Functional Mockup Interface (FMI) A General Standard for Model Exchange and Simulator Coupling

Adeel Asghar and Willi Braun
Linköping University
University of Applied Sciene Bielefeld

2017-02-07

LINKÖPING UNIVERSITY

FH Bielefeld
University of
Applied Sciences

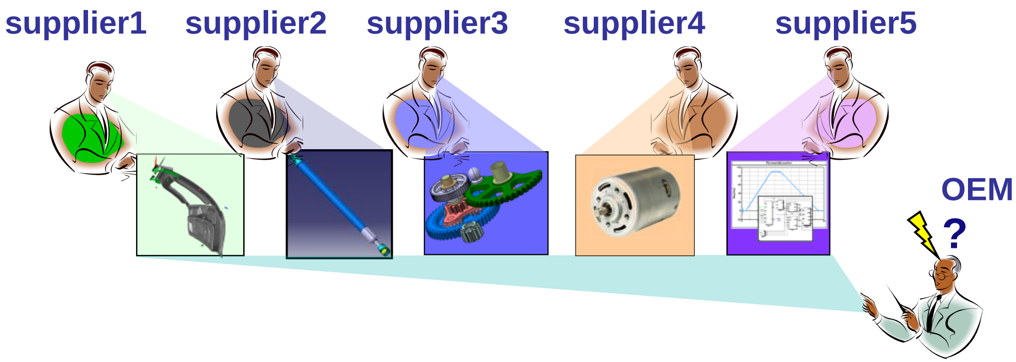
MODELICA

FMI – Motivation 1

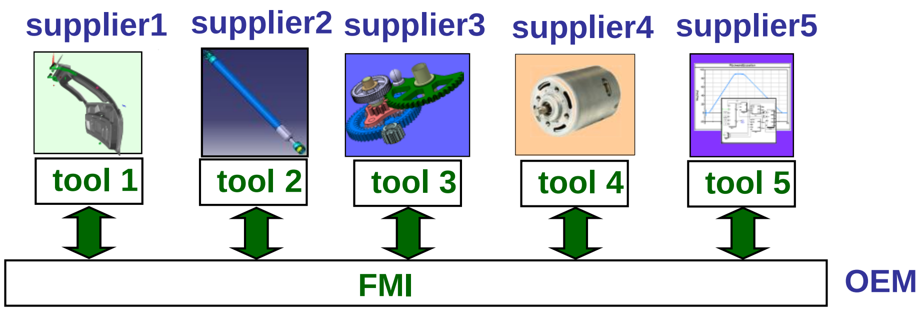
- Need to SOLVE **large integrated** modeling and simulation engineering **problems**
- Hundreds of simulation tools, different model formats
- Exchange dynamic models between different tools and define tool coupling for dynamic system simulation environments.
- *Two main approaches:*
  - 1. **Export** models from some tools,
    import into other tools for **simulation**
  - 2. **Co-simulation** of models in different tools
- Implementation Package Format: **Functional Mockup Unit (FMU)**
- Solution: Functional Mockup Interface (FMI) standard **www.fmi-standard.org**

FMI – Motivation 2

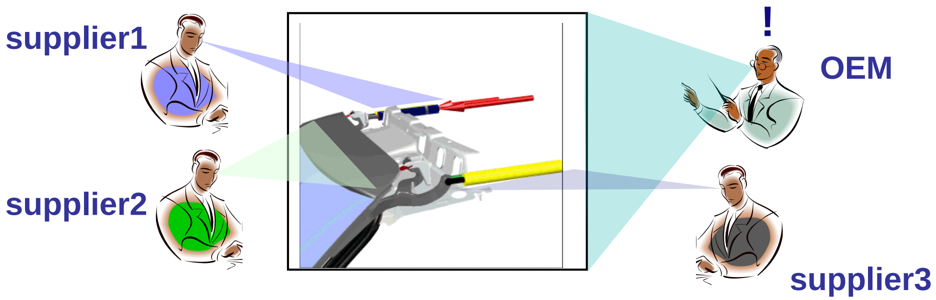
- Problems / Needs
  - Component development by supplier
  - Integration by OEM
  - Many different simulation tools



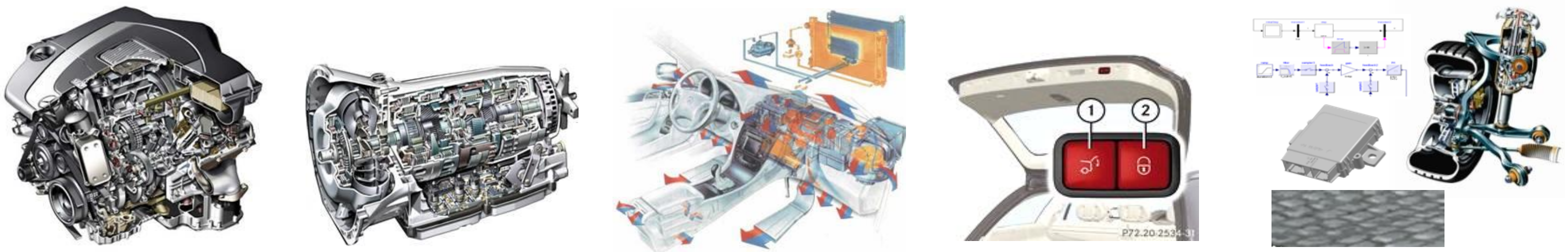
- Solution
  - Reuse of supplier models by OEM:
    - DLL (model import) and/or
    - Tool coupling (co-simulation)



- Added Value
  - Early validation of design
  - Increased process efficiency and quality



Functional Mock-up Interface (FMI) – Overview



Engine with ECU | **Gearbox with ECU** | **Thermal systems** | **Automated cargo door** | **Chassis components, roadway, ECU (e.g. ESP)**

functional mockup interface for model exchange and tool coupling

courtesy Daimler

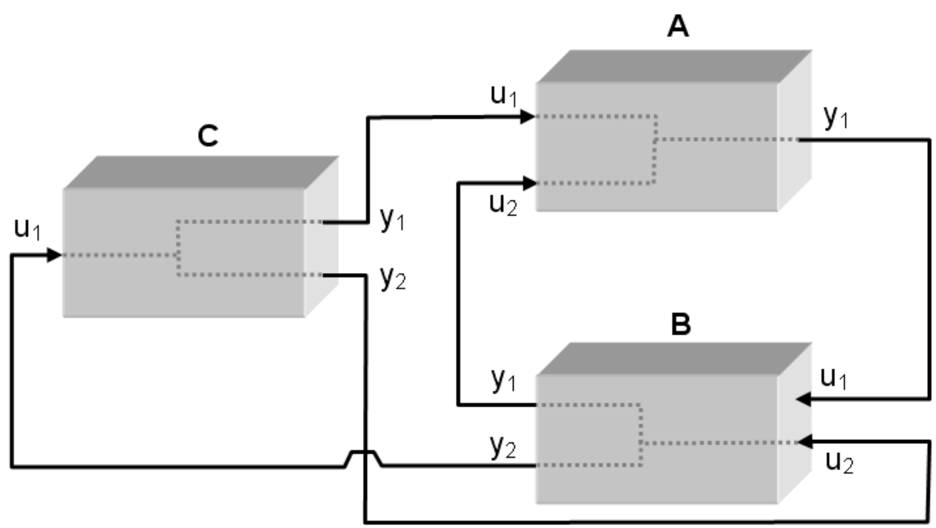
- FMI development was started by ITEA2 MODELISAR project. FMI is a Modelica Association Project now.
- **Version 1.0**
  - FMI for Model Exchange (released Jan 26,2010)
  - FMI for Co-Simulation (released Oct 12,2010)
- **Version 2.0**
  - FMI for Model Exchange and Co-Simulation (released July 25,2014)
- **73 tools** supporting it (https://www.fmi-standard.org/tools)

FMI – Main Design Idea

- A component which implements the interface is called
  Functional Mockup Unit (FMU)
- Separation of
  - Description of interface data (XML file)
  - Functionality (C code or binary)
- A FMU is a zipped file (*.fmu) containing the XML description file and the implementation in source or binary form
- Additional data and functionality can be included

Functional Mockup Units

- Import and export of input/output blocks – **Functional Mock-Up Units – FMU**s
- described by
  - differential-, algebraic-, discrete equations,
  - with time-, state, and step-events
- An FMU can be large (e.g. 100 000 variables)
- An FMU can be used in an embedded system (small overhead)
- FMUs can be connected together

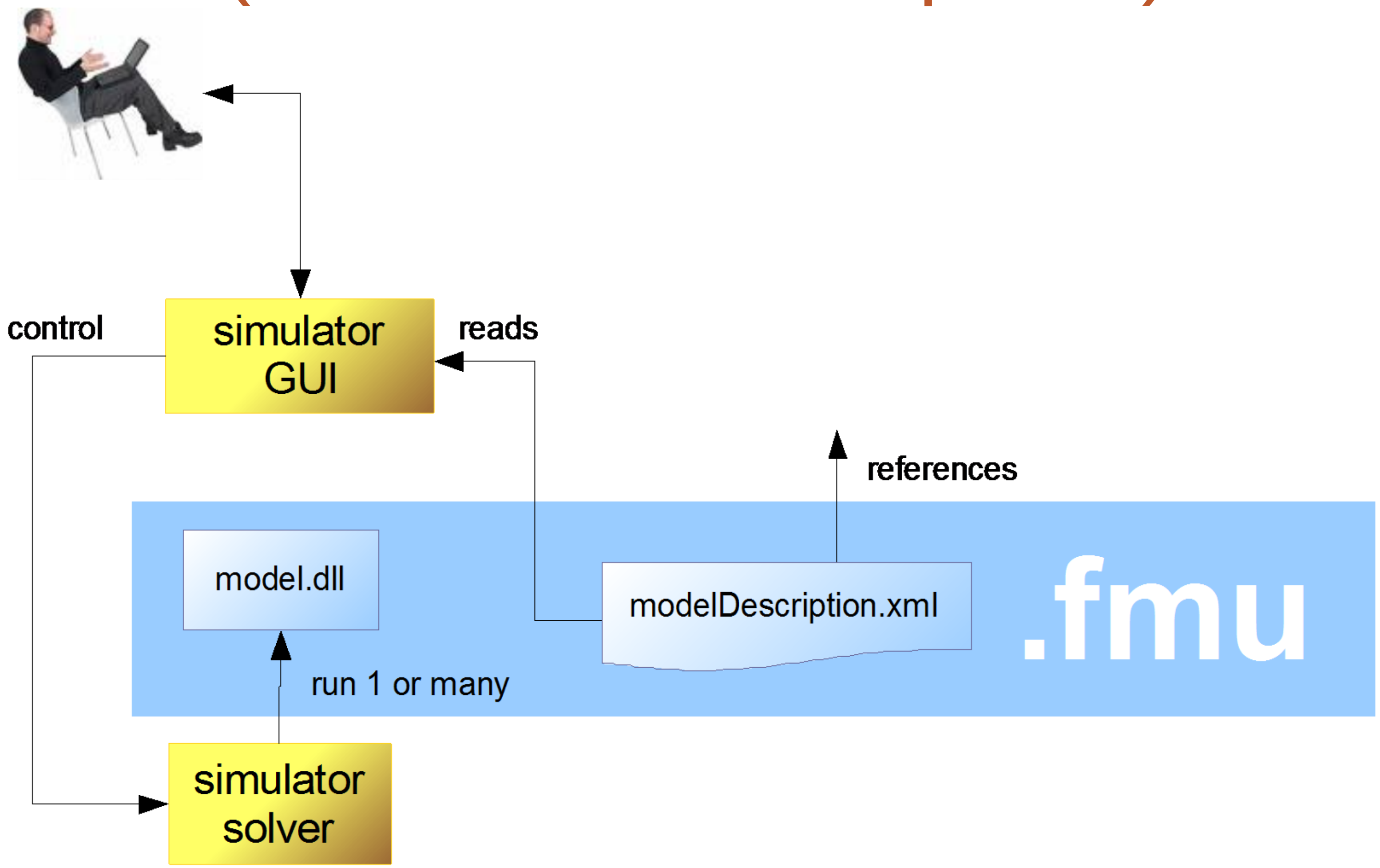


Model Distribution as a Zip-file (.fmu file)

A model is distributed as one zip-file with extension ".fmu", containing:

- **XML model description file**
  All model information that is not needed during integration of model, e.g., signal names and attributes. Advantage:
  - No overhead for model execution.
  - Tools can read this information with their preferred language (C/C++, C#, Java, ...)
- **Model equations** defined by a small set of **C-functions**. In zip-file:
  - **C source code** and/or
  - **Binary code** (DLL) for one or more platforms (Windows, Linux, ...)
- **Resources**
  - Documentation (html files)
  - Model icon (bitmap file)
  - Maps and tables (read by model during initialization)

Simulator with GUI and Solver Executing Imported Model = FMU (Functional Mockup Unit)



