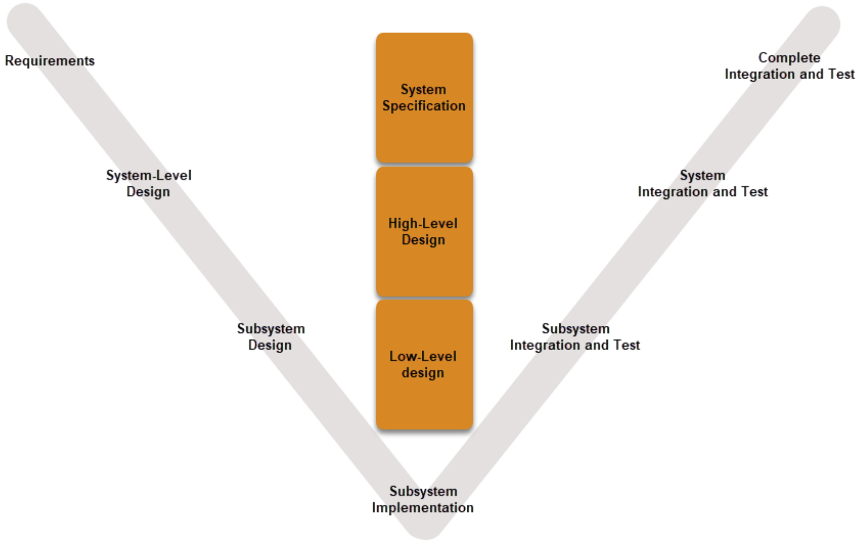
Model-Based Design for Cyber-Physical Systems and the Functional Mockup Interface

Michele Lora, Stefano Centomo and Stefano Spellini

Model-based Design

- Fast and cost-effective development of dynamic systems
  - Control systems
  - Signal processing
  - Communication systems
  - Etc…
- **Model as the center of the development process**
  - **Executable specification** continually refined
  - Purely **top-down** flow
- Advantages:
  - Common design environment
    - Hiding unnecessary details (abstraction)
  - Early validation and testing
  - Design reuse



Mathworks Simulink

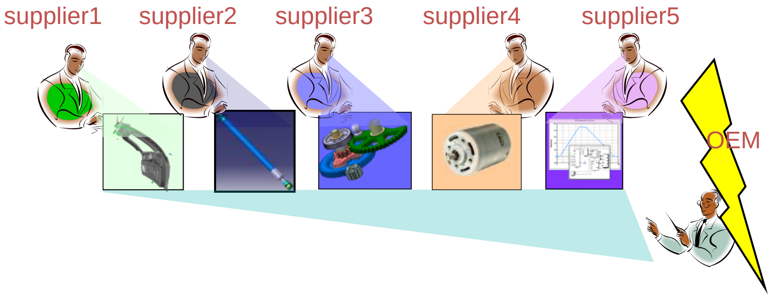
- **Block diagram** environment for
  - **Multidomain simulation**
  - **Model-based Design**

- Main features:
  - Graphical editor
  - Customizable block libraries
  - Solvers for dynamic systems simulation
    - Incorporated within MATLAB:
      MATLAB algorithms for numerical analysis

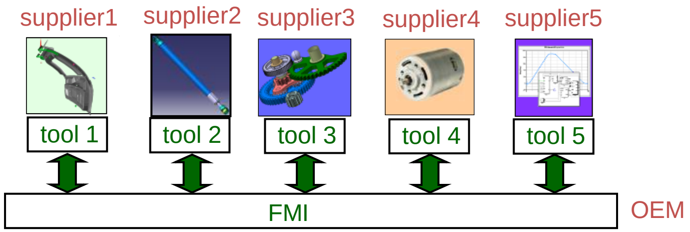
- Further information:
  - https://www.mathworks.com/products/simulink/
  - Gazillion of tutorials and guides online
    - Probably the most used tool in Engineering

Functional Mock-up Interface (FMI) - Motivation

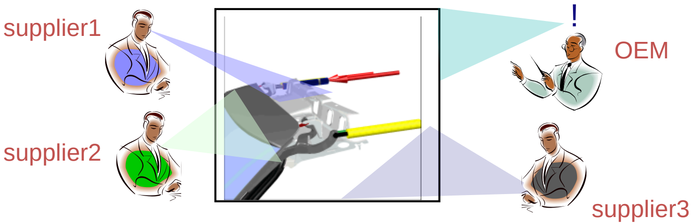
- **Problems / Needs**
  - Component development by supplier
  - Integration by OEM
  - **Many different simulation tools**



