absolute path to the FMU resource directory is now reported also in Model Exchange, in order that the FMU can read all of its resources (for example maps, tables, ...) independently of the "current directory" of the simulation environment where the FMU is used.
- An ordering is defined for input, output, and state variables in the XML file of an FMU, in order for this order to be defined in the FMU, and not (arbitrarily) selected by the simulation environment. This is essential, for example when linearizing an FMU, or when providing "sparsity" information (see below).
- Several optional features have been added:
  - The complete FMU state can be saved, restored, and serialized to a byte vector (that can be stored on file). As a result, a simulation (both for Model Exchange and for Co-Simulation) can be restarted from a saved FMU state. Rejecting steps for variable step-size Co-Simulation master algorithms is now performed with this feature (instead of the less powerful method of FMI 1.0).
  - The dependency of state derivatives and of output variables from inputs and states can be defined in the XML file, in other words the sparsity pattern for Jacobians can be defined. This

allows simulating stiff FMUs with many states (> 1000 states) since sparse matrix methods can be utilized in the numerical integration method. Furthermore, it can be stated whether this dependency is linear (this allows to transform nonlinear algebraic equation systems into linear equation systems when connecting FMUs).

- The partial derivatives of the state derivatives and the outputs with respect to the states and the inputs can be computed by a new function. If the exported FMU performs this computation analytically, then all numerical algorithms based on this Jacobian (for example the numerical integration method or nonlinear algebraic solvers) are more efficient and more reliable.
- Directional derivatives can be computed for state derivatives and outputs. This is useful when connecting FMUs and the partial derivatives of the connected FMU shall be computed.
- Every scalar variable definition can have an additional “annotation” data structure that is arbitrary (“any” element in XML). A tool vendor can store tool-dependent information here (that other tools can ignore), for example to store the graphical layout of parameter menus. The `VendorAnnotations` element was also generalized from (name, value) pairs to any XML data structure.

- Many smaller improvements have been included, due to the experience in using FMI 1.0 (for example the causality/variability attributes have been changed and more clearly defined, the `fmiModelFunctions.h` header has been split into two header files (one for the function signature, and one for the function names), in order that the header files can be directly used both for DLLs and for source code distribution).

A.3.2 Main changes

This section gives the details about the changes with respect to versions 1.0 for Model Exchange and 1.0 for Co-Simulation:

In this version, the documents of version 1.0 for Model Exchange and for Co-Simulation have been merged and several new features have been added.

The following changes in FMI 2.0 are <u>not backwards compatible</u> due to the merging:

- File fmiModelTypes.h (in FMI for Model Exchange) has been renamed to fmiTypesPlatform.h (the file name used in FMI for Co-Simulation).
- File fmiModelFunctions.h (in FMI for Model Exchange) has been renamed to fmiFunctions.h (the file name used in FMI for Co-Simulation), and the function prototypes in this header files have been merged from “Model Exchange” and from “Co-Simulation”). Additionally, a new header files has been added, `fmiFunctionTypes.h` that contains a definition of the function signatures. This header file is also used in `fmiFunctions.h` (so the signature is not duplicated). The benefit is that `fmiFunctionTypes.h` can be directly used when loading a DLL/sharedObject (in FMI 1.0, the tool providers had to provide this header file by themselves).
- Fixing ticket #47:
  In FMI 1.0 for Model Exchange the fmiModelDescription.version was defined as string, whereas in Co-Simulation it was defined as integer. This has been changed, so that version is a string.

The following <u>backwards compatible</u> improvements have been made in FMI 2.0:

- The FMI 1.0 documents have been merged (for example all common definitions have been placed in the new chapter 2).

The following not backwards compatible improvements have been made in FMI 2.0:

- Element "fmiModelDescription.Implementation" in the model description schema file as been replaced by a different structure where one hierarchical level is removed. There are now 2 elements directly under fmiModelDescription: "ModelExchange" and "CoSimulation".
  File "fmiImplementation.xsd" has been removed.
  New capability flags have been introduced both for `ModelExchange` and for `CoSimulation`, such as `canGetAndSetFMUstate`, `canSerializeFMUstate` etc.