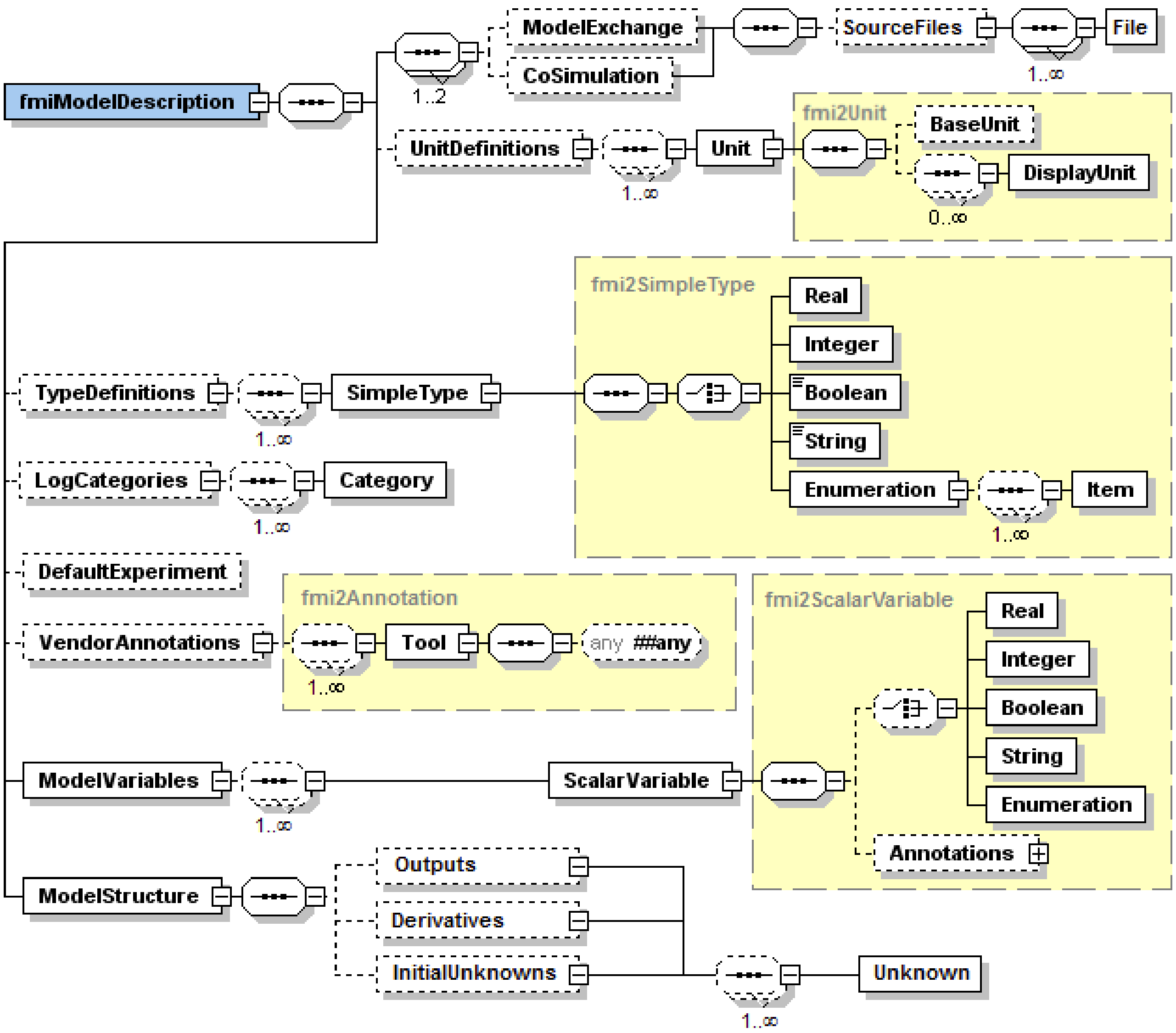
FMU
Functional Mockup Unit
in more detail

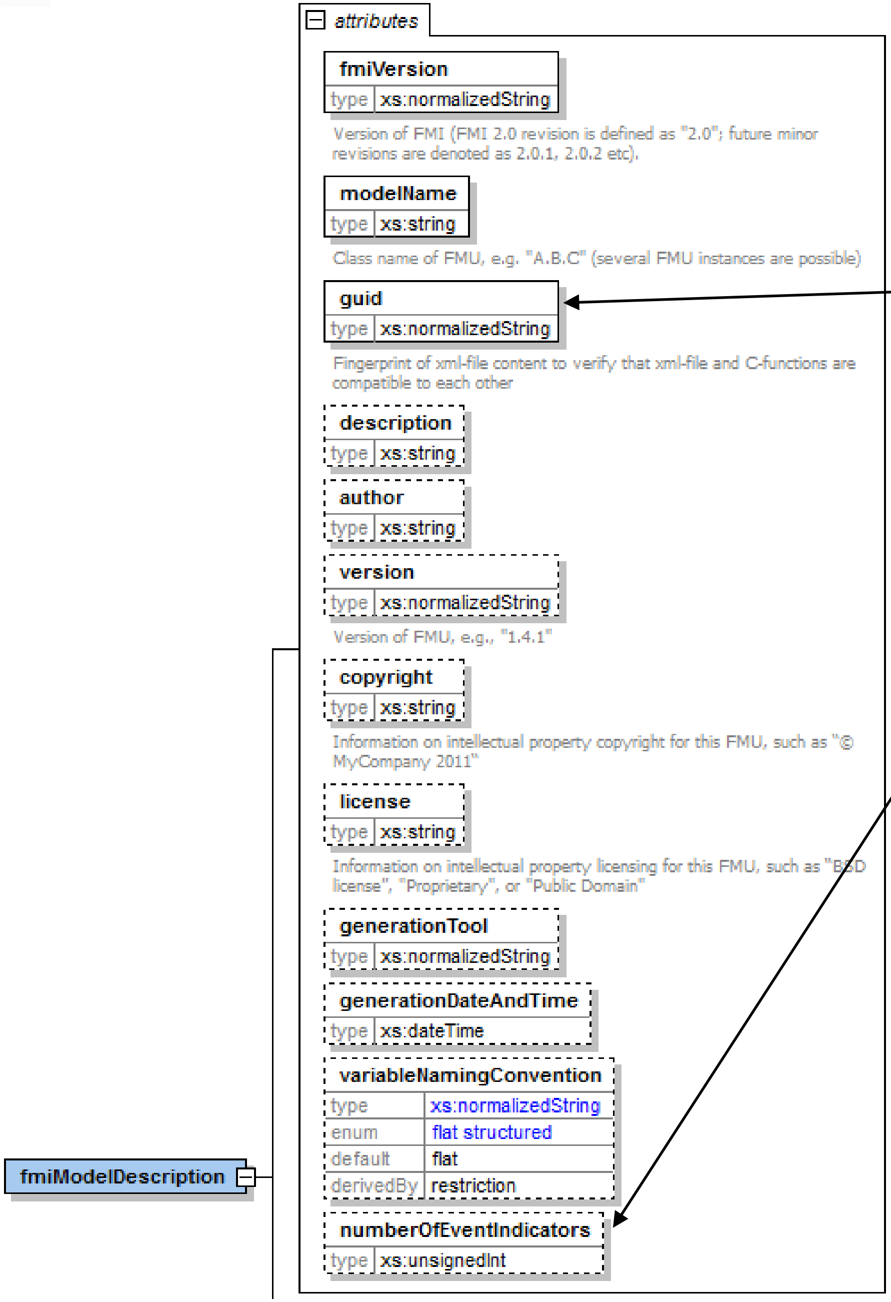
Structure of an FMU zip-file

```
1. modelDescription.xml     // Description of model (required file)
2. model.png                // Optional image file of model icon
3. documentation            // Optional directory containing the model
                         // documentation
   _main.html               // Entry point of the documentation
   <other documentation files>
4. sources                  // Optional directory containing all C-
sources
   // all needed C-sources and C-header files to compile and link the model
   // with exception of: fmiModelTypes.h and fmiModelFunctions.h
5. binaries                 // Optional directory containing the binaries
   win32 // Optional binaries for 32-bit Windows
      <modelIdentifier>.dll   // DLL of the model interface implementation
      VisualStudio8           // 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
      ...
6. resources   // Optional resources needed by the model
   < data in model specific files which will be read during initialization >
```

Model Description Schema

- **Model information** not needed for execution is stored in one **xml-file** (modelDescription.xml in zip-file) defined by **xml schema (.xsd) files**.





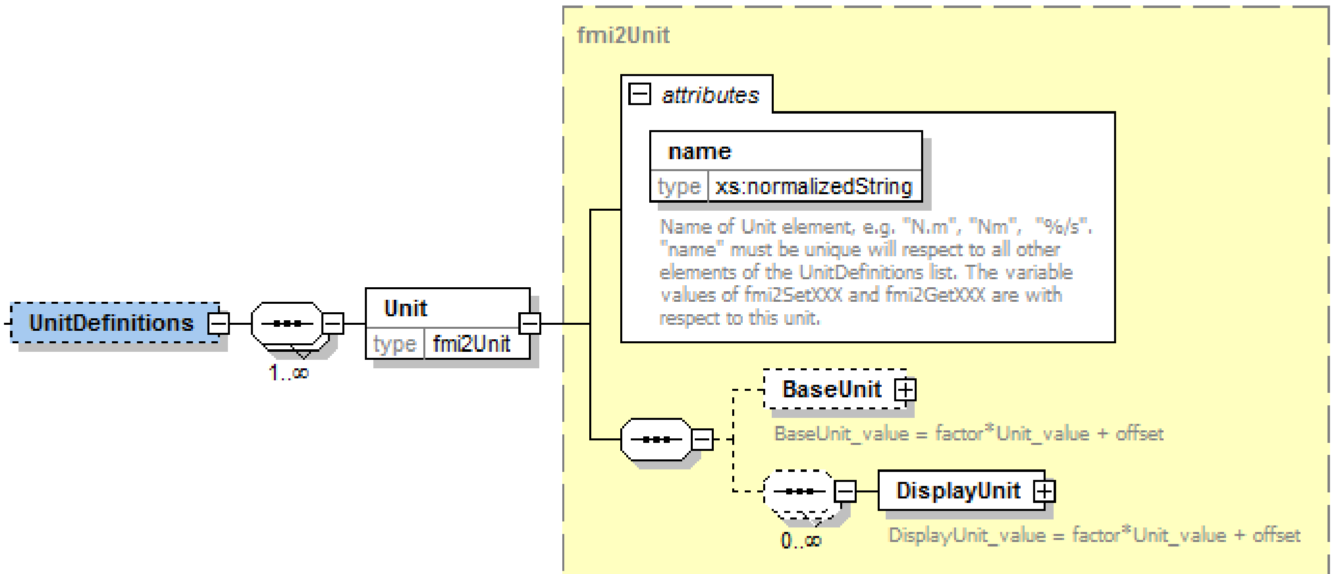
Model Attributes

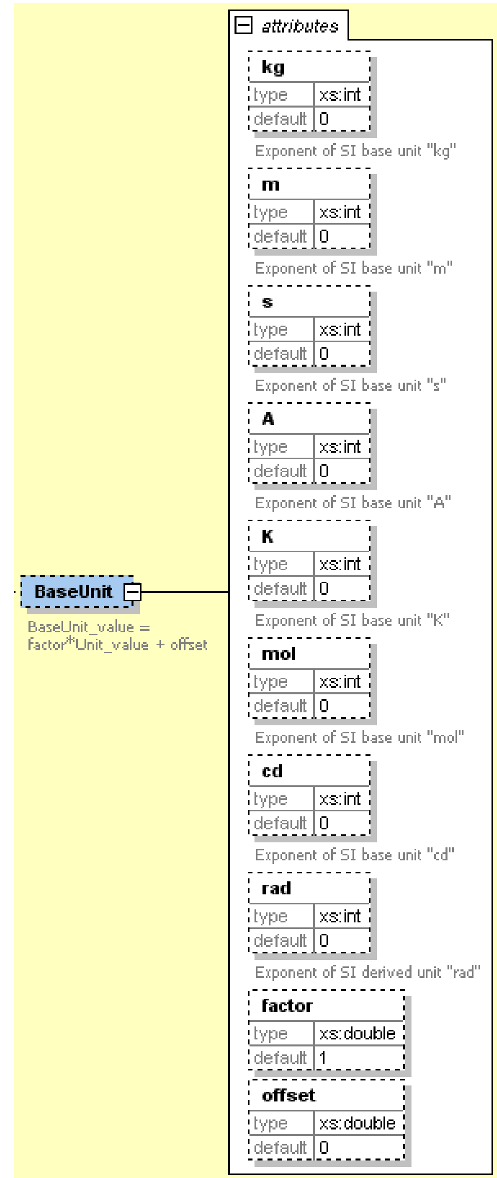
guid is a globally unique identifier ("fingerprint" of all releveant information in the xml file) that is also stored in the C-functions to gurantee consisteny

Number of event indicators; numbers are fixed (**numberofContinuousStates** have been removed in FMI 2.0 because it can be deduced from other information in the xml file.)

Unit Definitions

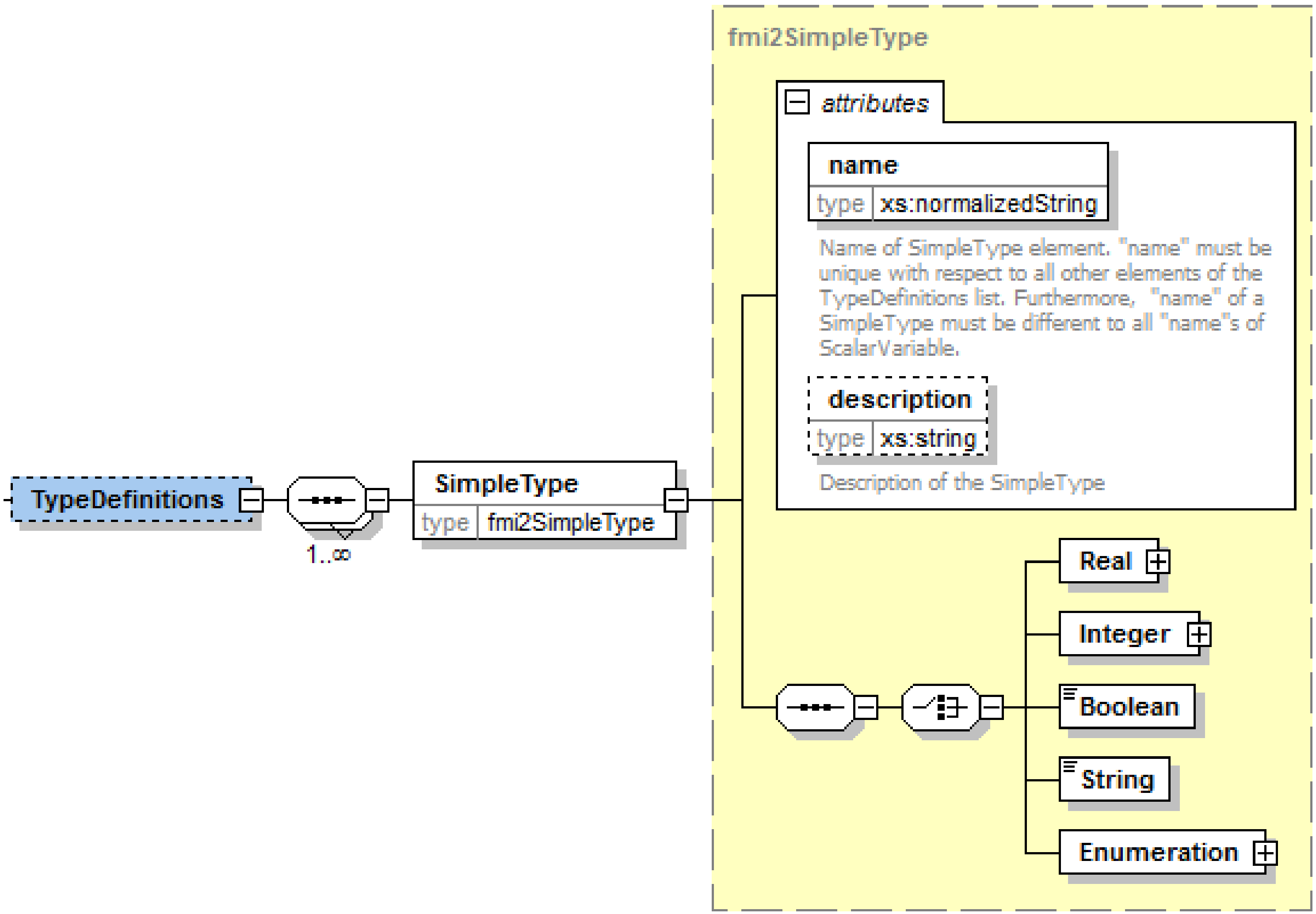
7 SI base units “kg”, “m”, “s”, “A”, “K”, “mol”, “cd”, and SI derived unit “rad”.