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



- **Added Value**
  - Early validation of design
  - Incread process efficiency and quality



FMI – Motivation (2)

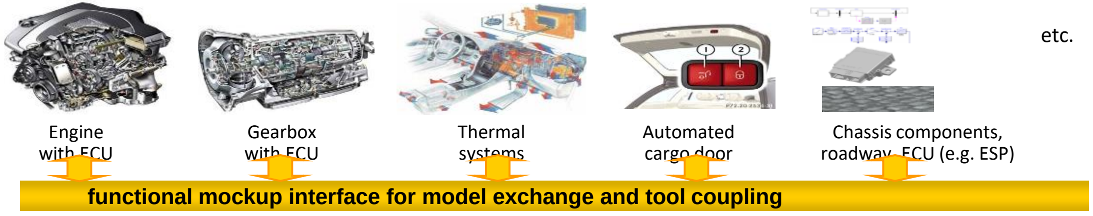
- No standards available for
  - Model interface based on C or binaries
  - Co-simulation between simulation tools

- Lots of proprietary interfaces:
  - Simulink: S-function
  - Modelica: external function, external object interface
  - QTronic Silver: Silver-Module API
  - SimulationX: External Model Interface
  - NI LabVIEW: External Model Interface, Simulation Interface Toolkit
  - Simpack: uforce routines
  - SystemVue: Model Builder
  - Etc…

FMI - Overview

- MODELISAR (2008 – 2011)
  - 29 Partners, Budget 30M €
  - FMI development initiated, organized and headed by Daimler AG
  - Improved Software/Model/Hardware-in-the-Loop Simulation, of physical models from different vendors.
  - Open Standard
  - 14 Automotive Use-Cases to evaluate FMI.

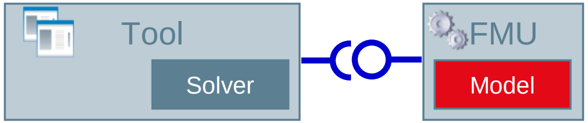
- Modelica Association Project (2011 -)
  - Development, Standardization and promotion
  - Standard developed by 16 partners
  - Extended support to 91 tools



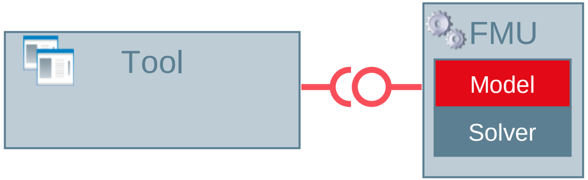
courtesy Daimler

FMI – Main Idea

- FMI for Model Exchange



  - Version 1.0 released in January 2010
  - Version 2.0 released in July 2014
  - Version 2.0.1 released in October 2019
- FMI for Co-Simulation
  - Reuses as much as possible from FMI for Model Exchange standard



  - Version 1.0 released in October 2010
  - Version 2.0 released in July 2014
  - Version 2.0.1 released in October 2019

FMI - Main Idea

- A component which implements the interface is called *Functional Mockup Unit (FMU)*