  Attribute `modelIdentifier` has been moved from an `fmiModelDescription` attribute to an attribute in `ModelExchange` and `CoSimulation`. This allows providing different identifiers, and then an FMU may contain both distribution types with different DLL names (which correspond to the `modelIdentifier` names).
  A new attribute `needsExecutionTool` has been introduced both in `ModelExchange` and in `CoSimulation` in order to define whether a tool is needed to execute the FMU. The previous elements in `CoSimulation_Tool` have been removed.
- In order to clearly define when a global event iteration over several connected FMU is terminated, the new function `fmiCompletedEventIteration(..)` must be called once this appeared.
- Fixing ticket #9:
  A new element `LogCategory` was introduced in `fmiModelDescription`. This is an unordered set of strings representing the possible values of the log categories of the FMU (for example `logEvent`).
  Function `fmiSetDebugLogging` has two new arguments to define the categories (from `LogCategory`) to be used in log messages.
- Fixing ticket #33:
  The `causality` and `variability` attributes of a `ScalarVariable` have not been fully clear. This has been fixed by changing the enumeration values of `variability` from "`constant, parameter, discrete, continuous`" to "`constant, fixed, tunable, discrete, continuous`" and `causality` from "`input output internal none`" to "`parameter, input, output, local`".
  This change includes now also the support of parameters that can be tuned (changed) during simulation.
- Fixing ticket #35:
  In order to simplify implementation (for example no longer an "element event handler" needed in SAX XML parser), the only location where data is not defined by attributes, is changed to an attribute definition: Element `DirectDependency` in `ScalarVariable` is removed. The same information can now be obtained from the `InputDependency` attribute inside `fmiModelDescription.ModelStructure.Outputs`.
- Fixing ticket #37:
  The new status flag `fmiTerminate` is added to the Co-Simulation definition. This allows a slave to terminate the simulation run before the stop time is reached without triggering an error.
- Fixing ticket #39:
  Wong example in the previous section 2.10 of Co-Simulation has been fixed.
- Fixing ticket #41:
  New types introduced in fmiTypesPlatform.h :
  `fmiComponentEnvironment, fmiFMUstate, fmiByte`.
  Struct `fmiCallbackFunctions` gets a new last argument:
  `fmiComponentEnvironment componentEnvironment`
  The first argument of function `logger` is changed from type `fmiComponent` to

`fmiComponentEnvironment`.
By these changes, a pointer to a data structure from the simulation environment is passed to the `logger` and allows the `logger`, for example to transform a `valueReference` in the log message to a variable name.

- Fixing ticket #42:
  Enumerations defined in fmiType.xsd are now defined with (name, value) pairs. An enumeration value must be unique within the same enumeration (to have a bijective mapping between enumeration names and values, in order that results can optionally be presented with names and not with values).
  Furthermore, the `min/max` values of element `Enumeration` in `TypeDefinition` have been removed, because they are meaningless.

- Fixing ticket #43:
  The previous header file fmiFunctions.h is split into 2 header files, fmiFunctionTypes.h and fmiFunctions.h, in order to simplify the dynamic loading of an FMU (the typedefs of the function prototypes defined in fmiFunctionTypes.h can be used to type case the function pointers of the dynamic loading).

- Fixing ticket #45:
  Contrary to the ticket proposal, no new function `fmiResetModel` is added. Instead 6 new functions are added to get and set the internal FMU state via a pointer and to serialize and deserialize an FMU state via a byte vector provided by the environment. For details, see section 2.1.7. This feature allows, for example to support more sophisticated co-simulation master algorithms which require the repetition of communication steps. Additionally, two capability flags have been added (`canGetAndSetFMUstate`, `canSerializeFMUstate`) in order to define whether these features are supported by the FMU.

- Fixing ticket #46:
  The unit definitions have been enhanced by providing an optional mapping to the 7 SI base units and the SI derived unit “rad”, in order for a tool to be able to check whether a signal provided to the FMU or inquired by the FMU has the expected unit.