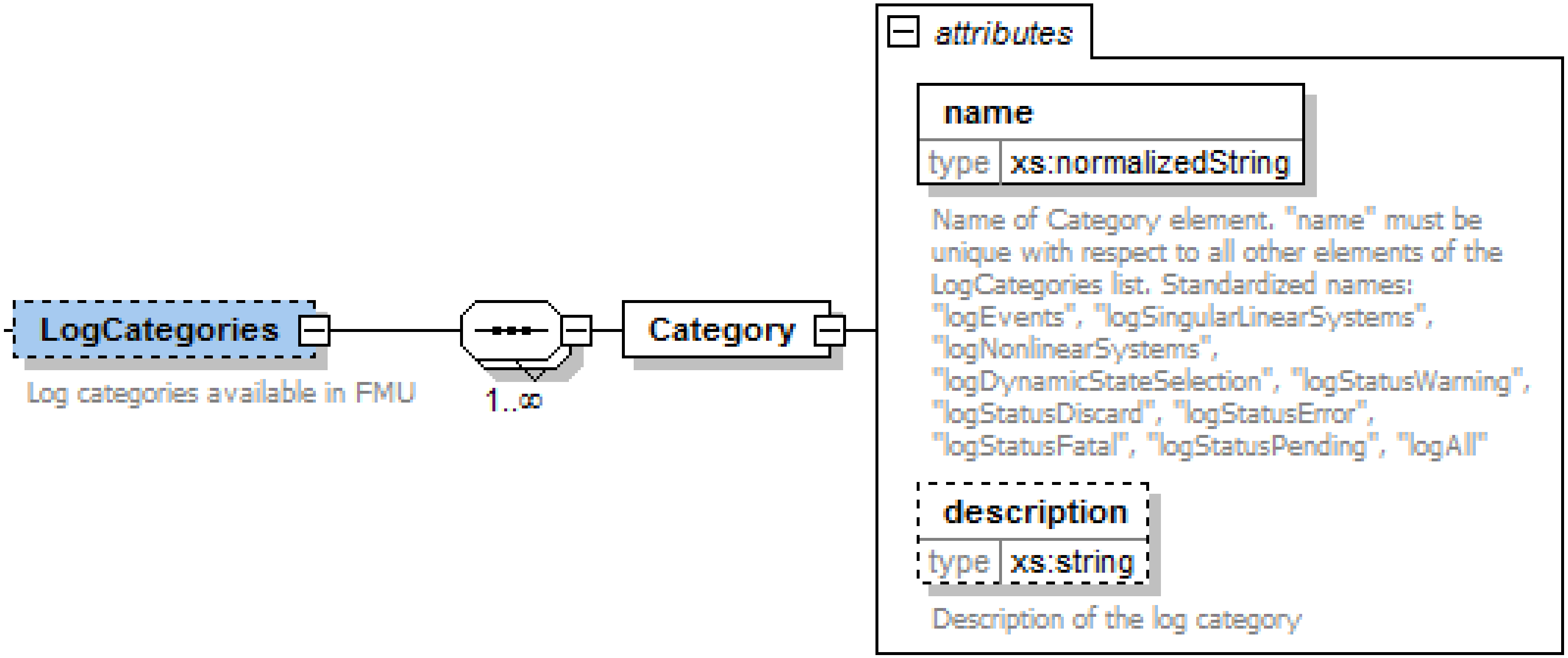
BaseUnit_value = `factor`*Unit_value + `offset`

Type Definitions

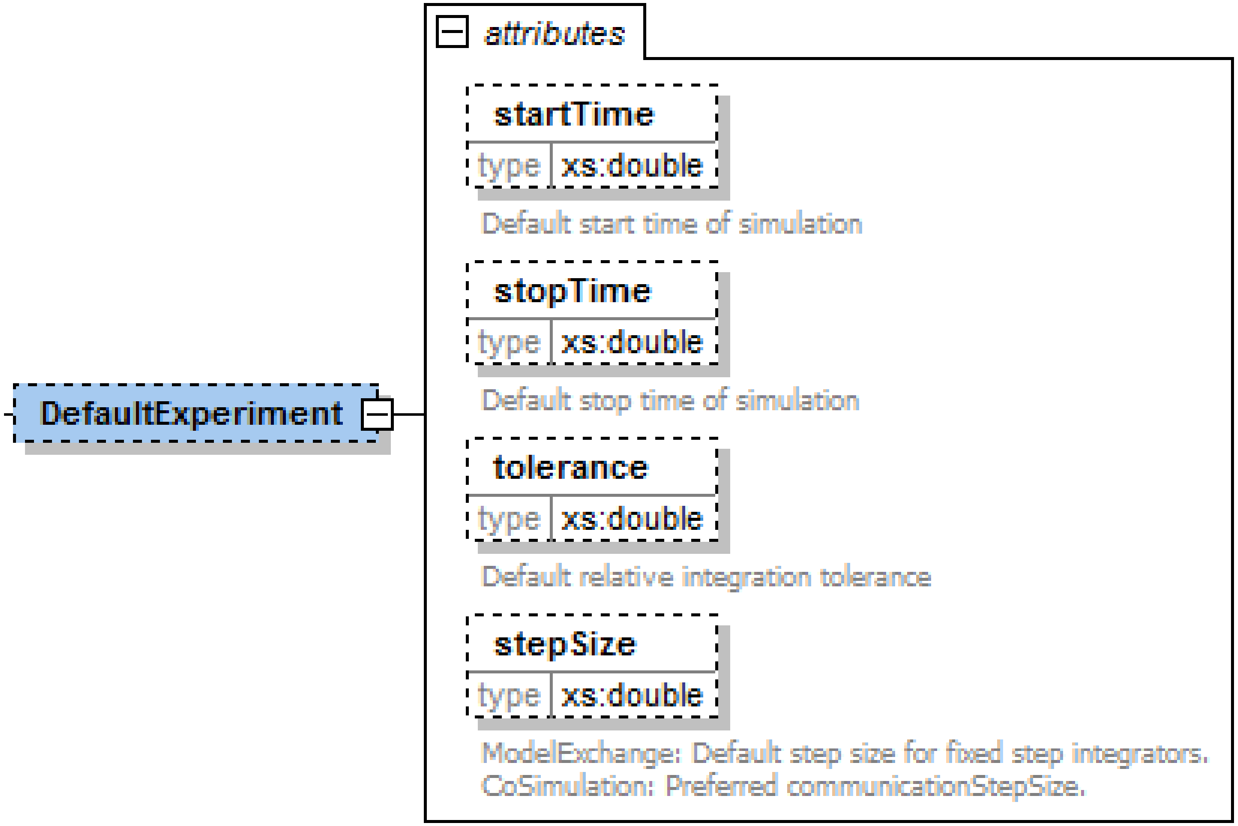


Log Categories

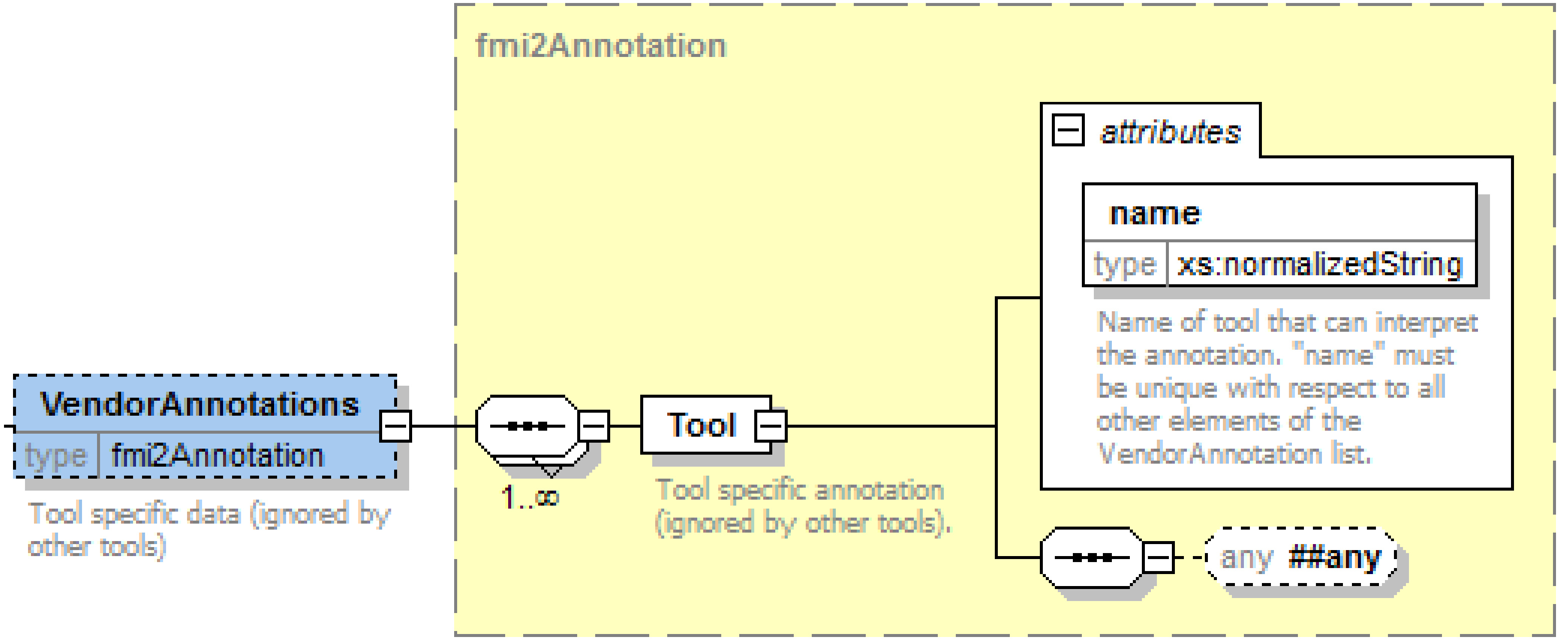
- unordered set of category strings that can be utilized to define the log output via function “logger”



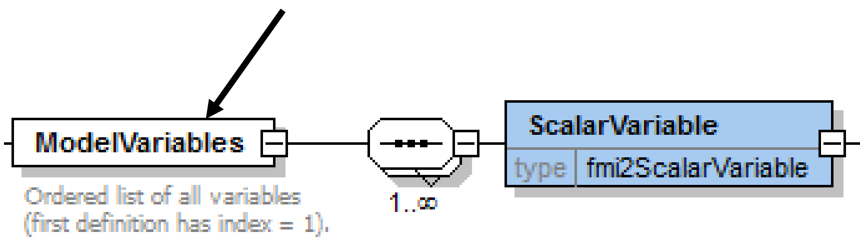
Default Experiment

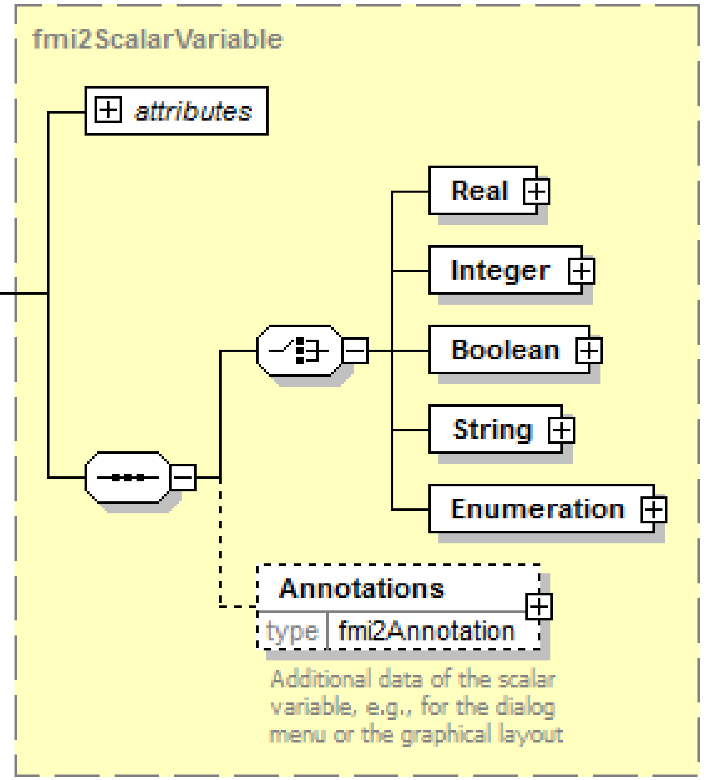


Vendor Annotations



Model Variables

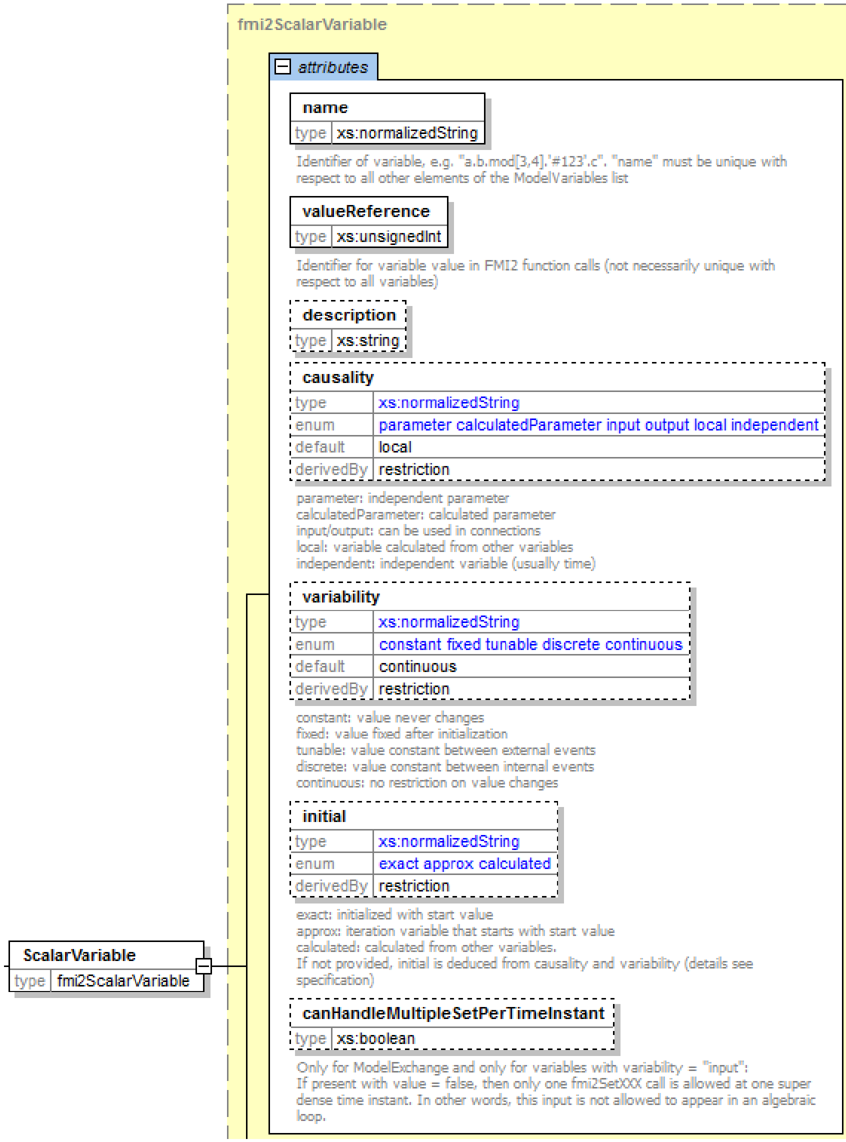




data types

ordered set of scalar variables (arrays, records, etc. must be mapped to scalars when generating code).

