

# Tutorial

FMI for Composite Modelling, Co-Simulation and Model Exchange

Lennart Ochel, Robert Braun

13<sup>th</sup> MODPROD Workshop, February 5-6, 2019

# Outline

- Installing OpenModelica (v1.13.2 or later)
- Introduction to FMI
- FMU Export (OpenModelica/OMEdit)
- Composite Modelling and Simulation (OMSimulator)
- Exercises

# Installing OpenModelica

- v1.13.2 or later (e.g. nightly build)
  - Windows: <https://openmodelica.org/download/download-windows>
  - Linux: <https://openmodelica.org/download/download-linux>
  - MacOS: <https://openmodelica.org/download/download-mac>
- Documentation
  - <https://www.openmodelica.org/doc/OpenModelicaUsersGuide/latest/>
  - <https://www.openmodelica.org/doc/OMSimulator/html/>
- Tickets (feature request & bug report)
  - <https://trac.openmodelica.org/OpenModelica/newticket>
  - <https://github.com/OpenModelica/OMSimulator/issues/new/choose>

# Introduction to FMI

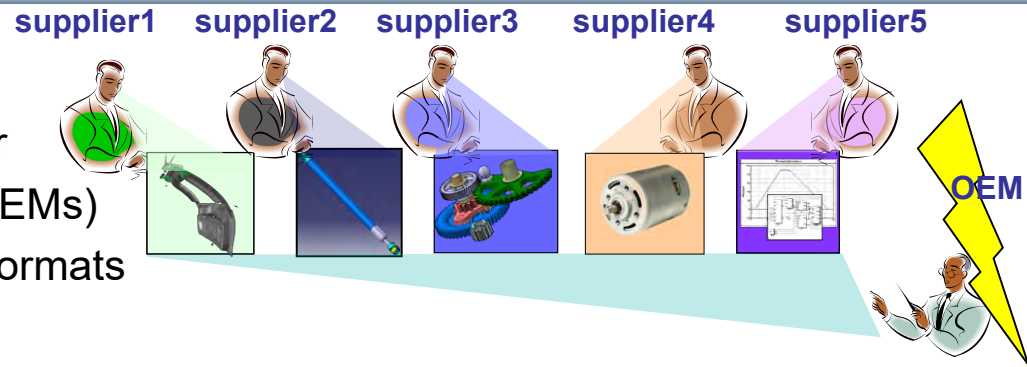
# FMI – Motivation 1

- 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](http://www.fmi-standard.org)

# FMI – Motivation 2

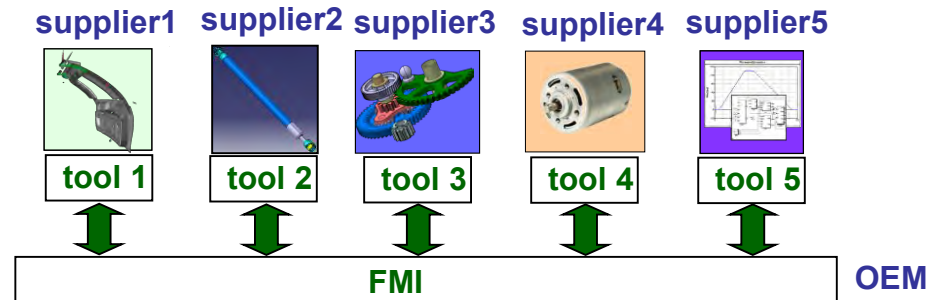
## Problems / Needs

- Component development by supplier
- Integration by product integrators (OEMs)
- Many different simulation tools and formats



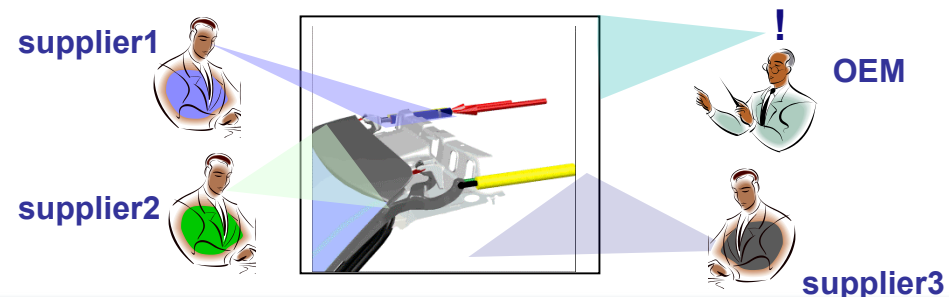
## Solution

- Reuse of supplier models by Product integrator companies (OEMs):
  - Binary (DLL) (model import) and/or
  - Tool coupling (co-simulation)
- Protection of model IP (Intellectual Property) of supplier since binary models