- Fixing ticket #48:
  The definition of `fmiBoolean` in `fmiTypesPlatform.h` for “standard32” was changed from `char` to `int`. The main reason is to avoid unnecessary casting of Boolean types when exporting an FMU from a Modelica environment or when importing it into a Modelica environment.
  The current definition of `char` for a Boolean was not meaningful, since, for example for embedded code generation usually Booleans are packed on integers and `char` for one Boolean would also not be used. It is planned to add more supported data types to an FMU in the future, which should then also include support for packed Booleans.

- Fixing ticket #49:
  Argument `fmiComponent` in function pointer `stepFinished` was changed to `fmiComponentEnvironment` (when `stepFinished` is called from a co-simulation slave and provides `fmiComponentEnvironment`, then this data structure provided by the environment can provide environment specific data to efficiently identify the slave that called the function).

- Fixing ticket #54:
  In section 2.3 it is now stated, that the FMU must include all referenced resources. This means especially that for Microsoft VisualStudio the option “MT” has to be used when constructing a DLL in order to include the run-time environment of VisualStudio in the DLL.

- Function `fmiInitialize` was renamed to `fmiInitializeModel`.

- Function `stepEvent` in `struct fmiCallbackFunctions` had different locations in the FMI documentation and in the header file. This inconsistency has been corrected by using the location in the header file (at the end of the `struct`).
- The `struct fmiCallbackFunctions` is provided as a pointer to the `struct` when instantiating an FMU, and not as the `struct` itself. This simplifies the importing of an FMU into a Modelica environment.
- Defined how to utilize the min/max attributes for fmiSetReal, fmiSetInteger, fmiGetReal, fmiGetInteger calls.
- Attributes “numberOfScalarVariables”, “numberOfContinuousStates”, “numberOfInputs”, “numberOfOutputs” available in FMI 1.0 have been removed, because they can be deduced from the remaining xml file (so in FMI 2.0 this would have been redundant information).

A.3.3 Contributors

The development group for this version was headed by Torsten Blochwitz (ITI) and Martin Otter (DLR). The essential part of the design of this version was performed by (alphabetical list):

Johan Akesson, Modelon, Sweden
Martin Arnold, University Halle, Germany
Torsten Blochwitz, ITI, Germany
Christoph Clauss, Fraunhofer IIS EAS, Germany
Hilding Elmqvist, Dassault Systèmes AB, Sweden
Markus Friedrich, SIMPACK AG, Germany
Andreas Junghanns, QTronic, Germany
Jakob Mauss, QTronic, Germany
Dietmar Neumerkel, Daimler AG, Germany
Hans Olsson, Dassault Systèmes AB, Sweden
Martin Otter, DLR RMC-SR, Germany
Antoine Viel, LMS International, Belgium

The FMI 2.0 document was edited by Martin Otter (DLR), Torsten Blochwitz (ITI), and Martin Arnold (Uni Halle). The State Machines and tables for the Calling Sequences for Model Exchange and Co-Simulation are from Jakob Mauss (QTronic).

This version was evaluated with prototypes implemented for (alphabetical list):

XXXX

These prototypes have been used to refine the design of “FMI 2.0 for Model Exchange and Co-Simulation”.

The following MODELISAR partners participated at FMI 2.0 design meetings and contributed to the discussion (alphabetical list):

Martin Arnold, University Halle, Germany
Bernd Relovsky, Daimler AG, Germany
Mongi Ben-Gaid, IFP, France
Torsten Blochwitz, ITI, Germany
Christoph Clauss, Fraunhofer IIS EAS, Germany
Alex Eichberger, SIMPACK AG, Germany
Hilding Elmqvist, Dassault Systèmes AB, Sweden
Markus Friedrich, SIMPACK AG, Germany
Andreas Junghanns, QTronic, Germany