Attributes of Model Variables



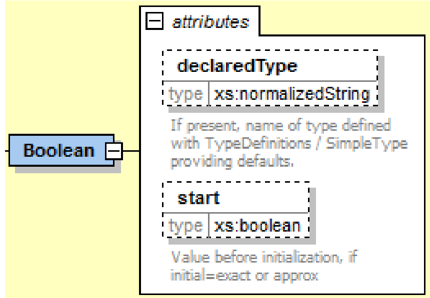
...

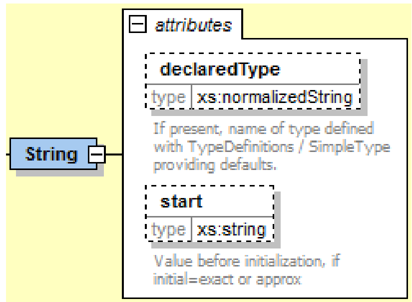
<u>unique name</u>

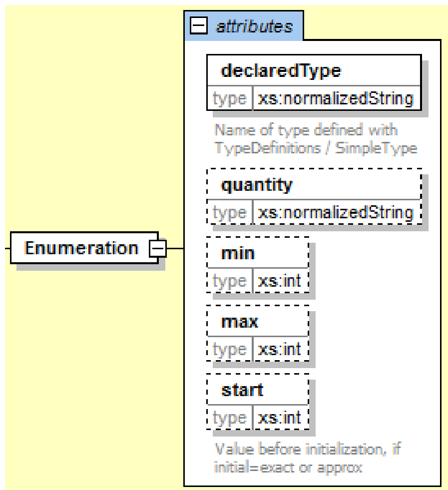
<u>handle</u> to identify variable in C-functions

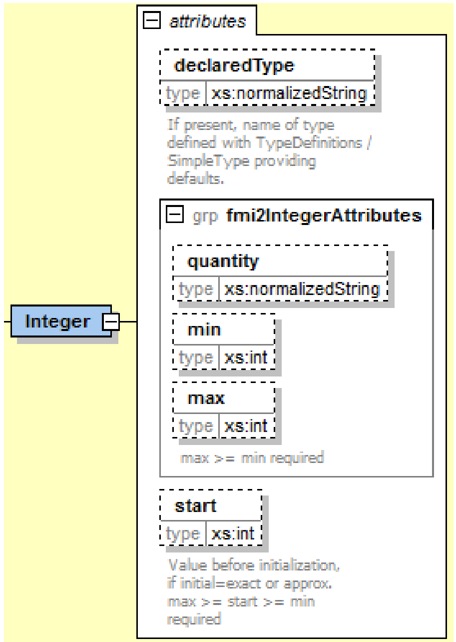
Data Types

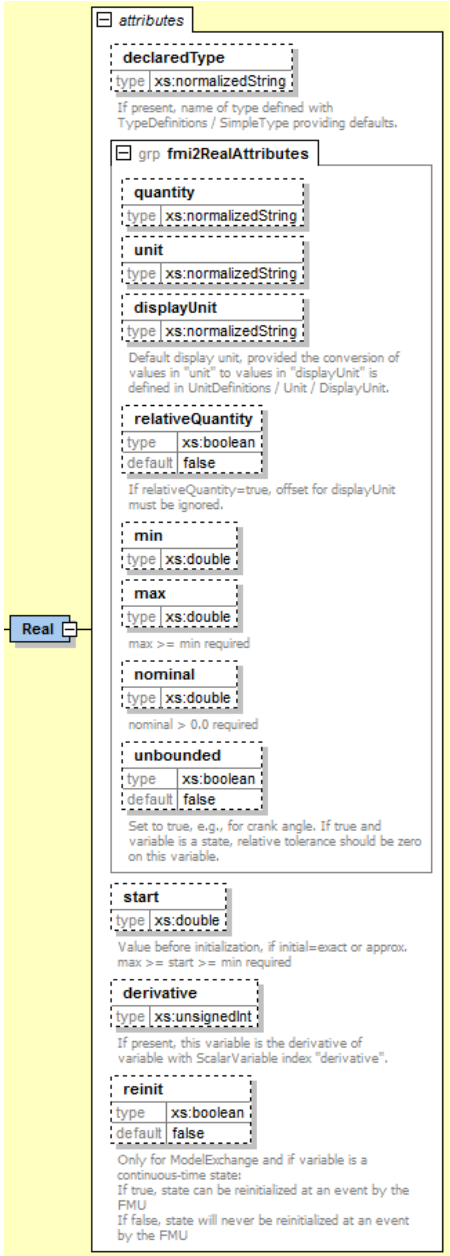
Data types allow to store all (relevant) Modelica attributes, including units. Defaults from TypeDefinitions



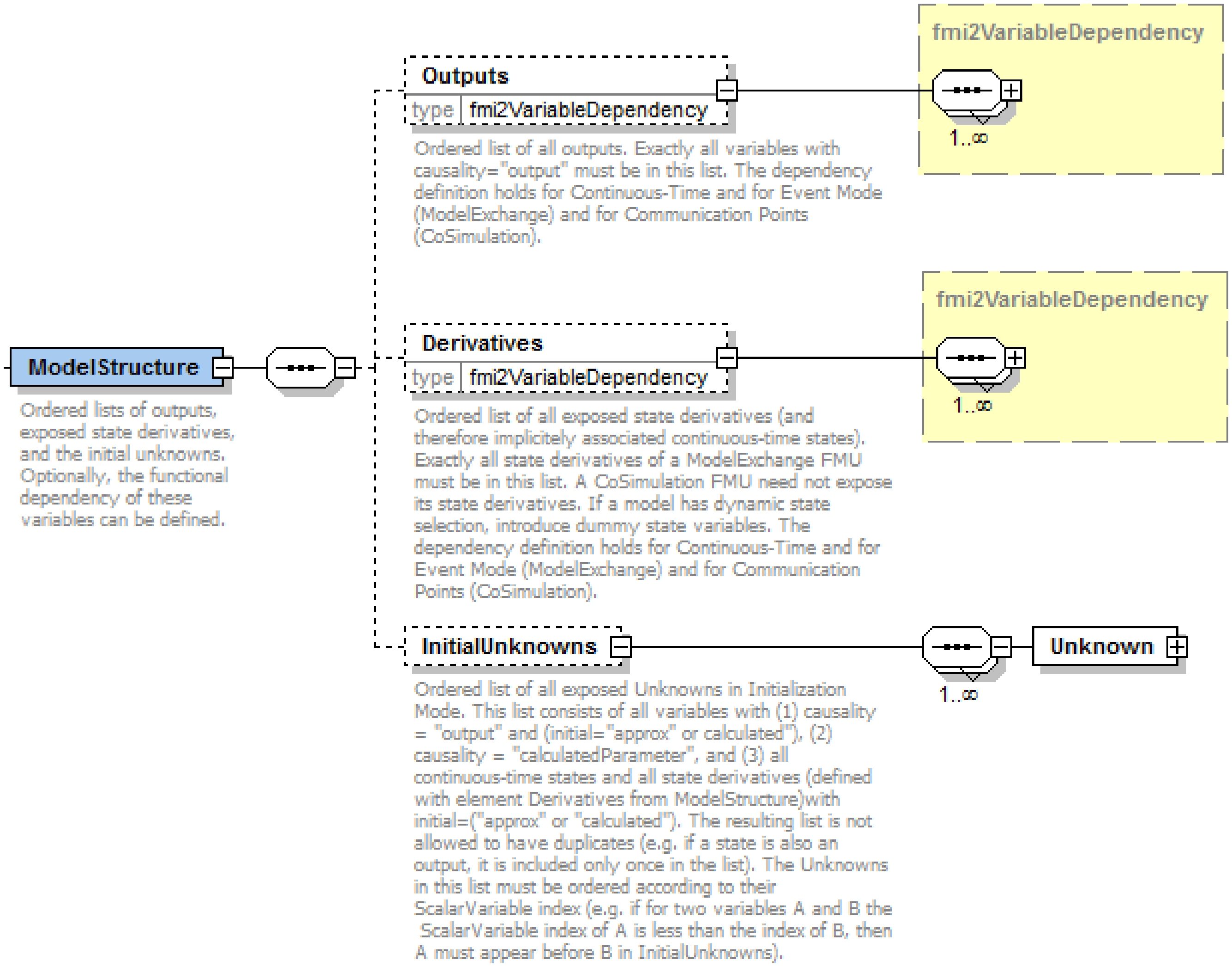








Model Structure



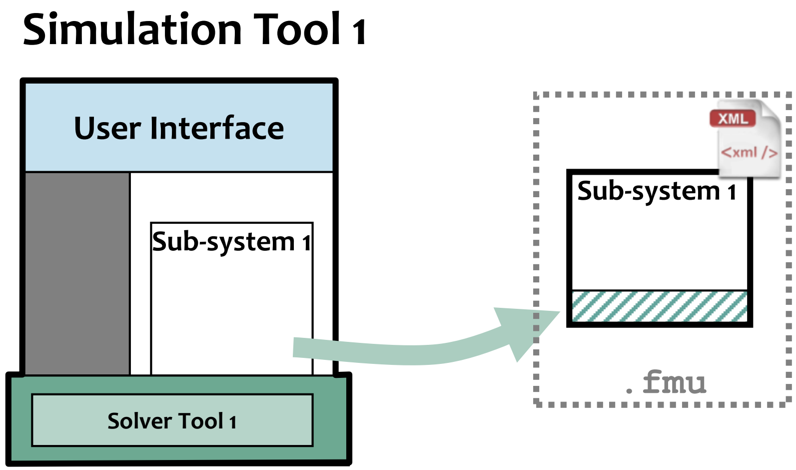
Example Model Description XML File

```
<?xml version="1.0" encoding="UTF8"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="Modelica.Mechanics.Rotational.Examples.Friction"
  modelIdentifier="Modelica_Mechanics_Rotational_Examples_Friction"
  guid="{8c4e810f-3df3-4a00-8276-176fa3c9f9e0}"
  ...
  numberOfEventIndicators="34"/>
  <UnitDefinitions>
    <Unit name="rad">
      <BaseUnit rad="1"/>
      <DisplayUnit name="deg" factor="57.2957795130823"/>
    </Unit>
  </UnitDefinitions>
  <TypeDefinitions>
    <SimpleType name="Modelica.SIunits.Inertia">
      <Real quantity="MomentOfInertia" unit="kg.m2" min="0.0"/>
    </SimpleType>
  </TypeDefinitions>
  <ModelVariables>
    <ScalarVariable
      name="inertia1.J"
      valueReference="1073741824"
      description="Moment of load inertia"
      causality="parameter"
      variability="fixed">
    <Real declaredType="Modelica.SIunits.Inertia" start="1"/>
    </ScalarVariable> <!—index="1" -->
    ...
  </ModelVariables>
</fmiModelDescription>
```

FMI for Model Exchange

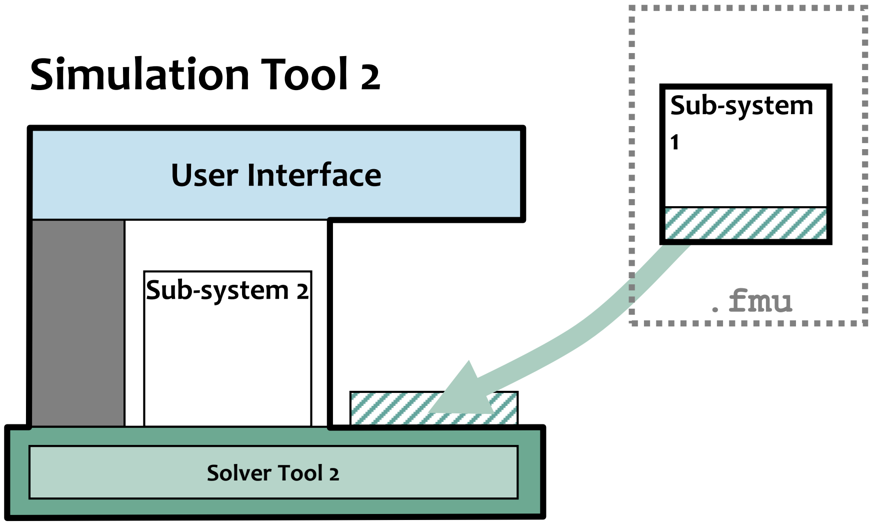
FMI for Model Exchange Export

- **Export**: **Subsystem model is exported** from its **simulation tool**
  - Preparation as FMU-archive containing
    - – model description (xml-file)
    - – executable dll-file containing model equations
    - – optionally C source code