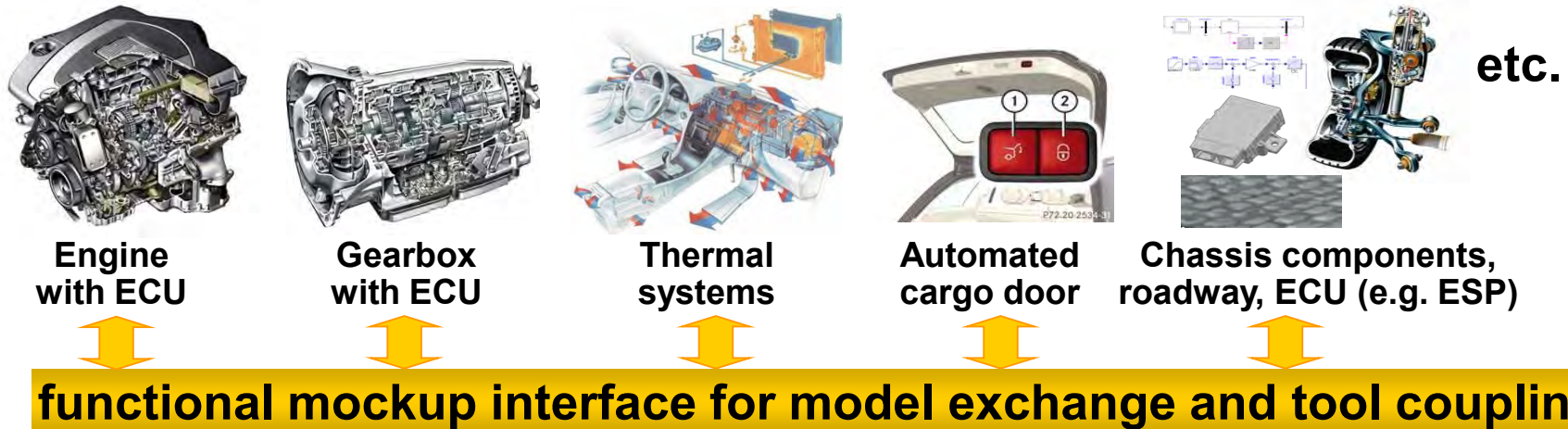


## Added Value

- Early validation of design
- Increased process efficiency and quality



# Functional Mock-up Interface (FMI) – Overview



- 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)
- **> 120 tools** supporting it (<https://www.fmi-standard.org/tools> )

# Exported Model in (Functional Mockup Unit) FMU Form

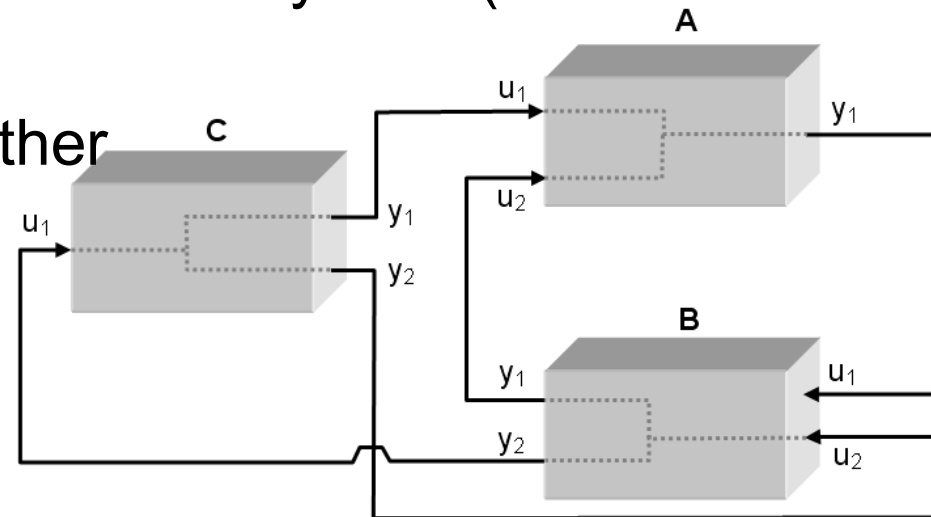
A model is distributed in one zip-file that contains several files:

- **XML** file of static model information
- **C code or shared library** with model equations converted into **causal form**
- Further **data**, documentation, maps, icon



# Functional Mockup Units

- Import and export of input/output blocks – **Functional Mock-Up Units – FMUs**
- 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 (= complicated data structure) with their preferred language (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, ...)

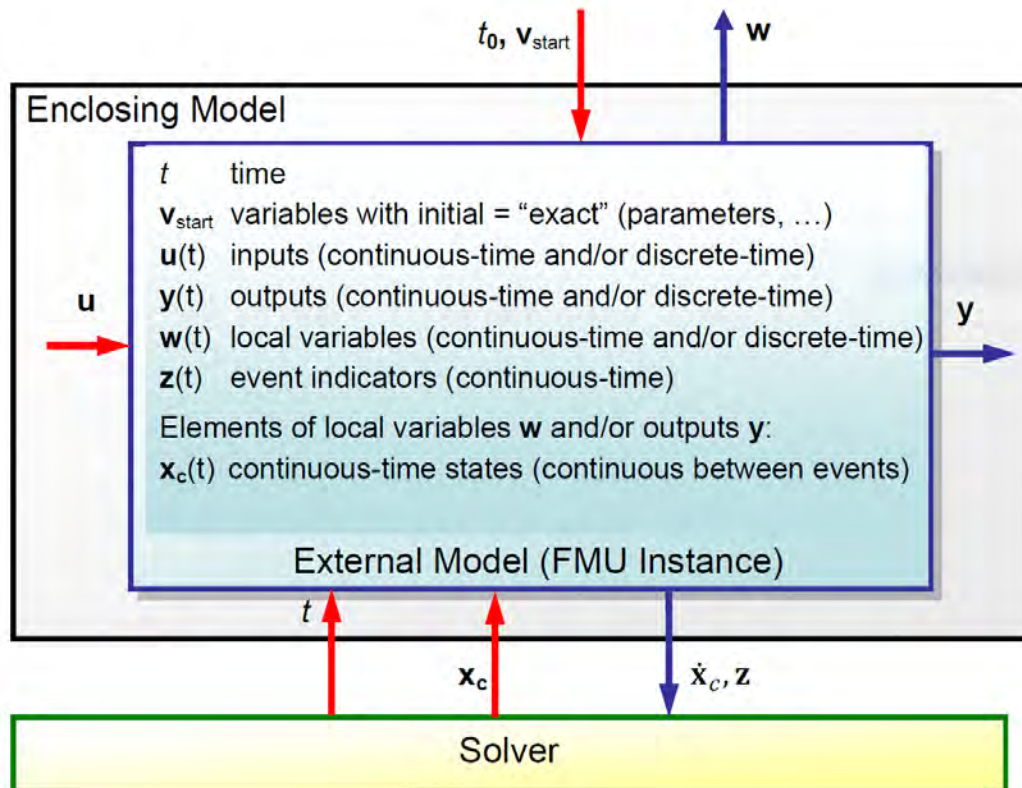
- **Resources**

- Documentation (html files)
- Model icon (bitmap file)
- Maps and tables (read by model during initialization)