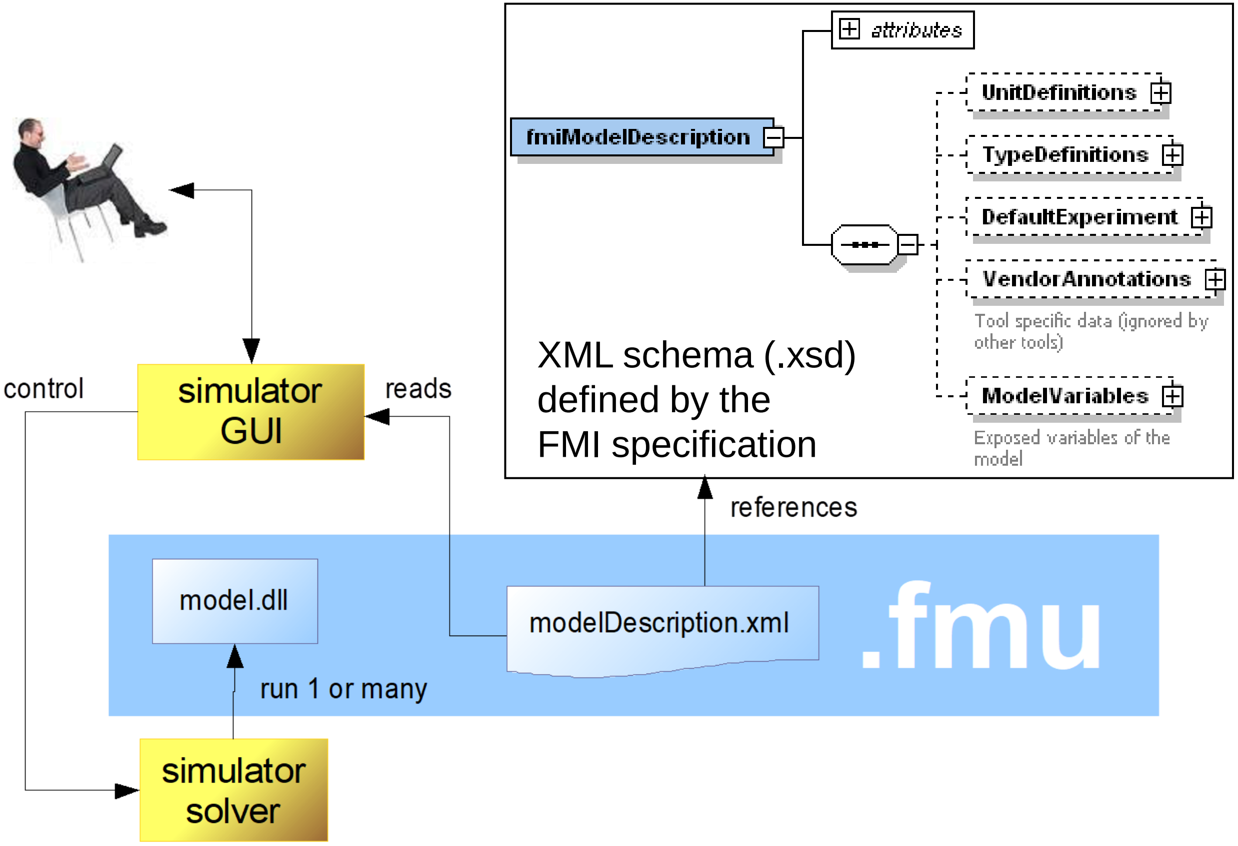
- Separation of
  - Description of interface data (XML file)
  - Functionality (C code or binary)

- 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

- Interface specification:
  - https://www.fmi-standard.org/

Functional Mock-up Unit



FMI XML Schema

- **Information** not needed for execution is stored in one **xml-file**
  - Introduction: information to identify the model
    - `fmiVersion` : Usually 2.0, version of the Standard implemented
    - `modelName` : Name of the system, same as in the implementation
    - `Guid` : Unique identifier for the FMU
    - `Description` : Any text based description for the module
  - Model Variables:
    - Each variable is a `ScalarVariable`
      - name : variable name
      - `valueReference` : variable index in the set of variables sharing the variable type
      - `description` : Any text based descrition for the variable
      - `causality` : input, output or inout
      - …
    - `Type: Boolean, Integer, String or Real`
      - `start`: initial value of the variable.

FMI XML Schema: example (part 1)

```
<?xml version="1.0" encoding="UTF-8"?>
<fmiModelDescription
fmiVersion="2.0“
modelName="controller_system“
guid="{controller-a5cf1fbd-89ca-495b-b5ff-ec53a96c6173}"
description="Controller for the Water Tank system”
generationTool="Manual"
generationDateAndTime="None“
variableNamingConvention="structured“
numberOfEventIndicators="0">
```

FMI XML Schema: example (part 2)

```
<CoSimulation
modelIdentifier="controller_system"
canHandleVariableCommunicationStepSize="false"/>
<LogCategories>
<Category name="logEvents" />
<Category name="logSingularLinearSystems" />
<Category name="logNonlinearSystems" />
<Category name="logDynamicStateSelection" />
<Category name="logStatusWarning" />
<Category name="logStatusDiscard" />
<Category name="logStatusError" />
<Category name="logStatusFatal" />
<Category name="logStatusPending" />
<Category name="logAll" />
<Category name="logFmi2Call" />
</LogCategories>
```