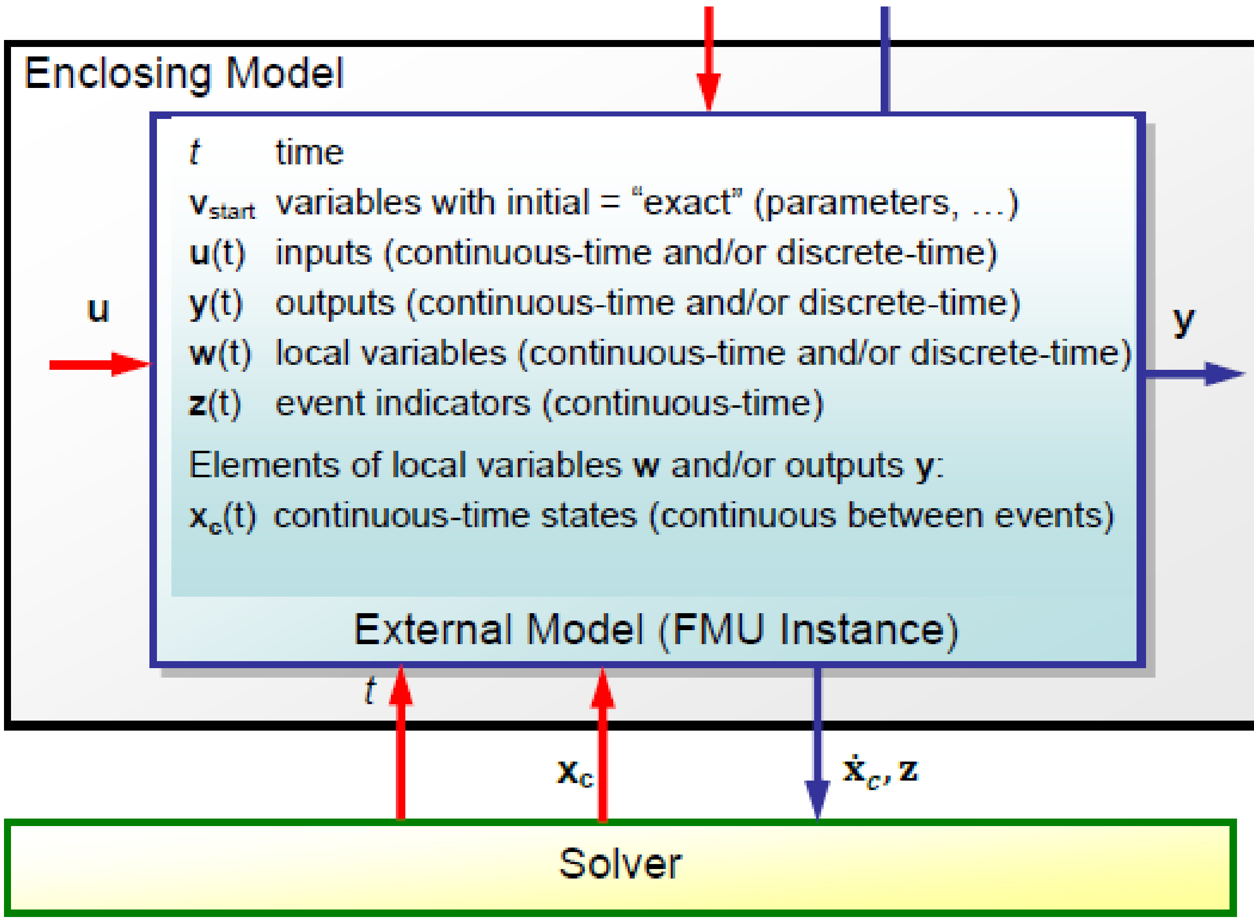
FMI for Model Exchange Import

- **Import: Subsystem model is imported** into simulation system for **system simulation**
  - Reading FMU-archive
    - model information from xml-file
    - connecting subsystem variables
    - executable model equations (dll)
    - running system simulation



FMI for Model Exchange Interface

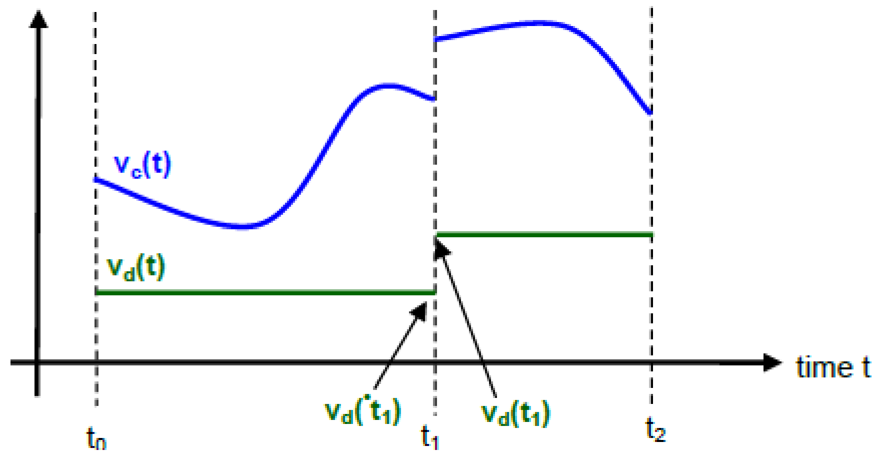
- Interfaces to Simulation Tool



Mathematical Description

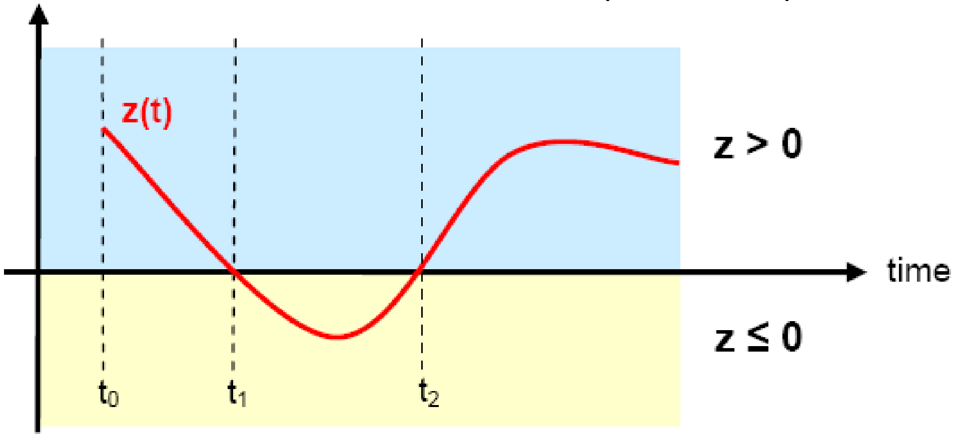
- Hybrid ODEs supported by FMI are described as piecewise continuous-time systems
- Continuous and discrete states

Index	*Description*
c	A **continuous-time** variable, that is a variable that is a continuous function of time inside each interval $t_i^+ \le t \le {}^-t_{i+1}$
d	A **discrete-time** variable, that is a variable that changes its value only at an event instant t_i.
c+d	A set of **continuous-time** and **discrete-time** variables



Events

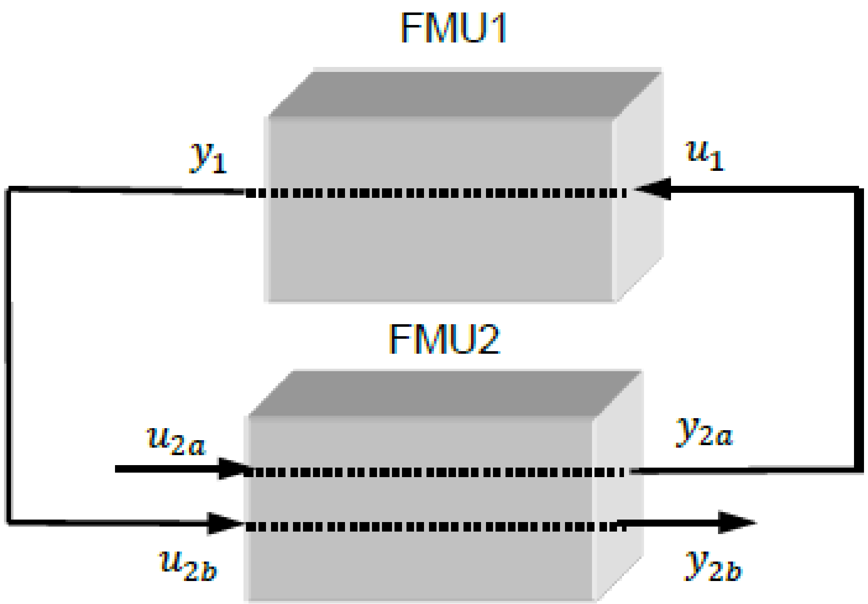
- Event instant t_i is defined by one of the following conditions,
  1. External Events
     - At least one discrete-time input changes its value.
     - A continuous-time input has a discontinuous change.
     - A tunable parameter changes its value.
  2. Time Events
     - A predefined time instant $t_i = (T_{next}(t_{i-1}), 0)$ that was defined at the previous event instant t_{i-1} by the FMU.
  3. State Events
     - When an event indicator $z_j(t)$ changes its domain from $z_i > 0$ to $z_i \leq 0$ or from $z_j \leq 0$ to $z_j > 0$.
  4. Step Events
     - At every completed step of an integrator.



Handling of Algebraic Loops

- Dependency information is needed e.g which outputs depends directly on inputs.
- `<ModelStructure>` defined in the fmu.

artifical algebraic loop



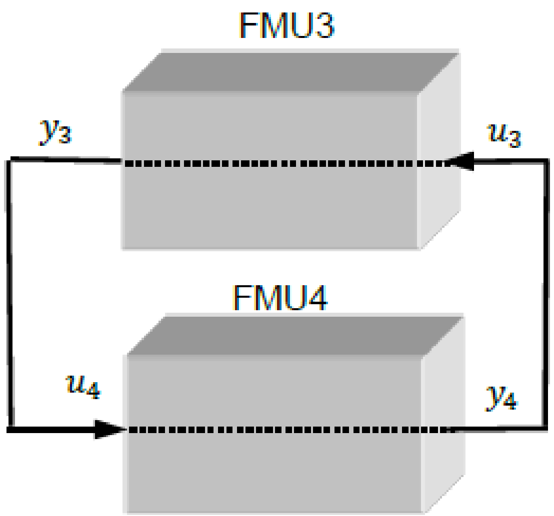
sequential calling sequence:

fmiSetXXX(m2,< u_{2a}>, ...)
y_{2a} := fmiGetXXX(m2, ...)
fmiSetXXX(m1, < u_1 := y_{2a} >, ...)
y_1 := fmiGetXXX(m1,..)
fmiSetXXX(m2, < u_{2b} := y_1 >, ...)
y_{2b} := fmiGetXXX(m2, ...)

Handling of Algebraic Loops

- Iterative Newton method.
- In each iteration $u4$ is provided by the solver and the residue is computed and is provided back to the solver. Based on the residue a new value of $u4$ is provided. The iteration is terminated when the residue is close to zero.

“real” algebraic loop



iterative calling sequence:

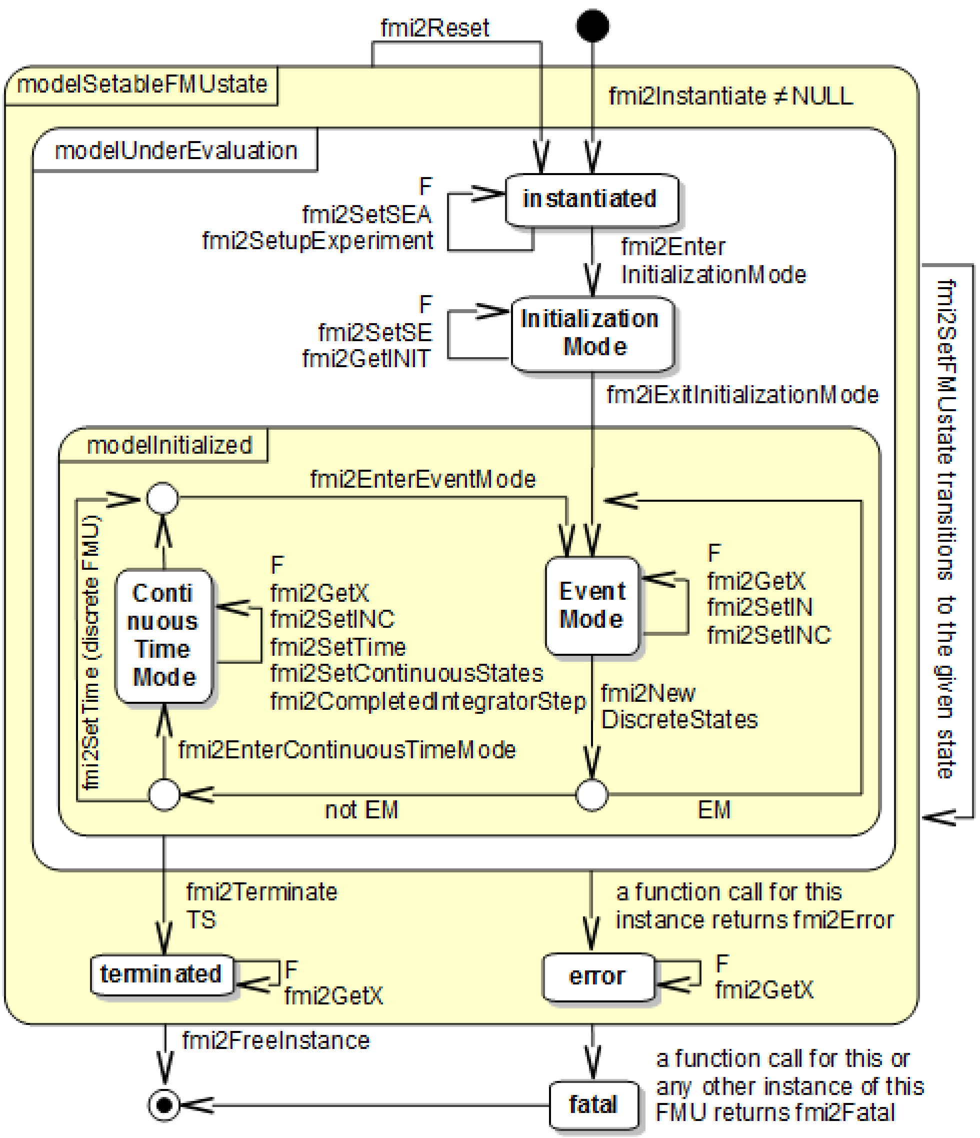
In every Newton iteration evaluate:
input: u_4 // provided by solver
output: residue // provided to solver
fmiSetXXX(m4,< u_4>, …)
y_4 := fmiGetXXX(m4, ...)
fmiSetXXX(m3, < $u_3 := y_4$ >, ...)
y_3 := fmiGetXXX(m3,..)
residue := $u_4 - y_3$

Model Exchange FMU Solution

In order to solve a FMI model we need to split its solution process into different phases,

- Initialization Mode
  - Compute initial values of states at time t_0.
- Continuous-time Mode
  - Compute continuous-time variables between events.
  - Discrete-time variables remains fixed.
- Event Mode
  - Compute new values for continuous-time and discrete-time variables.