---

# FMI for Model-Exchange

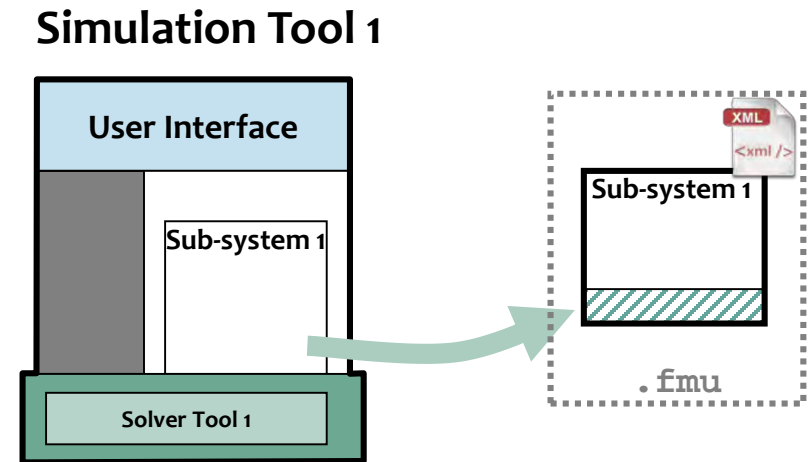
# FMI for Model Exchange Export



Functional Mock-up Interface for Model Exchange and Co-Simulation, Section 3

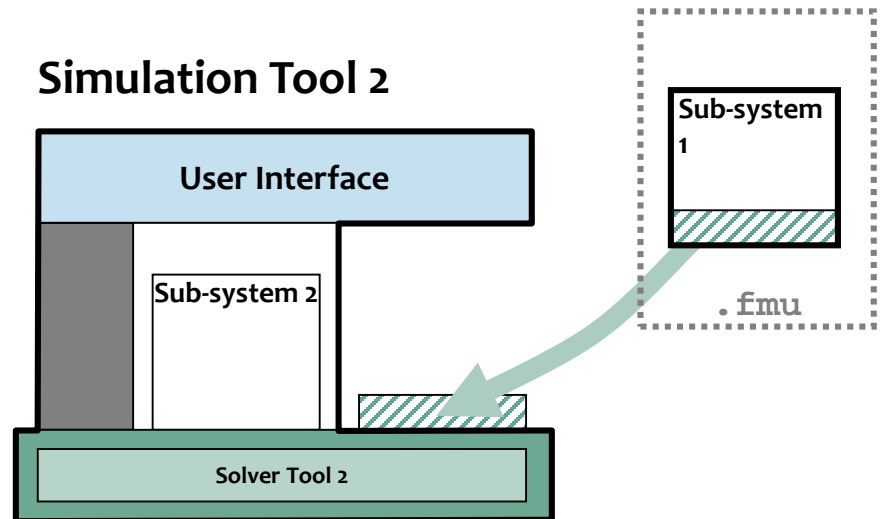
# FMI for Model Exchange Export

- **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

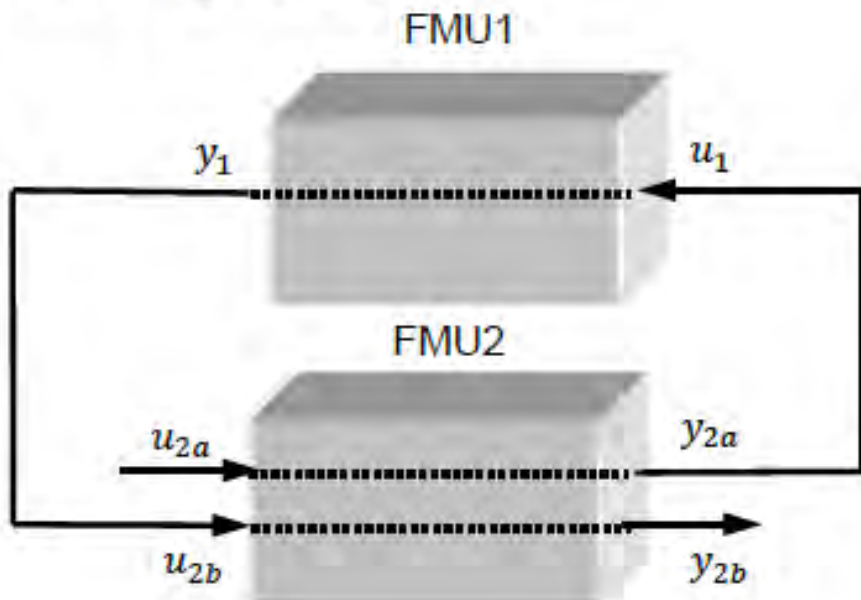
- **Import: Subsystem model is imported** into simulation system for **system simulation**
  - Reading FMU-archive
    - model information from xml-file
    - connecting subsystem variables
    - executable model equations (dll)
    - running system simulation



# Handling of Algebraic Loops

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

artificial algebraic loop



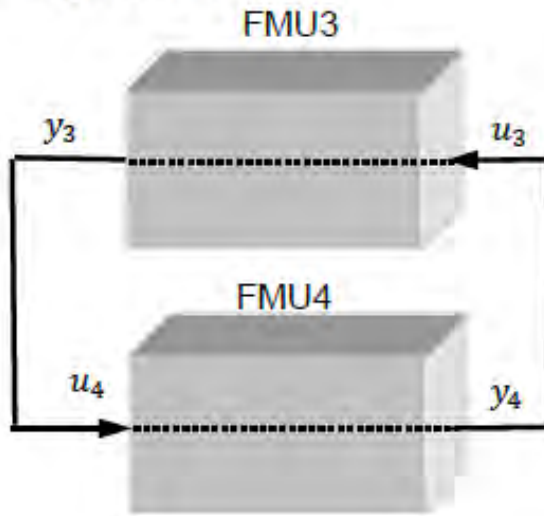
*sequential calling sequence:*

```
fmiSetXXX(m2, < u2a >, ...)  
y2a := fmiGetXXX(m2, ...)  
fmiSetXXX(m1, < u1 := y2a >, ...)  
y1 := fmiGetXXX(m1, ...)  
fmiSetXXX(m2, < u2b := y1 >, ...)  
y2b := fmiGetXXX(m2, ...)
```