FMI XML Schema: example (part 3)

```
<ModelVariables>
<ScalarVariable name="HIGH" valueReference="0" description="bool" causality="input"
variability="discrete" initial="approx">
<Boolean start="false"/>
</ScalarVariable>
<ScalarVariable name="LOW" valueReference="1" description="bool" causality="input"
variability="discrete" initial="approx">
<Boolean start ="false" />
</ScalarVariable>
…
<ScalarVariable name="threshold" valueReference="0" description="double"
causality="output" variability="discrete" initial="approx">
<Real start ="0.0" />
</ScalarVariable>
</ModelVariables>
…
</fmiModelDescription>
```

C-interface

- Three C-header files
  - fmi2TypesPlatform.h
    - **Platform dependent definitions** (basic types)

```c
/* Platform (combination of machine, compiler, operating system) */
#define fmiModelTypesPlatform "standard32"

/* Type definitions of variables passed as arguments */
    typedef void*           fmiComponent;
    typedef unsigned int    fmiValueReference;
    typedef double          fmiReal    ;
    typedef int             fmiInteger;
    typedef char            fmiBoolean;
    typedef const char*     fmiString ;

/* Values for fmiBoolean  */
#define fmiTrue  1
#define fmiFalse 0

/* Undefined value for fmiValueReference (largest unsigned int value) */
#define fmiUndefinedValueReference (fmiValueReference)(-1)
```

  - fmi2FunctionTypes.h & fmi2Functions.h
    - Core functions
    - Utility functions

C-interface (Initial phase)

- Instantiation
  - `fmi2Component fmi2Instantiate(…)`
    - `fmiComponent` is a parameter of the other functions
    - Opaque `void *` for importing tool
    - Used by FMU to hold any necessary information
  - Allocate the memory necessary for the data structure
    - One array for each type of variables
    - C struct containing all the information and data of the model

- Initialization
  - `fmi2Status fmi2SetupExperiment(…)`
    - `fmiComponent` as a parameter
    - Model initialization (default values, time, etc…)

C- interface (Variables access)

- Set and Get functions for each type
  - In this slide: TYPE = Boolean, Real, String, Integer
- Write an input variable
  - ```
    fmi2Status fmi2SetTYPE(fmi2Component c,
           const fmi2ValueReference vr[],
           const fmiTYPE value[])
    ```
  - Set the value of the variables in the model. Assigns the value contained in value[i] to the variable referenced by vr[i]
- Read an output variable
  - ```
    fmi2Status fmiGetTYPE(fmi2Component c,
           const fmi2ValueReference vr[],
           const fmiTYPE value[])
    ```
  - Get the value of the variables in the model. Assigns the value of the variable referenced by vr[i] to value[i]
- Identification by **`valueReference`**, defined in the XML description file for each variable
  - vr[] is the array containing the variable of type: TYPE

C-interface (Execution)

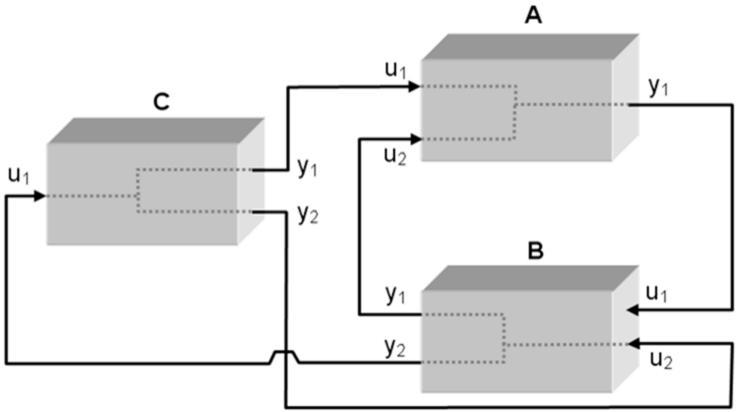
- The execution is carried on by a unique function: `fmi2DoStep`
  - Takes care of performing computation
  - Manipulate variables in the arrays
  - **Advances** the model internal **time**
    - Parameters: start time, stop time
  - Complete freedom on its internal implementation