Kristin Majetta, Fraunhofer IIS EAS, Germany
Sven Erik Mattsson, Dassault Systèmes AB, Sweden
Jakob Mauss, QTronic, Germany
Dietmar Neumerkel, Daimler AG, Germany
Peter Nilsson, Dassault Systèmes AB, Sweden
Hans Olsson, Dassault Systèmes AB, Sweden
Martin Otter, DLR RMC-SR, Germany
Tom Schierz, University Halle, Germany
Bernhard Thiele, DLR RMC-SR, Germany
Antoine Viel, LMS International, Belgium

The following OPENPROD partners participated at FMI 2.0 design meetings and contributed to the discussion (alphabetical list):

Johan Akesson, Modelon, Sweden
Peter Fritzson, PELAB, Sweden
Petter Lindholm, Modelon, Sweden

The following people contributed with comments (alphabetical list):

Peter Aaronsson, MathCore, Sweden
Bernhard Bachmann, University of Bielefeld, Germany
Iakov Nakhimovski, Modelon, Sweden
Andreas Pfeiffer, DLR RMC-SR

Appendix B Glossary

This glossary is a subset of (*MODELISAR Glossary, 2009*) with some extensions specific to this document.

Term	Description
algorithm	A formal recipe for solving a specific type of problem.
application programming interface (API)	A set of functions, procedures, methods or classes together with type conventions/declarations (for example C header files) that an operating system, library or service provides to support requests made by computer programs.
AUTOSAR	AUTomotive Open System Architecture (www.autosar.org). Evolving standard of the automotive industry to define the implementation of embedded systems in vehicles including communication mechanisms. An important part is the standardization of C functions and macros to communicate between software components. AUTOSAR is targeted to built on top of the real-time operating system OSEK (www.osek-vdx.org, de.wikipedia.org/wiki/OSEK). The use of the AUTOSAR standard requires AUTOSAR membership.
communication points	Time grid for data exchange between master and slaves in a co-simulation environment (also known as “sampling points” or “synchronization points”).
communication step size	Distance between two subsequent *communication points* (also known as “sampling rate” or “macro step size”).
co-simulation	Coupling (in other words dynamic mutualexchange and utilization of intermediate results) of several *simulation programs* including their numerical solvers in order to simulate a system consisting of several subsystems.
co-simulation platform	Software used for coupling several *simulation programs* for *co-simulation.*
ECU	Electronic Control Unit (Microprocessor that is used to control a sub-system in a vehicle).
event	The time instant at which the integration is halted and variables may change their values discontinuously. Between event instants, all variables are continuous.
FMI	Functional Mock-up Interface: Interface of a functional mock-up in form of a model. In analogy to the term digital mock-up (see *mock-up*), functional mock-up describes a computer-based representation of the functional behaviour of a system for all kinds of analyses.
FME	Functional Mock-up Environment: In the general scheme of a *simulation program* FMUE is the part, which is responsible for all control activities and computations of the simulation, including data exchange between coupled simulation programs. It does include neither a user interface nor a logic for a user interaction.
FMI for Co-Simulation	Functional Mock-up Interface for Co-Simulation: One of the MODELISAR *functional mock-up interfaces.* It connects the *master solver* component with one or more *slave solvers*.
FMI for Model Exchange	Functional Mock-up Interface for Model Exchange: One of the MODELISAR *functional mock-up interfaces.* It consists of the *model description interface* and the *model execution interface*. It connects the *external model* component with the *solver* component.