# Handling of Algebraic Loops

- Iterative method
- In each iteration  $u_4$  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  $u_4$  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, <  $u_4$  >, ...)
```

```
 $y_4 := \text{fmiGetXXX}(m4, ...)$ 
```

```
fmiSetXXX(m3, <  $u_3 := y_4$  >, ...)
```

```
 $y_3 := \text{fmiGetXXX}(m3, ...)$ 
```

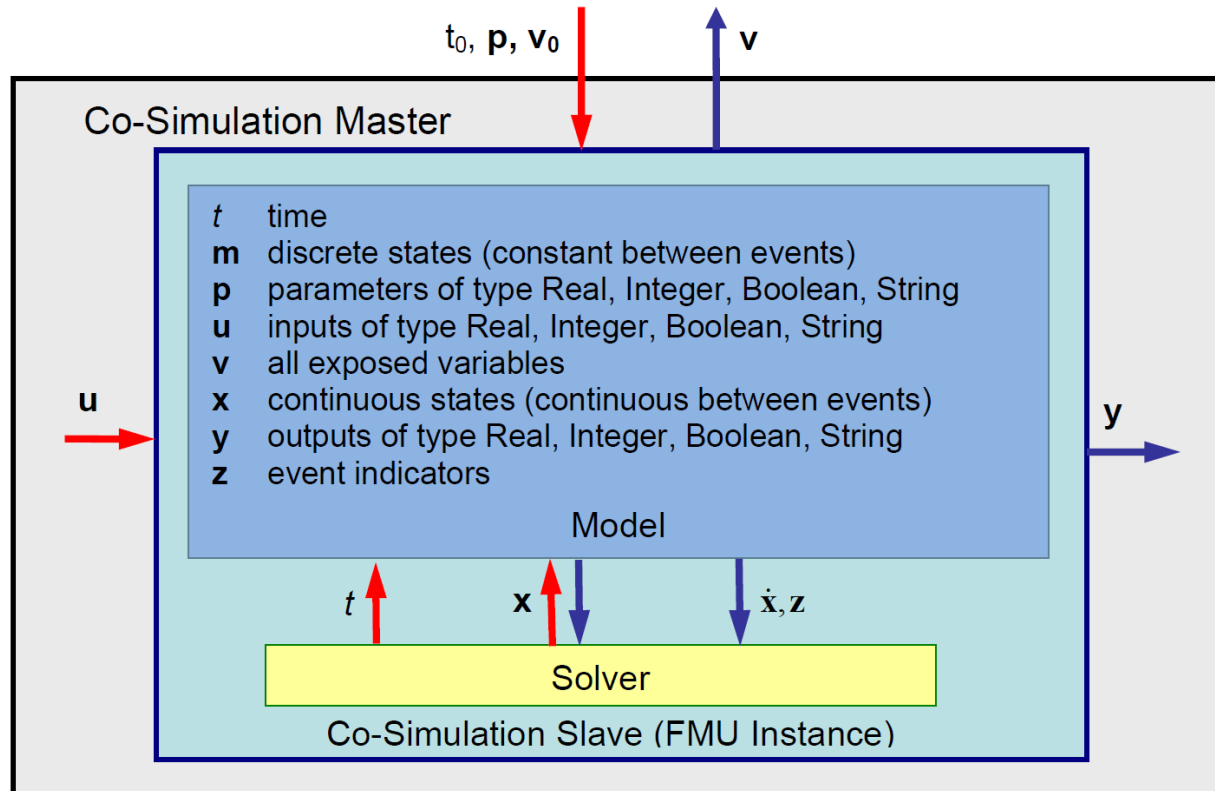
```
residue :=  $u_4 - y_3$ 
```



---

# FMI for Co-Simulation

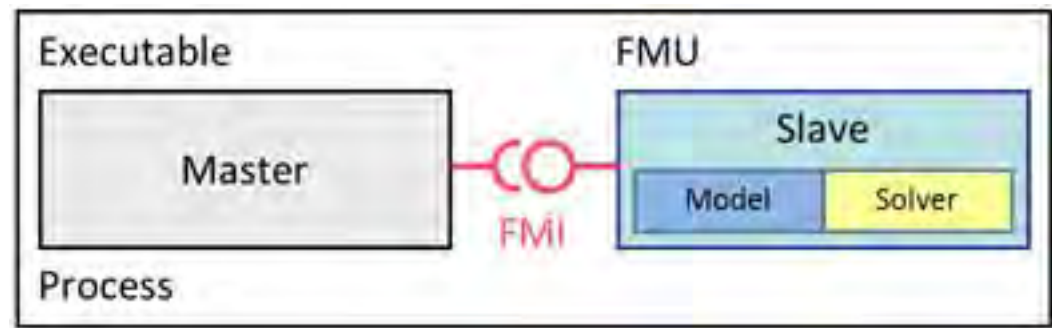
# FMI for Co-Simulation



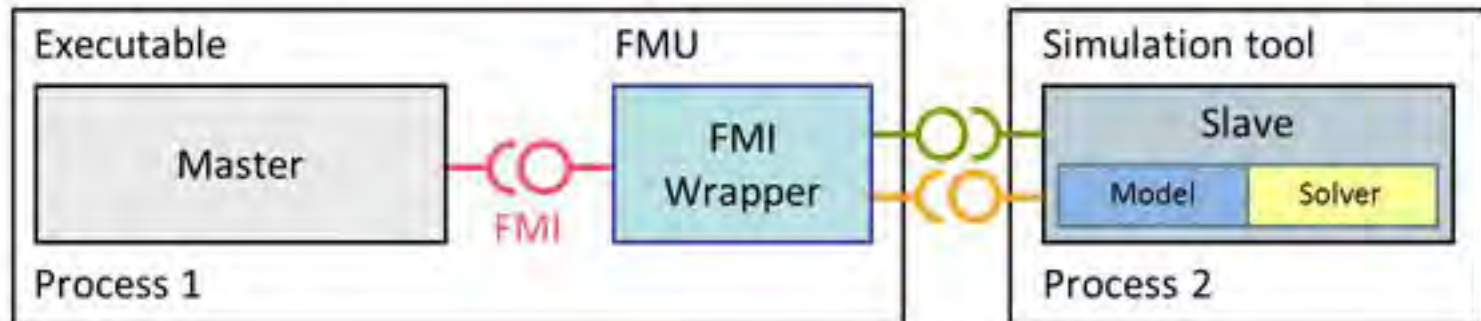
Functional Mock-up Interface for Model Exchange and Co-Simulation, Section 4