Call Sequence State Machine



Abbreviations used in transition labels

F is one of
- *fmi2GetTypesPlatform*
- *fmi2GetVersion*
- *fmi2SetDebugLogging*
- *fmi2GetFMUstate*
- *fmi2FreeFMUstate*
- *fmi2SerializedFMUstateSize*
- *fmi2SerializeFMUstate*
- *fmi2DeSerializeFMUstate*
- *fmi2GetNominalsOfContinuousStates*

X is one of
- *Real, Integer, Boolean, String*
- *Derivatives*
- *ContinuousStates*
- *EventIndicators*
- *DirectionalDerivative*

SEA is one of Real, Integer, Boolean, String for a variable with variability ≠ "constant" that has initial = "exact" or "approx"

SE is one of Real, Integer, Boolean, String for a variable with variability ≠ "constant" that has initial = "exact" or causality="input"

INC is 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"

INIT is one of
- *Real, Integer, Boolean, String for variables with causality="output", continuous-time states and state derivatives*
- *ContinuousStates*
- *DirectionalDerivative*

TS: terminateSimulation is returned by at least one FMU or the simulation run ended successfully

EM: newDiscreteStatesNeeded is returned by at least one FMU and no FMU returns terminateSimulation

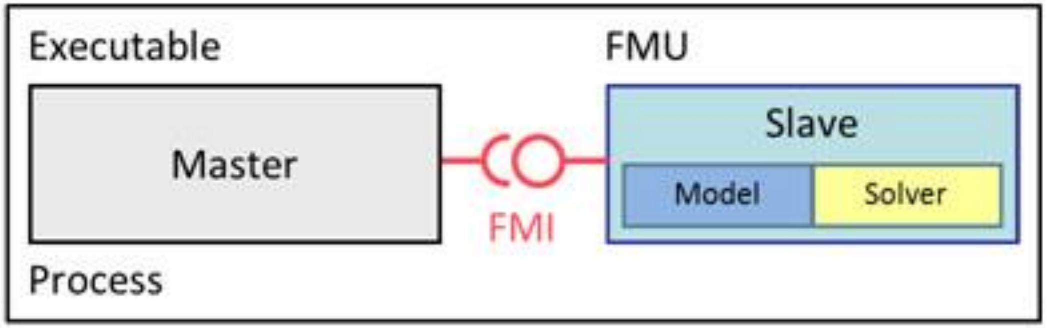
FMI for Co-Simulation

FMI for Co-Simulation

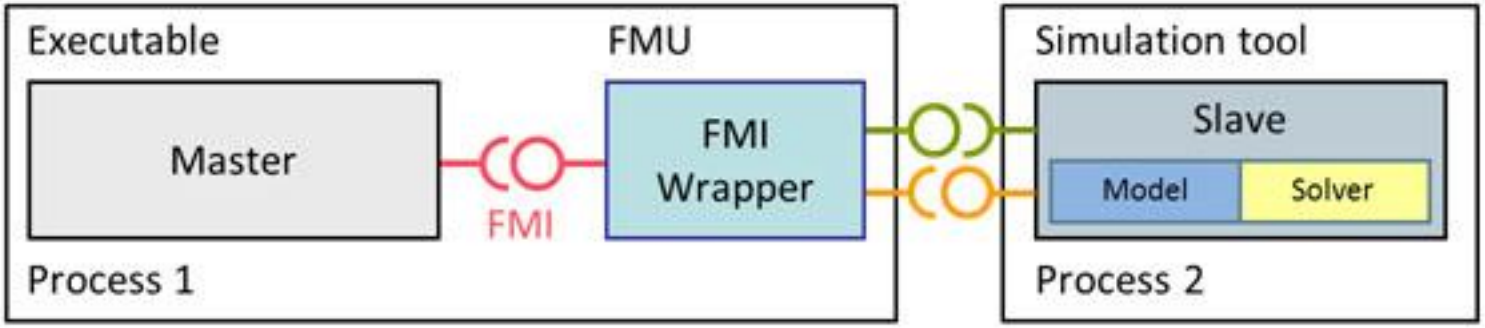
- Master/slave architecture
- Support of simple and sophisticated coupling algorithms:
  - Iterative and straight forward algorithms
  - Constant and variable communication step size
- Allows (higher order) interpolation of continuous inputs
- Support of local and distributed co-simulation scenarios
- FMI for Co-Simulation does not define:
  - Co-simulation algorithms
  - Communication technology for distributed scenarios

FMI for Co-Simulation Coupling

- Its been designed both for coupling with subsystem models, which have been exported by their simulators together with its solvers as runnable code,

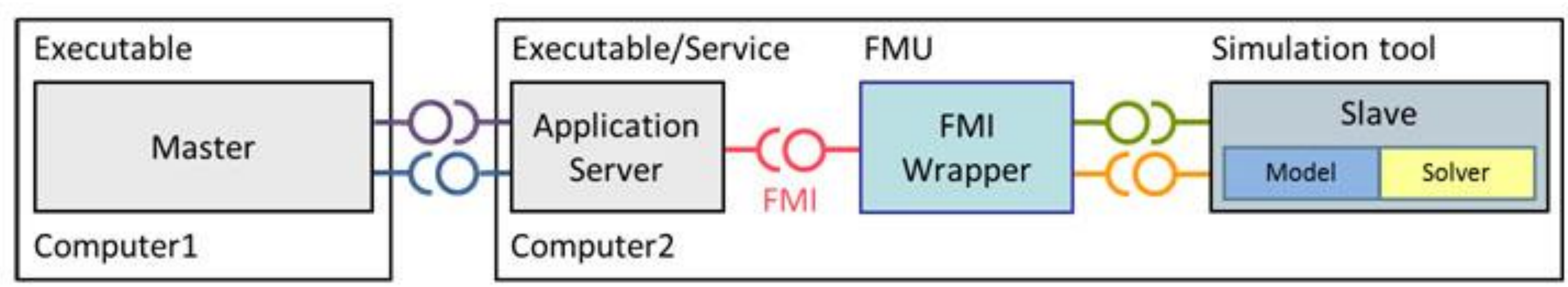


- And for coupling of simulation tools,



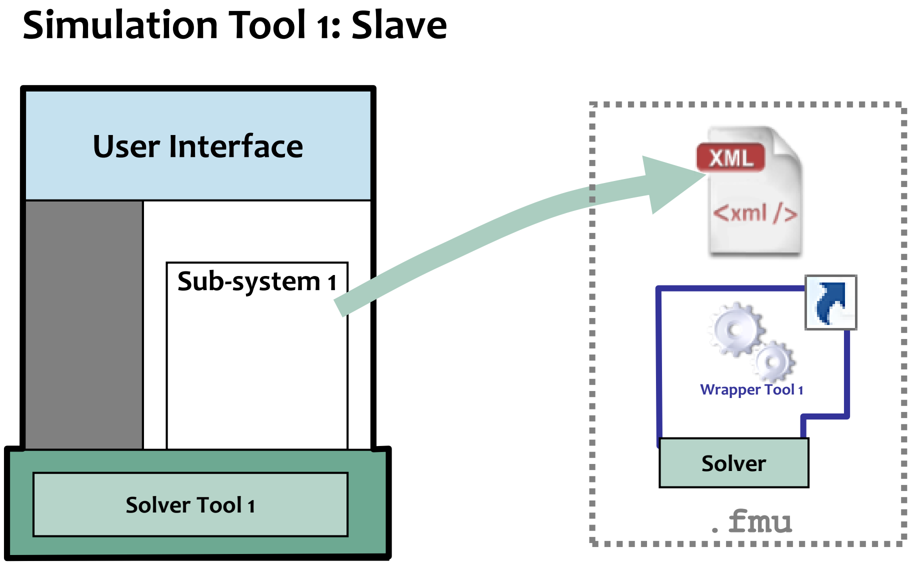
FMI for Co-Simulation Distributed

- Distributed Co-Simulation Scenario
  - Data exchange is handled by some network communication technology.
  - Communication layer not part of the FMI standard.
  - Master is responsible for the communication layer implementation.



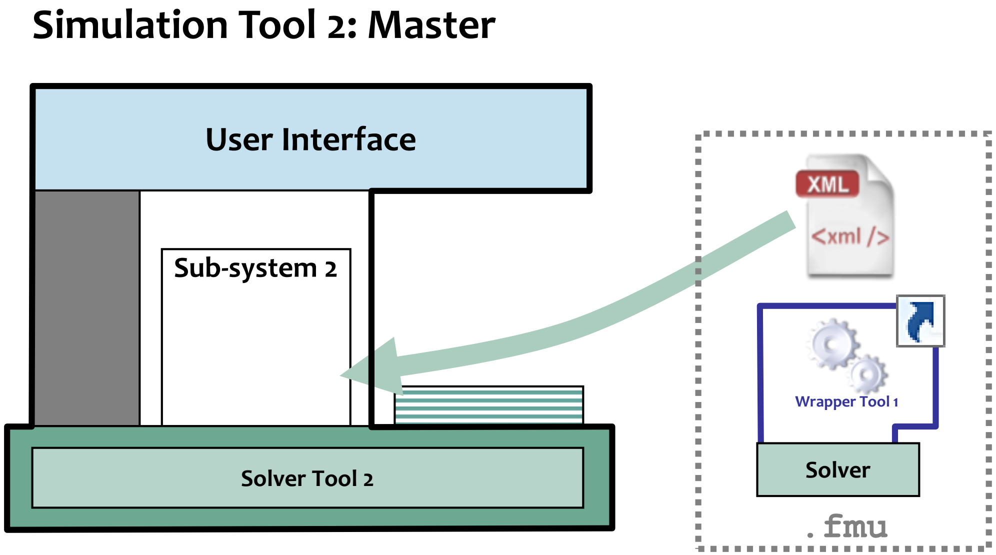
FMI for Co-Simulation Export FMU with Solver

- **Export: Subsystem description** is exported from its simulation tool
  - Preparation as FMU-archive containing
    - model description (xml-file), describes also solver/tool capabilities
    - reference to executable dll-file as, wrapper which provides a tool specific implementation of the co-simulation slave interface



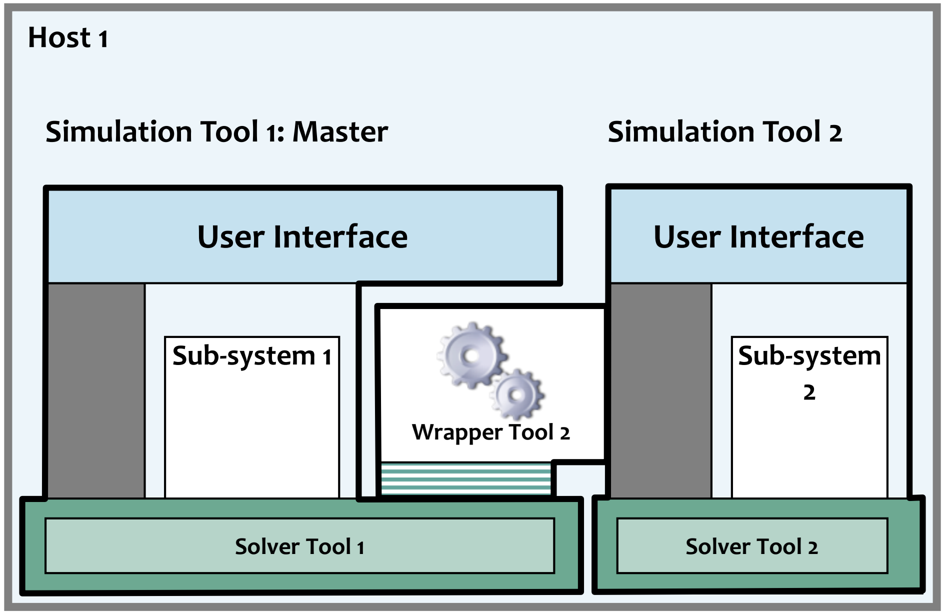
FMI for Co-Simulation Import Stand-alone

- **Import: Subsystem description** is imported into simulation system for system simulation
  - Reading FMU-archive
    - model information from xml-file
    - connecting subsystem variables



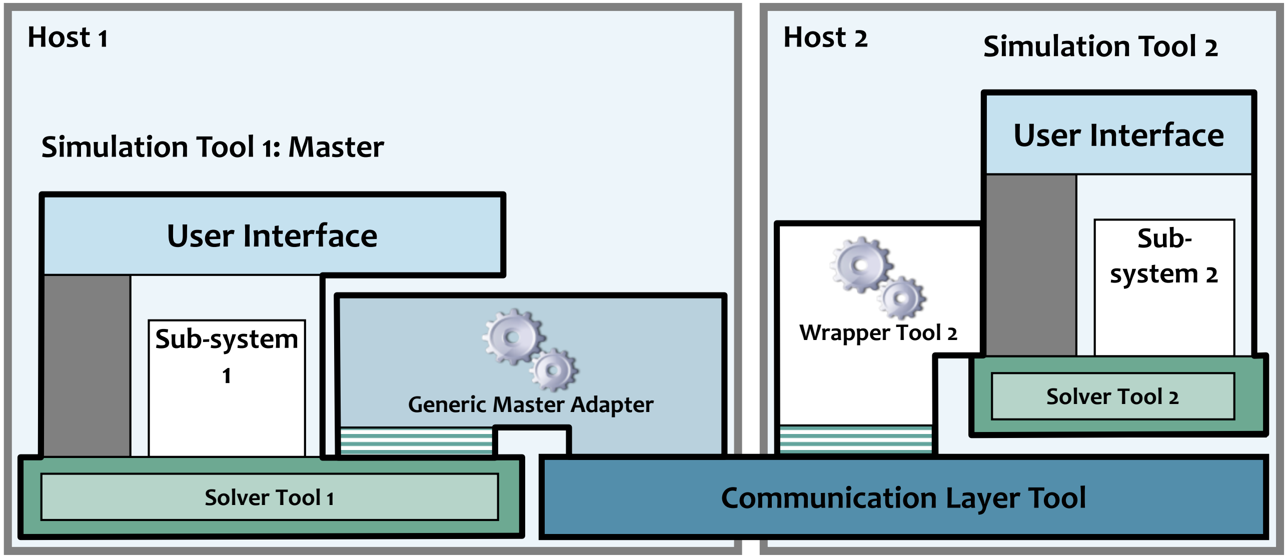
FMI for Co-Simulation Tool coupling

- **Run simulation on same host**
  - Master subsystem is connected with wrapper dll via co-simulation interface
  - Subsystem 2 is called via wrapper of tool 2 as if it would have been directly imported into master simulation tool