- E.g.,

```
fmi2Status fmi2DoStep(fmi2Component c,
    fmi2Real currentCommunicationPoint,
    fmi2Real communicationStepSize,
    fmi2Boolean noSetFMUStatePriorToCurrentPoint)
{
   ModelInstance *comp = (ModelInstance *)c;
   controller_implementation( comp );
   comp->time+=communicationStepSize;
   return fmi2OK;
}
```

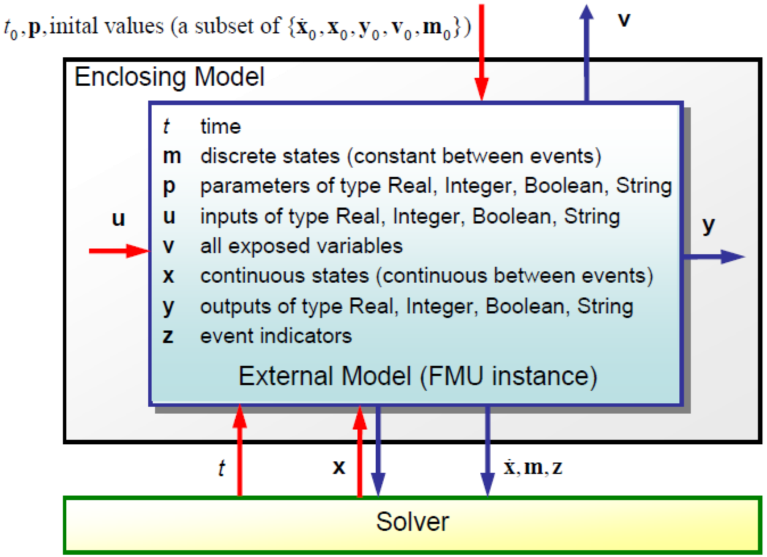
FMI for Model Exchange

- Import and export of input/output blocks (FMU – Functional Mock-up Unit)
- described by
  - differential-, algebraic-, discrete equations,
  - with time-, state, and step-events
- FMU can be large (e.g. 100000 variables)
- FMU can be used in an embedded system (small overhead)
- FMUs can be connected together



FMI for Model Exchange

- Signals of an FMU



For example: 10 input/output signals (u/y) for connection and 100000 internal variables (v) for plotting

Co-Simulation

- Definition:
  - Coupling of several simulation tools
  - Each tool treats one part of a modular coupled problem
  - Data exchange is restricted to discrete communication points
  - Subsystems are solved independently between communication points

- Motivation:
  - Simulation of heterogeneous systems
  - Partitioning and parallelization of large systems
  - Multirate integration
  - Hardware-in-the-loop simulation

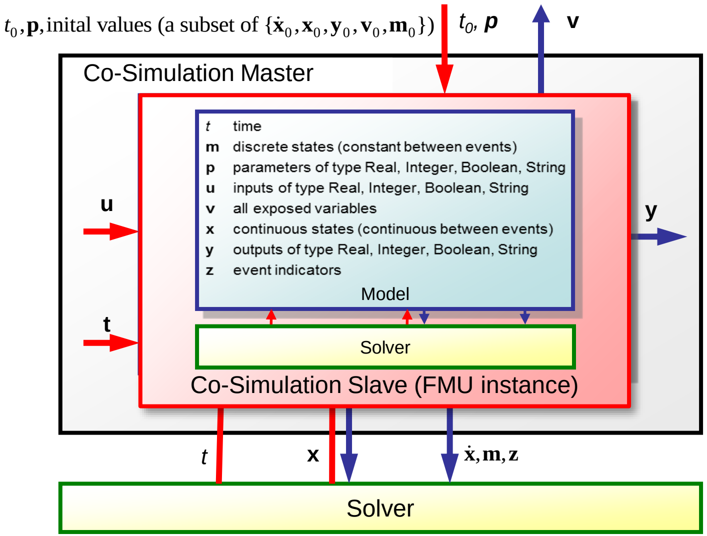
FMI for Co-Simulation

- Master/slave architecture
- Considers different capabilities of simulation tools
- 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

- Signals of an FMU for Co-Simulation



  - Inputs, outputs, and parameters, status information
  - Derivatives of inputs, outputs w.r.t. time can be set/retreived for supporting of higher order approximation