# FMI for Co-Simulation

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



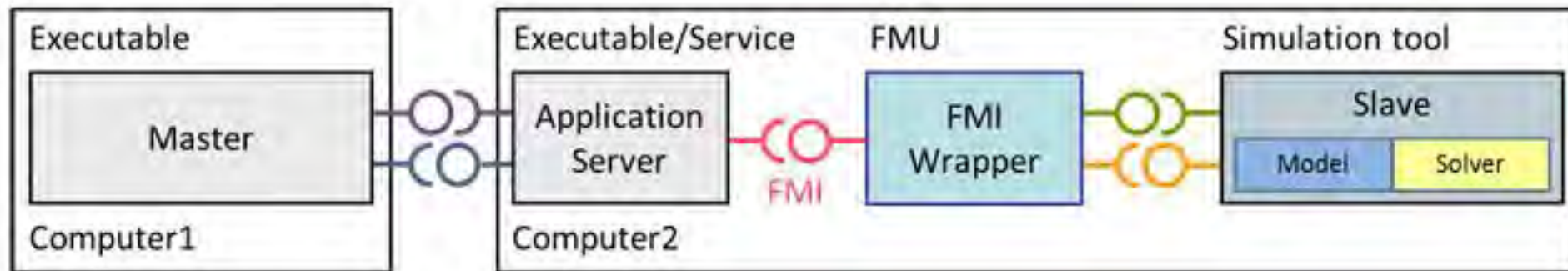
- And for coupling of simulation tools



# FMI for Co-Simulation

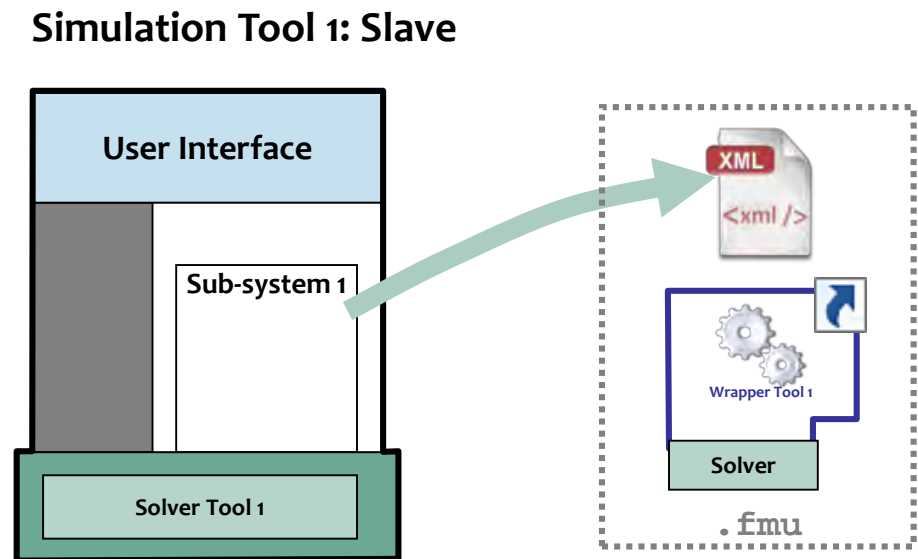
- Distributed Co-Simulation Scenario

- 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

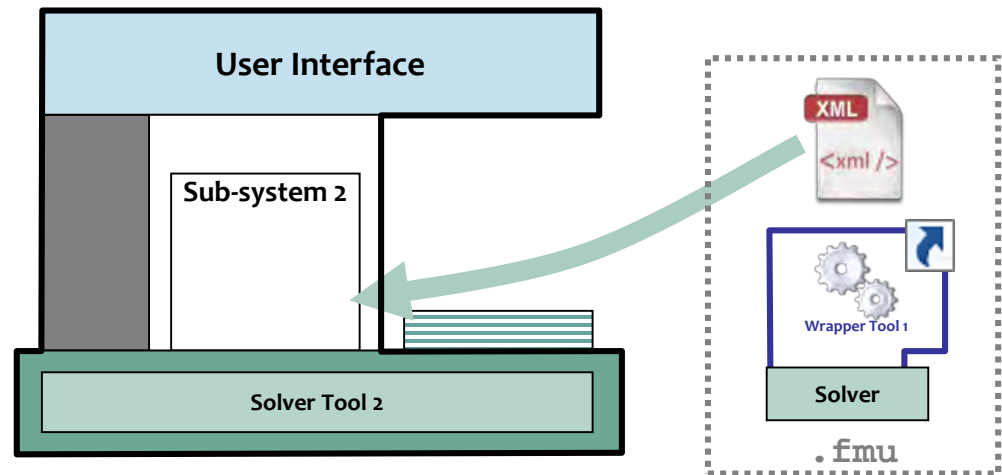
- **Export: Subsystem description** is exported from its simulation tool
  - Preparation as FMU-archive containing
    - model description (xml-file), describes also solver/tool capabilities
    - reference to executable dll-file as, wrapper which provides a tool specific implementation of the co-simulation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