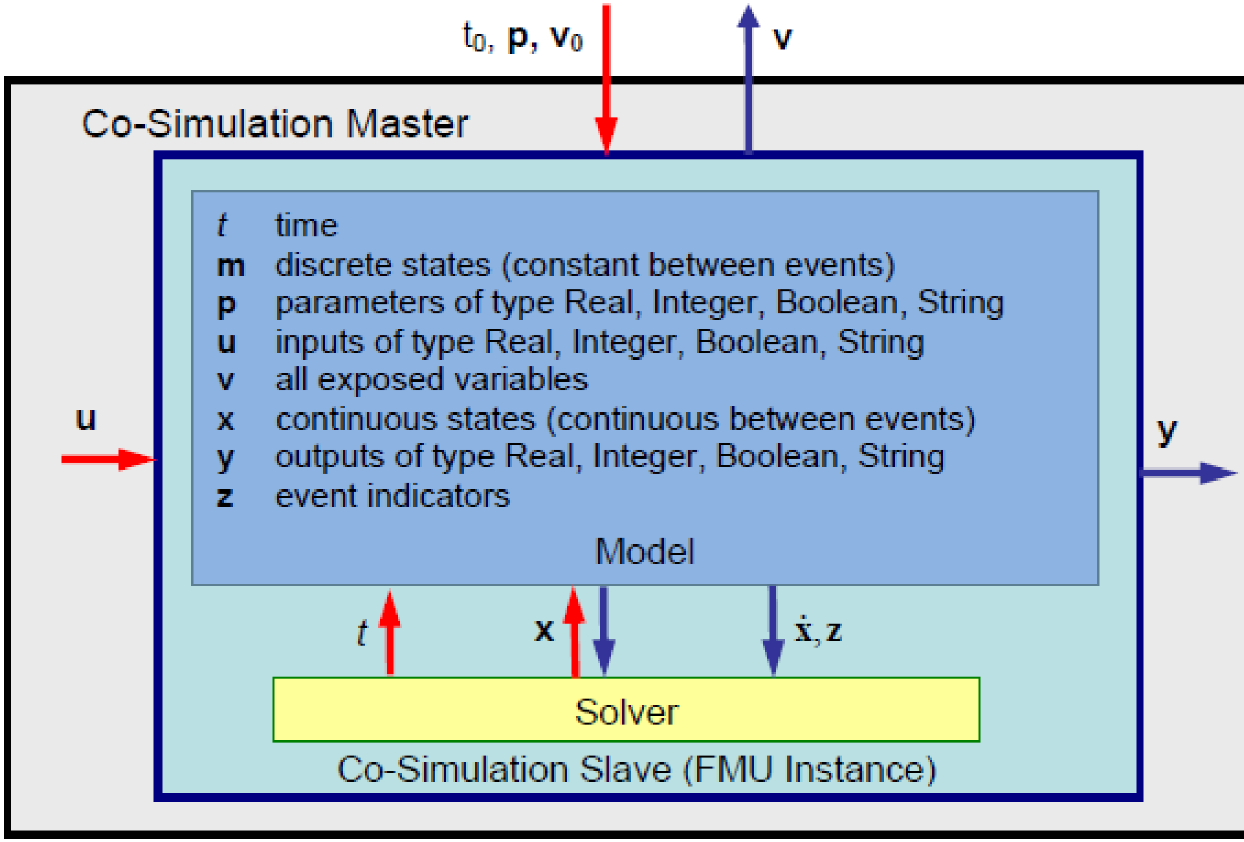


FMI for Co-Simulation distributed tool coupling

- **Run simulation on different hosts**
  - Master subsystem is connected via a generic adapter with a communication tool
    - Adapter provides co-simulation slave interface
  - Communication tool uses wrapper dlls of slave tools



FMI for Co-Simulation Interface



Mathematical Description

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 $tc_N := t_{stop}$.

Common Master Algorithm

- Stops at each communication point of all slaves
- Collects the output from all slaves
- Evaluates the slaves inputs
- Distributes the inputs to the slaves and continue simulation with the next communication step with fixed communication step size
- Slave’s solver is used for integration

FMI for Co-Simulation is designed to support a very general class of master algorithms but it does not define the master algorithm itself.

Sophisticated Master Algorithm

Capability flags,

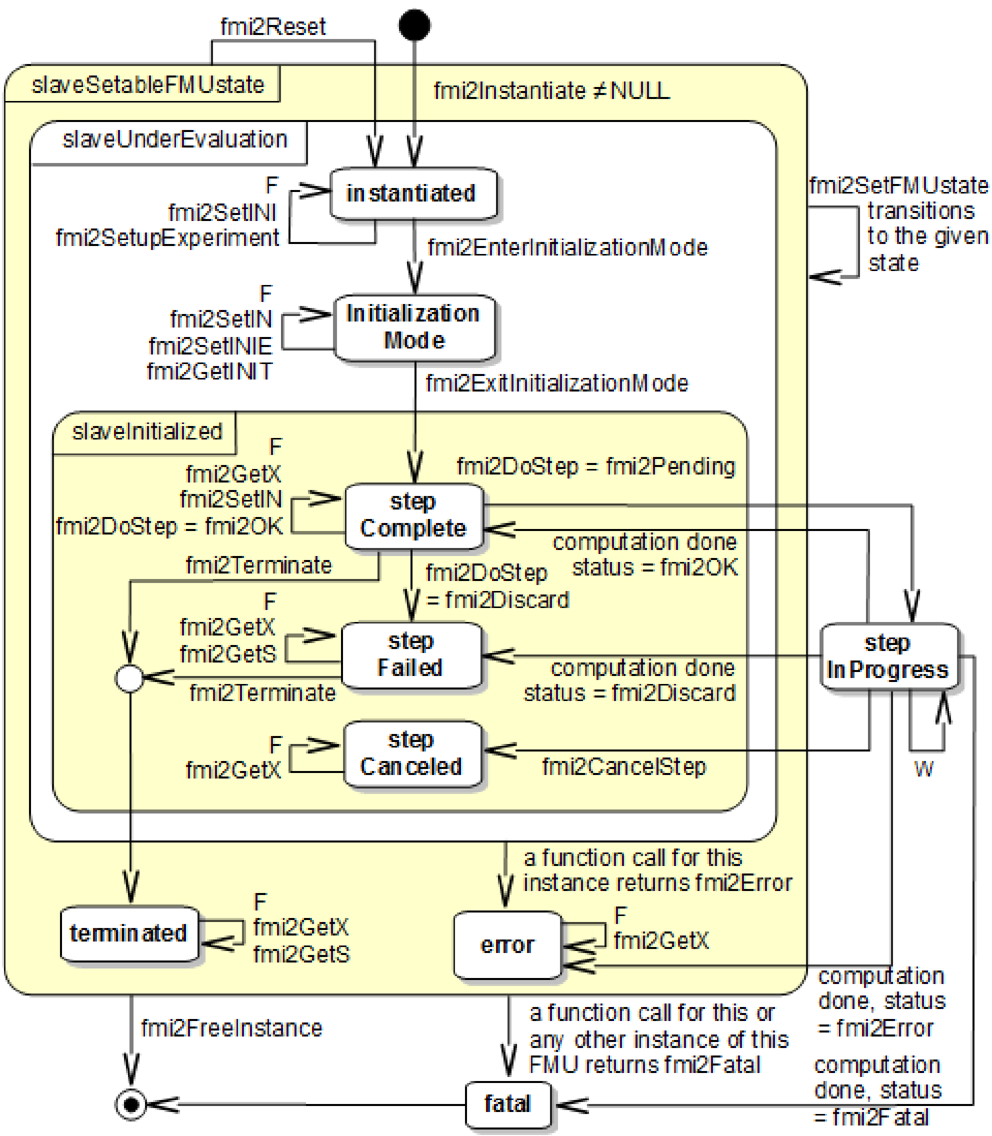
- Variable communication step size
- Repeat a rejected communication step $tc_i \rightarrow tc_{i+1}$ with reduced communication step size
- Provide derivatives w.r.t. time of outputs to allow interpolation
- Provides Jacobians.

Co-Simulation FMU Solution

In order to solve a FMI co-simulation model we need to split its solution process into two phases,

- Initialization Mode
  - Compute initial values of internal variables of the slave at time t_0.
- Step Mode
  - Compute the values of all (real) continuous-time variables at communication points.

FMI for Co-Simulation



Abbreviations used in transition labels

F is one of
- *fmi2GetTypesPlatform, fmi2GetVersion*
- *fmi2SetDebugLogging*
- *fmi2GetFMUstate, fmiFreeFMUstate*
- *fmi2SerializedFMUstateSize*
- *fmi2SerializeFMUstate*
- *fmi2DeSerializeFMUstate*

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

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

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

INIE is Real, Integer, Boolean, String for a variable with variability ≠ "constant" that has initial = "exact"

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

INIT is one of
- *Real, Integer, Boolean, String for variables with causality="output", if <Derivatives> present continuous-time states and state derivatives*
- *ContinuousStates (if <Derivatives> present)*
- *DirectionalDerivative*

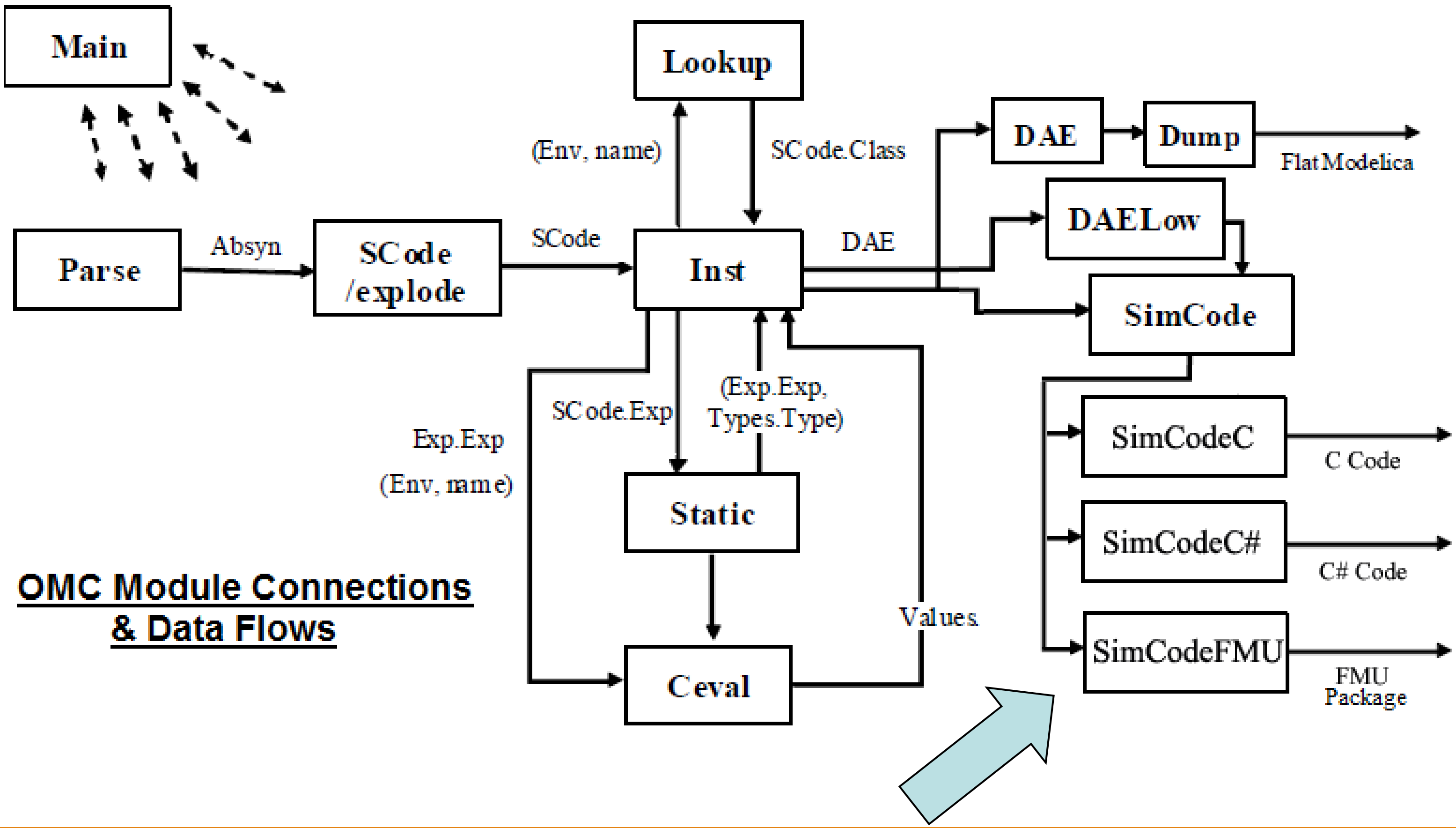
W is one of
- *fmi2GetTypesPlatform, fmi2GetVersion*
- *fmi2SetDebugLogging*
- *fmiGetS*

FMI
OpenModelica Implementation and Applications

FMI in OpenModelica

- Full Model Exchange support (FMI 1.0 & FMI 2.0)
- Co-simulation Export (FMI 2.0)
- Co-Simulation Import (under development)

OpenModelica Compiler and Code Generators Including New SimCodeFMU for FMI Export