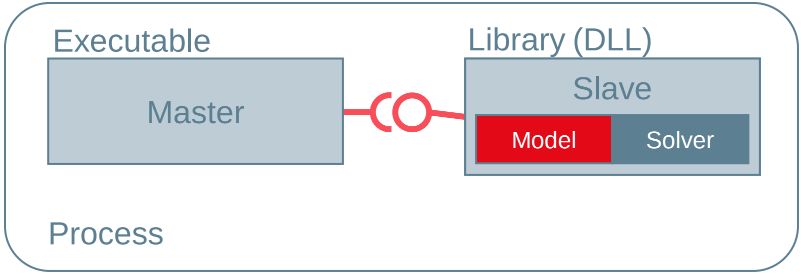
FMI for Co-Simulation C-Interface

- Execution of a time step:
```
fmi2Status fmi2DoStep(fmi2Component c,
 fmi2Real currentCommunicationPoint,
 fmi2Real communicationStepSize,
 fmi2Boolean newStep)
```

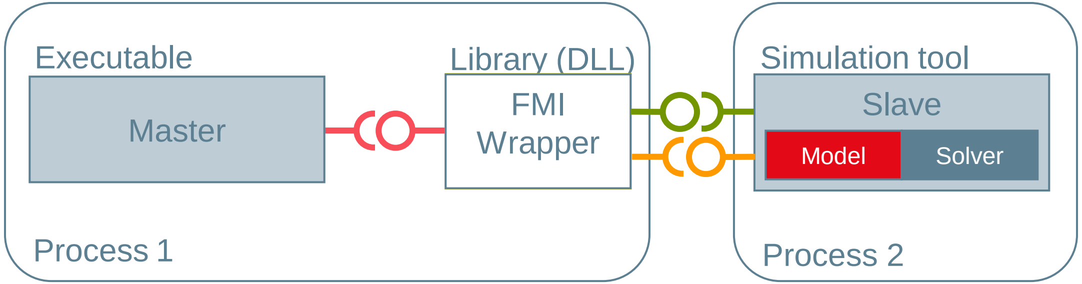
- `communicationStepSize` can be zero in case of event iteration
- `newStep = fmi2True` if last step was accepted by the master
- It depends on the capabilities of the slave which parameter constellations and calling sequences are allowed
- Depending on internal state of the slave and the function parameters, slave can decide which action is to be done before the computation
- Return values are `fmi2OK, fmi2Discard, fmi2Error, fmi2Pending`
- Asynchronous execution is possible

FMI for Co-Simulation Use Case

- Co-Simulation stand alone:

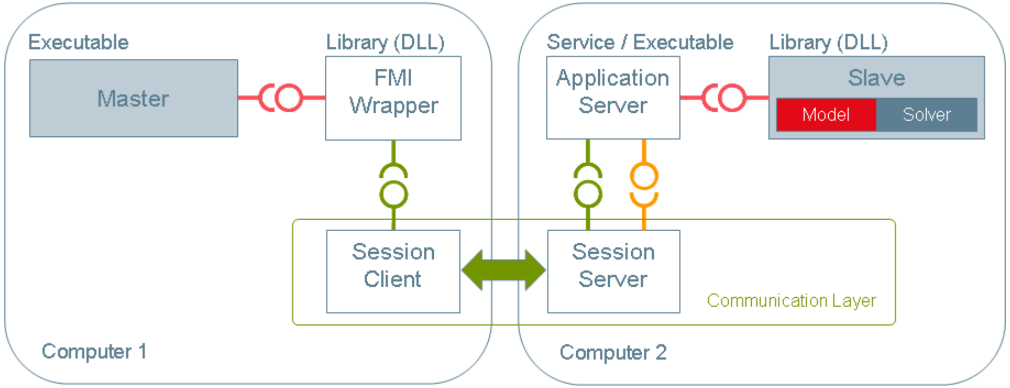


- Co-Simulation tool:



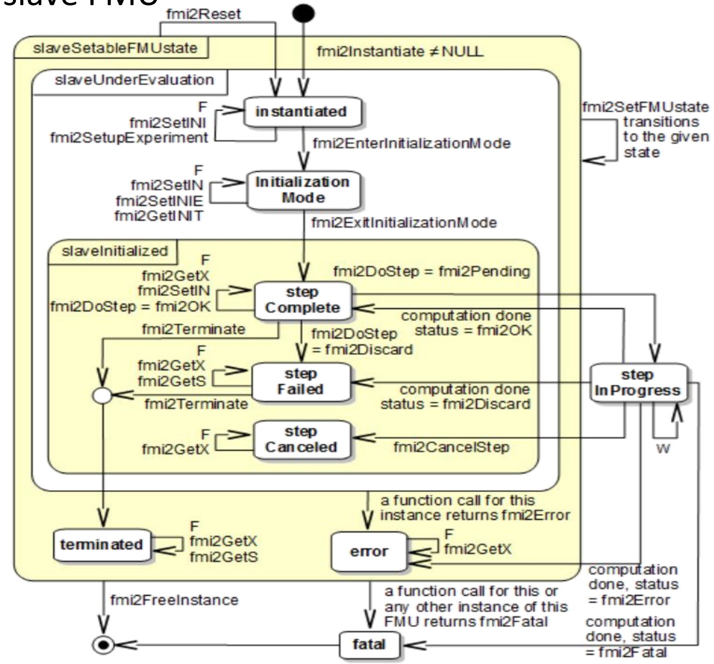
FMI for Co-Simulation Use Case

- Distributed co-simulation scenario



Master Algorithm

- Necessary to co-simulate a slave FMU



- Some **constraints**:
  - No Get after Set with no DoStep in between

FMI and Simulink into Action

- Download the archive from the e-learning and untar it
  - `$> tar xzfv 10_sources.tar.gz`
  - `$> cd 10_sources`
- Directories:
  - checkers: contains the FMUChecker implementation
    - Linux 64 bit executables within zip files
    - Launch it without parameters **for help**
      - `./fmuCheck.linux64 <uri of fmu> <-k <cs> >`
      - `-k` option checks only the XML file, `cs` for co-simulation
  - Models: Simulink model of the water tank system
    - No_FMU: only Simulink (continuous time and Stateflow)
    - FMU: Simulink model using an FMU to implement the controller
      - MDL file containing the Simulink model
      - Source code of the FMU
    - FMU_Assignment
      - Skeleton of folders and files for the assignment

Compile the FMU

- Compile the shared library (from the source_code directory):
  - Linux & WSL

```
$> gcc -shared -fPIC -Iinclude/ src/controller.c
     -o fmu/binaries/linux64/controller_system.so
```

  - OSX **(gcc or clang)**

```
$> gcc -shared -Iinclude/ src/controller.c
     -o fmu/binaries/darwin64/controller_system.dylib
```

  - Windows

```
Use Visual Studio to compile the .dll (you need a win32 compiler)
Copy the .dll into fmu/binaries/win64/controller_system.dll
```

- Create the FMU

```
$> cd fmu
$> zip -r controller_system.fmu modelDescription.xml
          binaries/
```

Useful references

- FMI-Standard web site
  - Home page: https://www.fmi-standard.org/
  - List of compatible tools: https://www.fmi-standard.org/tools
  - List of publications: https://www.fmi-standard.org/literature

- Suggested readings:
  - Blochwitz, Torsten, et al. "*The functional mockup interface for tool independent exchange of simulation models*" *Proc. of the 8th International Modelica Conference;* Linköping University Electronic Press, 2011.

- Suggested tools:
  - FMU SDK by QTronic: http://www.qtronic.de/en/fmusdk.html
  - FMU (Web)Check: https://fmu-check.herokuapp.com/
  - PyFMI: https://pypi.python.org/pypi/PyFMI
    - FMI Library: http://www.jmodelica.org/FMILibrary
- **Suggested FMU examples:**
  - https://github.com/modelica/Reference-FMUs

Lecture Assignment

- Create an FMU implementing the Multiplication Algorithm
  - Implement the fixed point arithmentic
  - Preserve the accuracy as in your SystemC module
    - Use the uint8_t and uint16_t types defined in stdint.h
    - Use masks to fix the bit span of the interface variables
- Adapt the Controller FMU to use the Multiplication FMU
  - The connections between Controller and Multiplier will be Integer
  - Try it on the plant:
    - Attach the new controller and multiplier to the Simulink model