# FMI for Co-Simulation Import Stand-alone

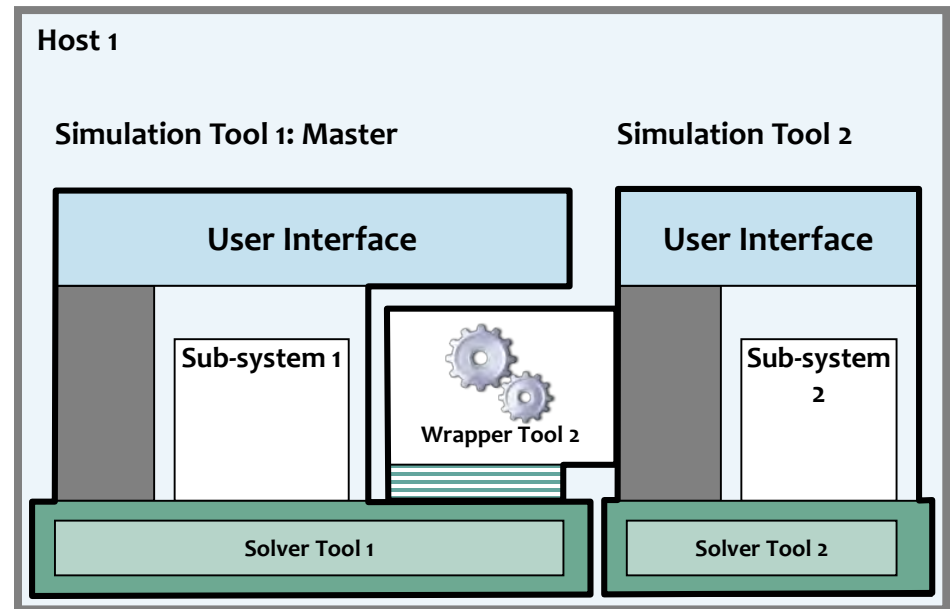
- **Import: Subsystem description** is imported into simulation system for system simulation
  - Reading FMU-archive
    - model information from xml-file
    - connecting subsystem variables

Simulation Tool 2: Master



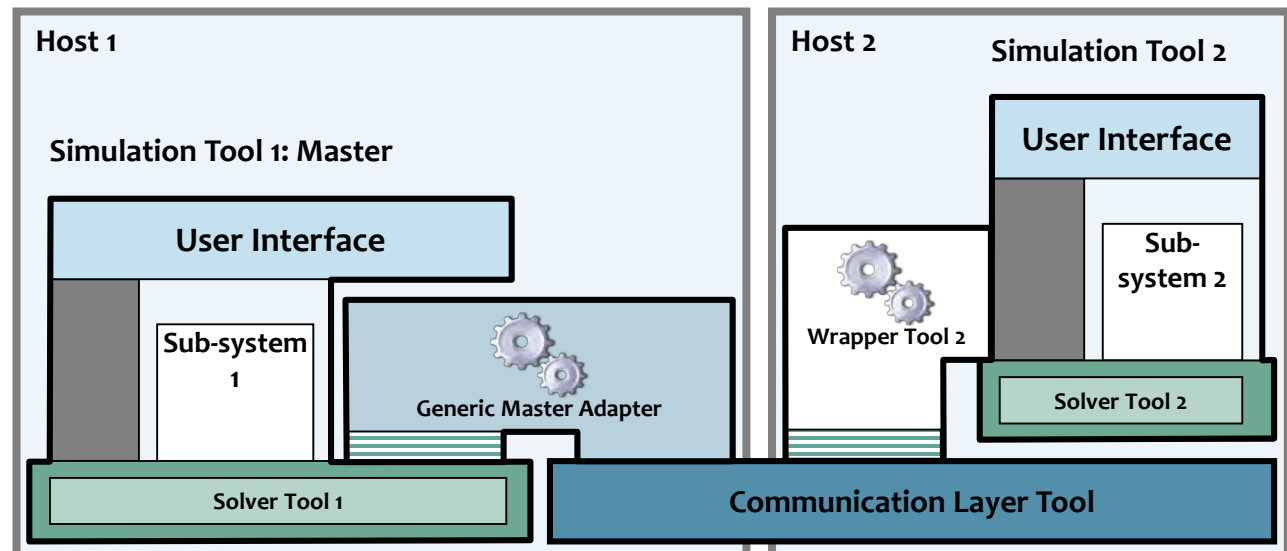
# FMI for Co-Simulation Tool coupling

- **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



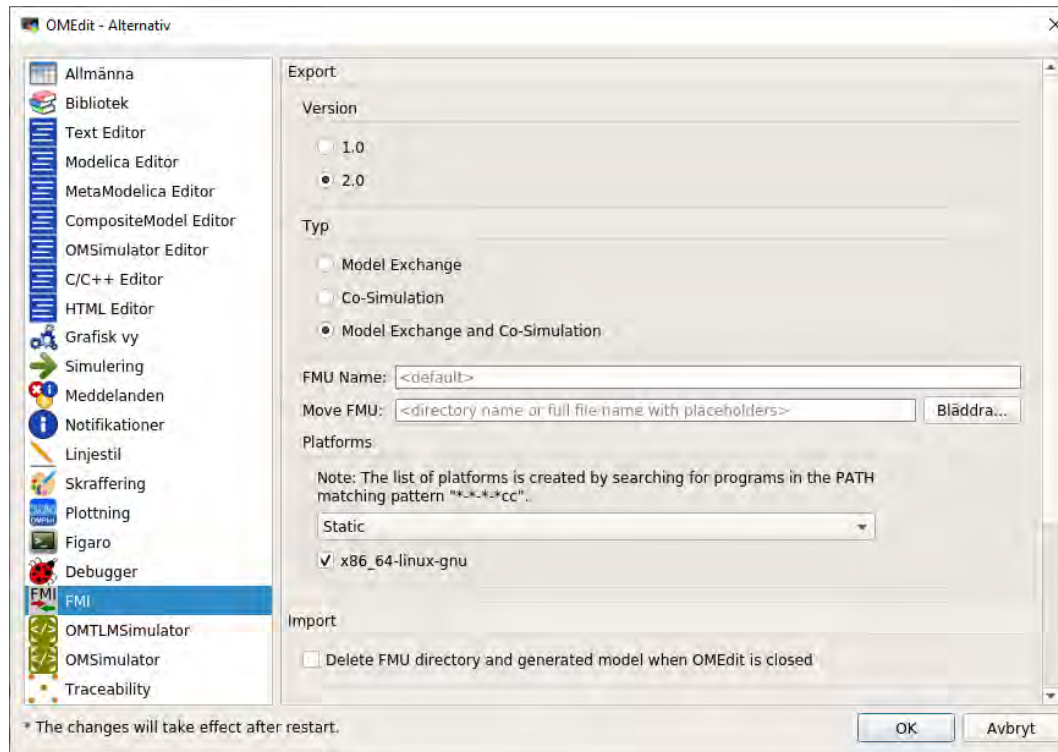


# FMU Export

OpenModelica/OMEdit

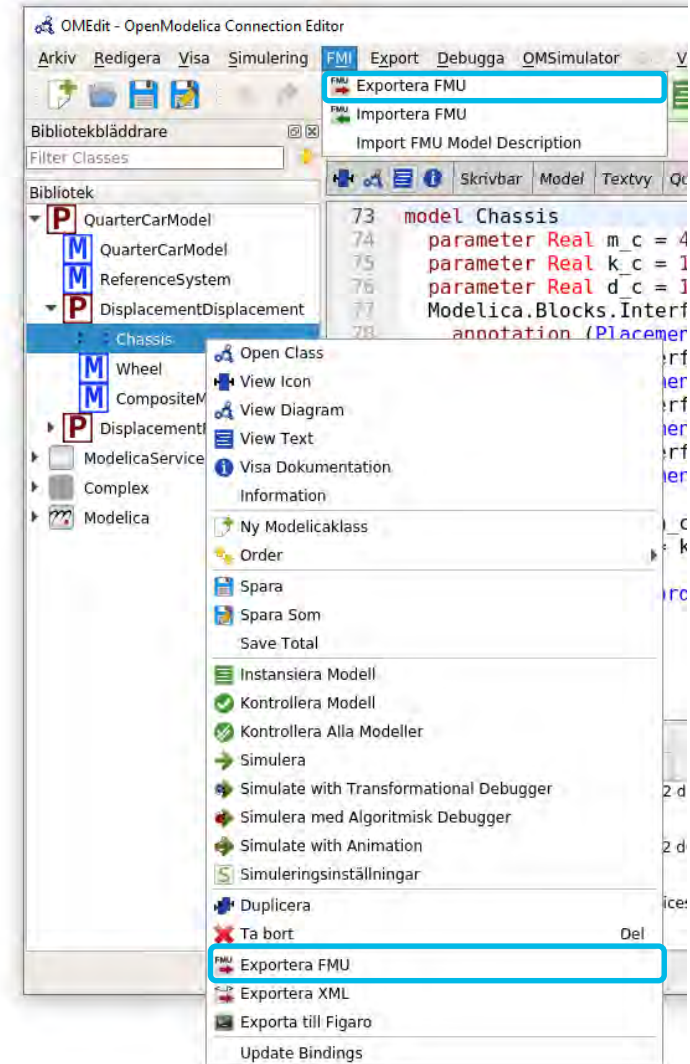