Term	**Description**
FMTC	Functional Mock-up Trust Center: As defined in the MODELISAR framework, FMTC describes a closed system providing *model* and *simulation* access to authenticated users and functional mock-up authorities through dedicated cryptographic interfaces.
FMU	Functional Mock-up Unit: A “model class” from which one or more “model instances” can be instantiated for simulation. An FMU is stored in one zip file as defined in section 2.3 consisting basically of one XML file (see section 0) that defines the model variables and a set of C functions (see section 2.1), in source or binary form, to execute the model equations or the simulator slave. In case of tool execution, additionally, the original simulator is required to perform the co-simulation (compare section 4.3.1)
gateway	A link between two computer programs allowing them to share information and bypass certain protocols on a host computer.
integration algorithm	The numerical algorithm to solve differential equations.
integrator	A *software component*, which implements an *integration algorithm*.
interface	An abstraction of a *software component* that describes its behavior without dealing with the internal implementation. *Software components* communicate with each other via interfaces.
master/slave	A method of communication, where one device or process has unidirectional control over one or more other devices. Once a master/slave relationship between devices or processes is established, the direction of control is always from the master to the slaves. In some systems a master is elected from a group of eligible devices, with the other devices acting in the role of slaves.
mock-up	A full-sized structural, but not necessarily functional model built accurately to scale, used chiefly for study, testing, or display. In the context of computer aided design (CAD), a digital mock-up (DMU) means a computer-based representation of the product geometry with its parts, usually in 3-D, for all kinds of geometrical and mechanical analyses.
model	A model is a mathematical or logical representation of a system of entities, phenomena, or processes. Basically a model is a simplified abstract view of the complex reality. It can be used to compute its expected behavior under specified conditions.
model description file	The model description file is an XML file, which supplies a description of all properties of a *model* (for example input/output variables).
model description interface	An interface description to write or retrieve information from the *model description file*.
Model Description Schema	An *XML* schema that defines how all relevant, non-executable, information about a “model class” (*FMU)* is stored in a text file in *XML* format. Most important, data for every variable is defined (variable name, handle, data type, variability, unit, etc.), see section 0.
numerical solver	see *solver*
output points	Tool internal time grid for saving output data to file (in some tools also known as “*communication points*” – but this term is used in a different way in FMI for Co-Simulation, see above).
output step size	Distance between two subsequent *output points*.

Term	Description
parameter	A quantity within a *model*, which remains constant during *simulation (fixed parameter) or may change at event instances (tunable parameter).*Examples are a mass, stiffness, etc.
slave	see *master/slave*
simulation	Compute the behavior of one or several *models* under specified conditions. (see also *co-simulation*)
simulation model	see *model*
simulation program	Software to develop and/or solve simulation *models*. The software includes a *solver*, may include a user interface and methods for post processing (see also: *simulation tool*, *simulation environment*). Examples of simulation programs are: AMESim, Dymola, SIMPACK, SimulationX, SIMULINK.
simulation tool	see *simulation program*
simulator	A simulator can include one or more *simulation programs*, which solve a common simulation task.
solver	*Software component,* which includes algorithms to solve *model*s, for example *integration algorithms* and *event handling* methods.
state	The “continuous states” of a model are all variables that appear differentiated in the model and are independent from each other. The “discrete states” of a model are time-discrete variables that have two values in a model: The value of the variable from the previous *event* instant, and the value of the variable at the actual event instant.
state event	*Event* that is defined by the time instant where the domain $z > 0$ of an event indicator variable z is changed to $z \le 0$, or vice versa. This definition is slightly different from the usual standard definition of state events: “$z(t)*z(t_{i-1}) \le 0$” which has the severe drawback that the value of the event indicator at the previous event instant, $z(t_{i-1}) \neq 0$, must be non-zero and this condition cannot be guaranteed. The often used term “zero crossing function” for z is misleading (and is therefore not used in this document), since a state event is defined by a change of a domain and not by a zero crossing of a variable.
step event	*Event* that might occur at a completed integrator step. Since this event type is not defined by a precise time or condition, it is usually not defined by a user. A program may use it, for example to dynamically switch between different states. A step event is handled much more efficiently than a *state event*, because the event is just triggered after performing a check at a completed integrator step, whereas a search procedure is needed for a state event.
time event	*Event* that is defined by a predefined time instant. Since the time instant is known in advance, the integrator can select its step size so that the event point is directly reached. Therefore, this event can be handled efficiently.
user interface	The part of the simulation program that gives the user control over the simulation and allows watching results.
XML	eXtensible Markup Language (www.w3.org/XML, en.wikipedia.org/wiki/XML) – An open standard to store information in text files in a structured form.