FMI Export in OpenModelica

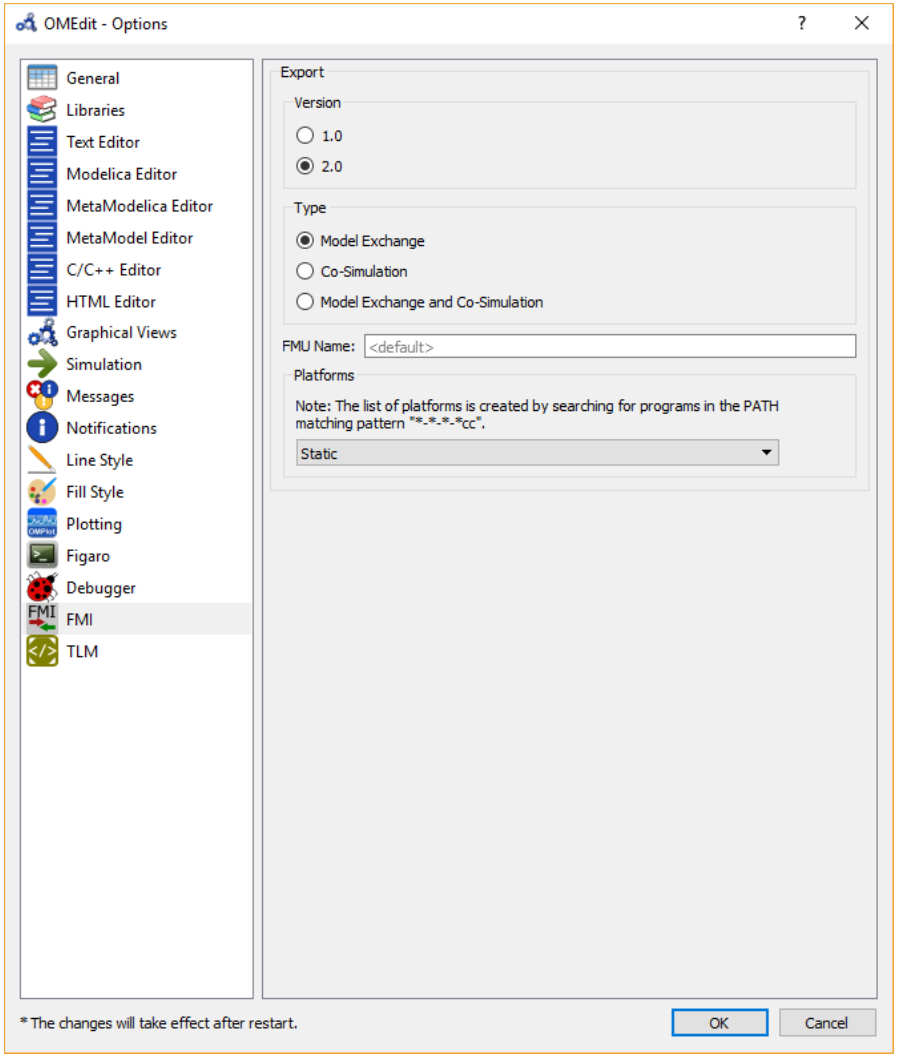
- OpenModelica scripting API

```
function translateModelFMU
  input TypeName className "the class that should translated";
  input String version = "2.0" "FMU version, 1.0 or 2.0.";
  input String fmuType = "me" "FMU type, me (model exchange), cs (co-simulation), me_cs
(both model exchange and co-simulation)";
end translateModelFMU;
```

- Creates an FMU of the model
- Version parameter specifies the version of the FMU
- Type parameter specifies the type of the FMU

- Export FMUs for different platforms.

FMI Export in OpenModelica



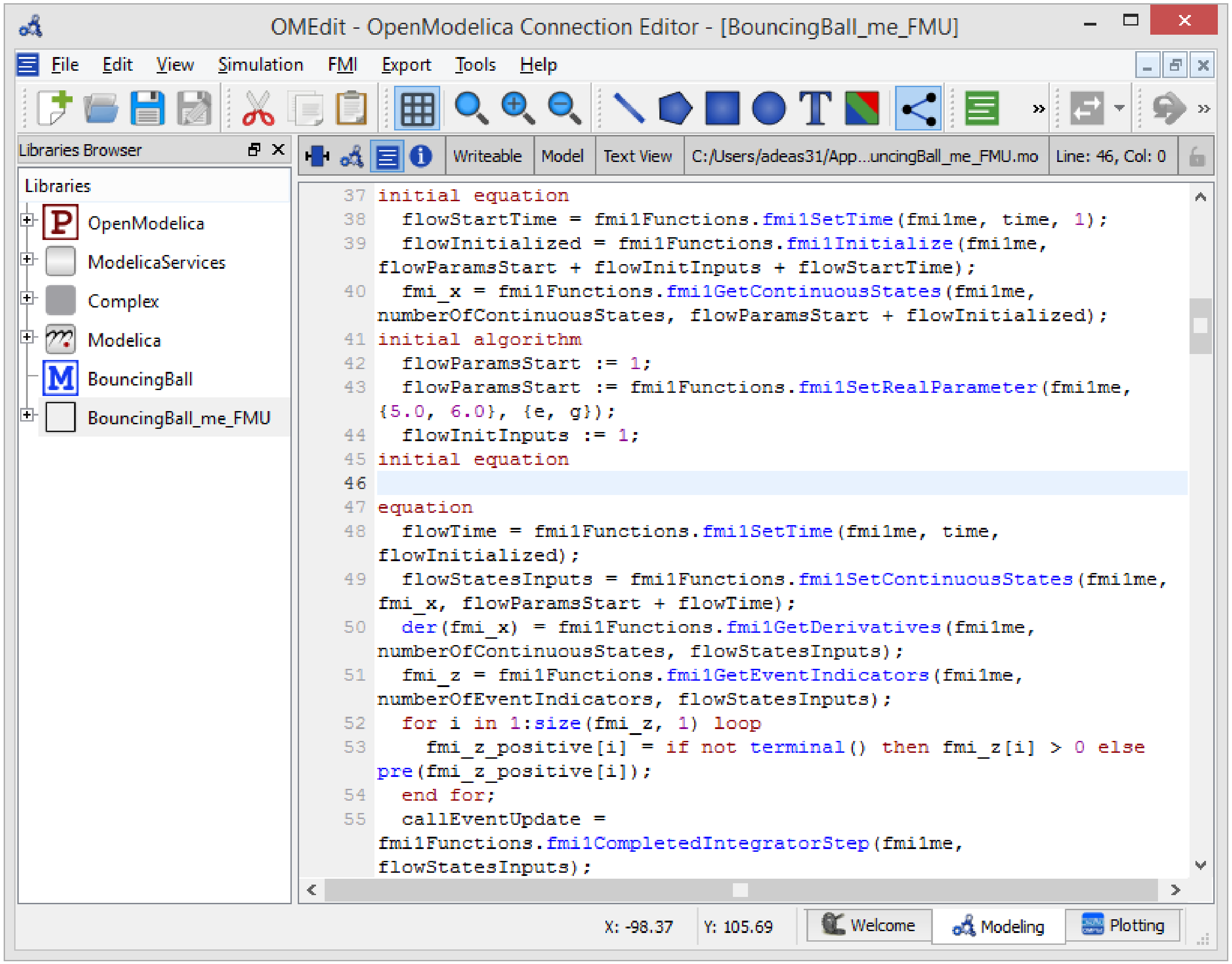
FMI Import in OpenModelica

- OpenModelica scripting API

```
function importFMU
 input String filename "the fmu file name";
 input String workdir = "<default>" "The output directory for imported FMU files.
<default> will put the files to current working directory.";
 input Integer loglevel = 3 "loglevel_nothing=0; loglevel_fatal=1; loglevel_error=2;
       loglevel_warning=3; loglevel_info=4;  loglevel_verbose=5;
loglevel_debug=6";
 input Boolean fullPath = false "When true the full output path is returned otherwise
           only the file name.";
 input Boolean debugLogging = false "When true the FMU's debug output is printed.";
 input Boolean generateInputConnectors = true "When true creates the input connector
              pins.";
 input Boolean generateOutputConnectors = true "When true creates the output
         connector pins.";
 output String generatedFileName "Returns the full path of the generated file.";
end importFMU;
```

- Imports the FMU
- Automatically detects the FMU version and generates a Modelica code to simulate the FMU model

Modelica Code of Imported FMU



FMI Import Process in OpenModelica

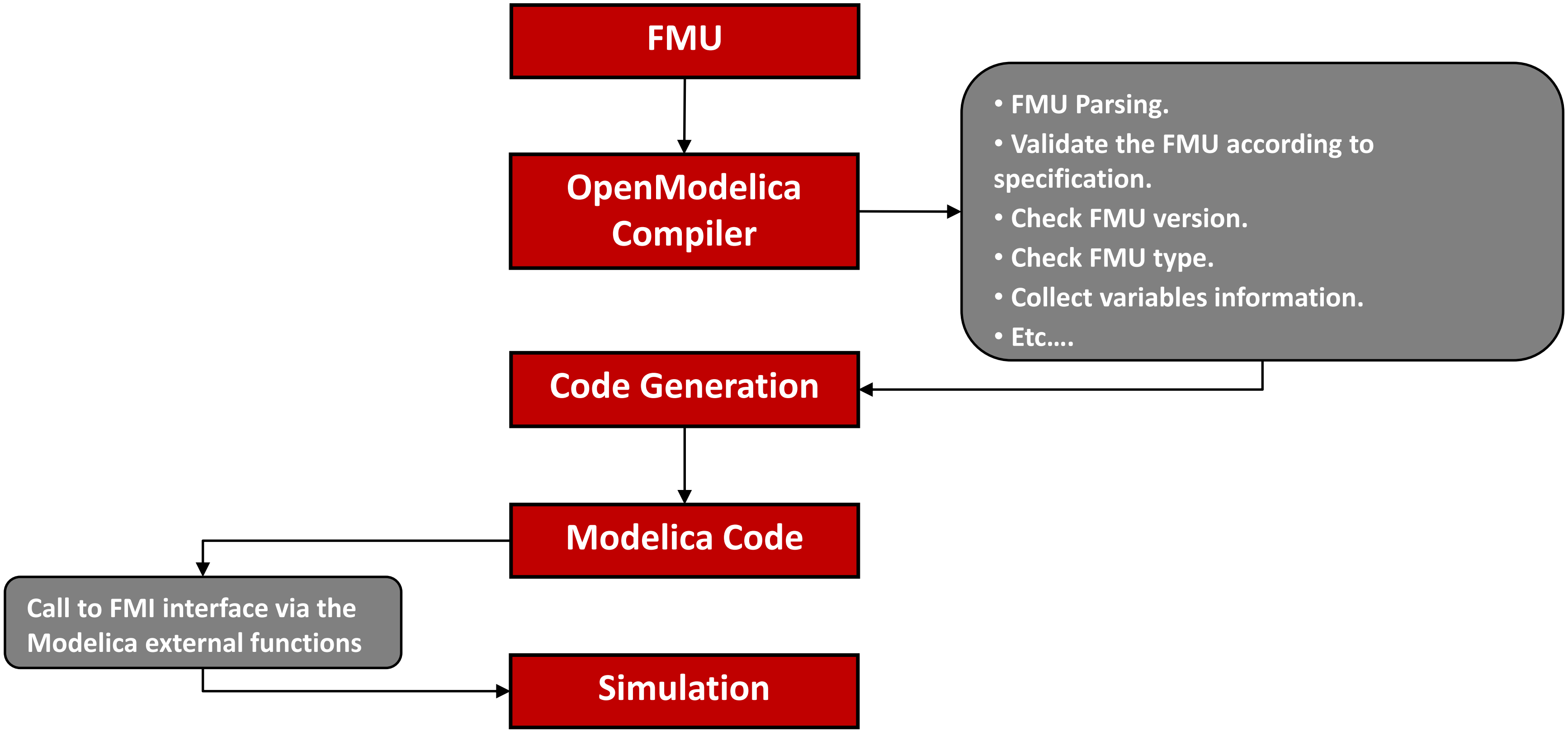


ABB OPTIMAX – OpenModelica Industrial Use Case

- ABB OPTIMAX® provides advanced model based control products for power generation and water utilities.

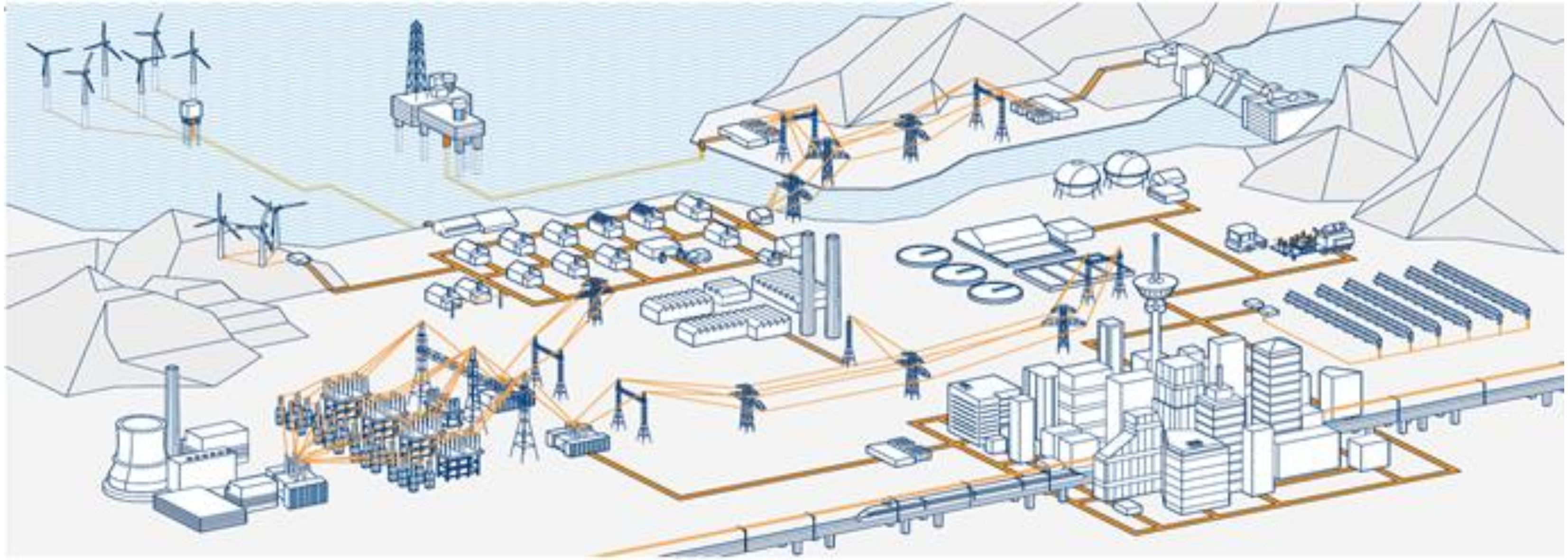


ABB OPTIMAX – OpenModelica Industrial Use Case

- Model-based optimization of power plants using OpenModelica FMI 2.0.
- Plant models are formulated in Modelica and deployed through FMI 2.0
- Link : http://new.abb.com/power-generation/power-plant-optimization