# FMU Export

- Check FMI setting in OMEdit (Tools->Options)



# FMU Export

- Check the FMI settings
- Open/Create a Modelica model
- Select "Export FMU"

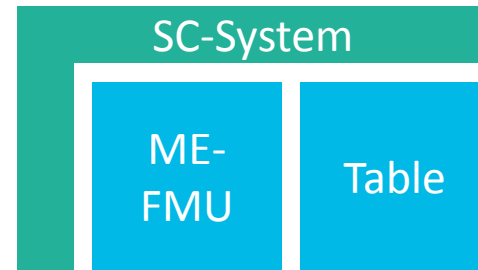


# Composite Modelling and Simulation

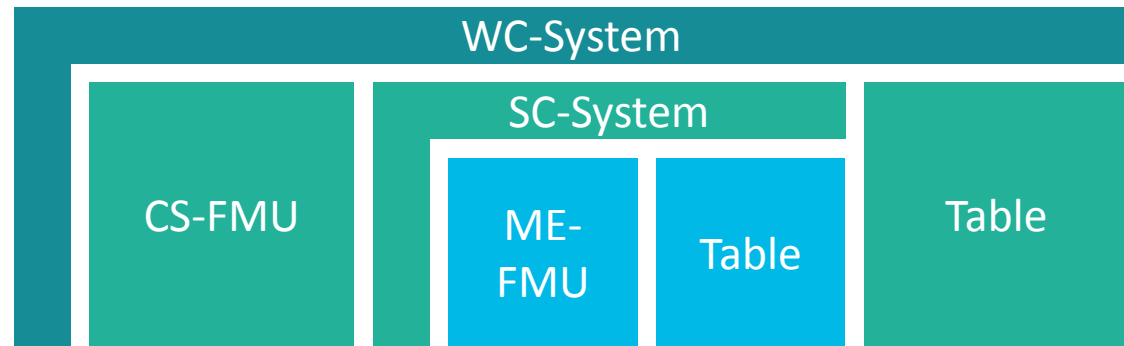
OMSimulator

# Composite Model Structure (I)

- Strongly Connected System
  - direct communication schema
- Detecting and handling algebraic loops
- Integration methods
  - Explicit euler
  - Cvode



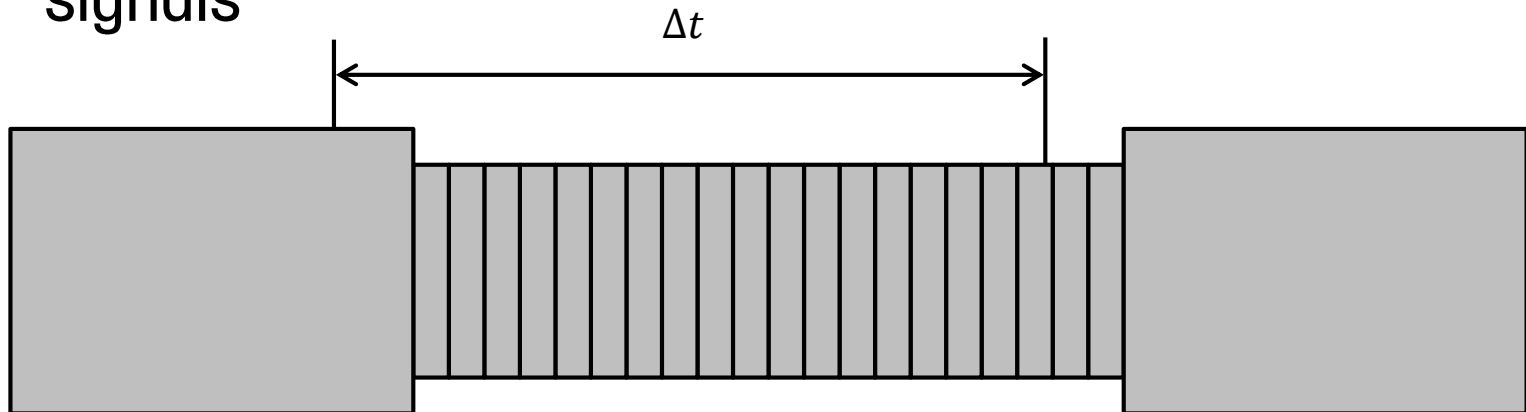
# Composite Model Structure (II)



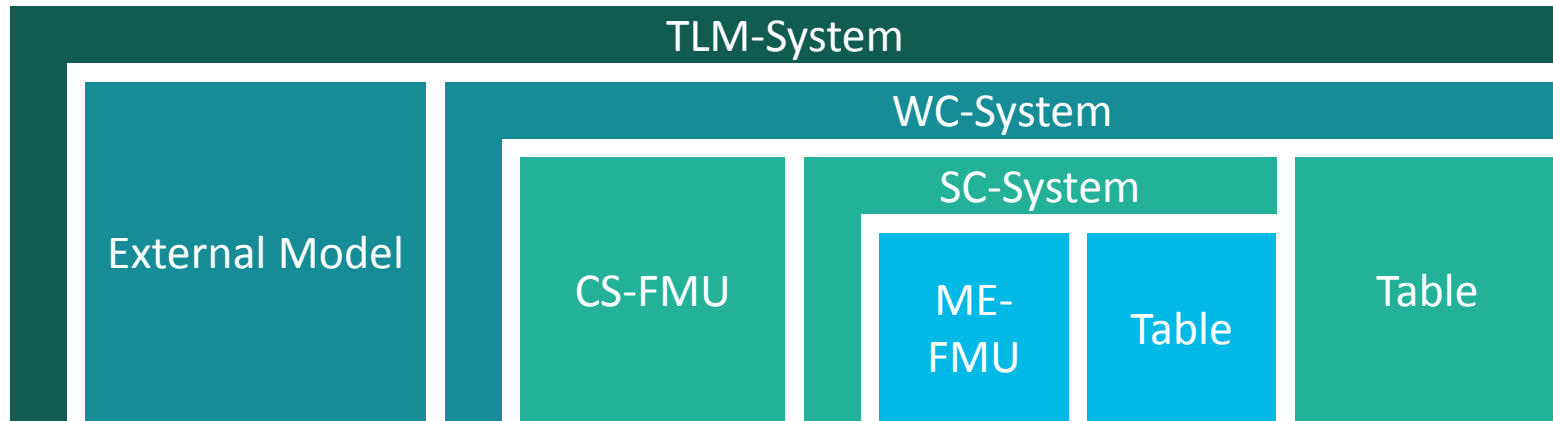
- Weakly connected system
  - Communication at communication time points
  - Extrapolation of inputs

# Transmission-line Modelling

- Technique which considers physical property of signals



# Composite Model Structure (III)



- Transmission Line Modelling
  - Physical signal connections

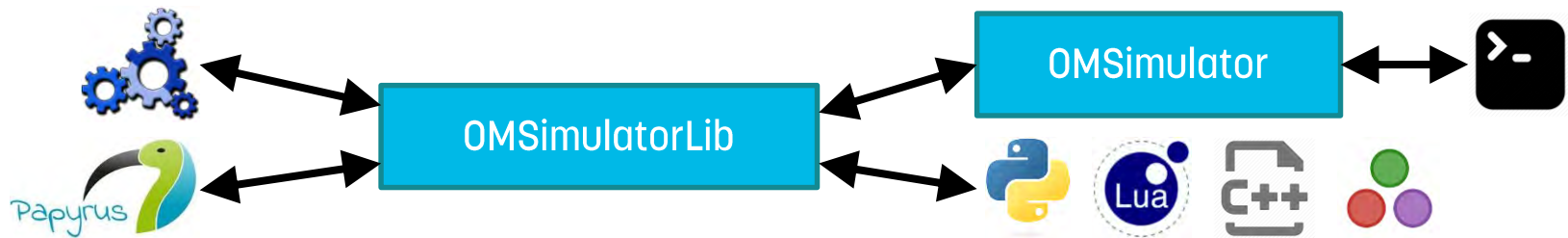
**MODPROD Session 2b:**  
"Numerically Robust FMI  
Co-simulation using TLM  
with OMSimulator"



# System Structure and Parameterization

- Status: 1.0-RC1 (January 21, 2019)
- Tool independent standard for
  - FMU-based model composition
  - FMU parameterization
- Missing...
  - Simulation information (solvers etc.)
  - Buses, Tables

# User Interface



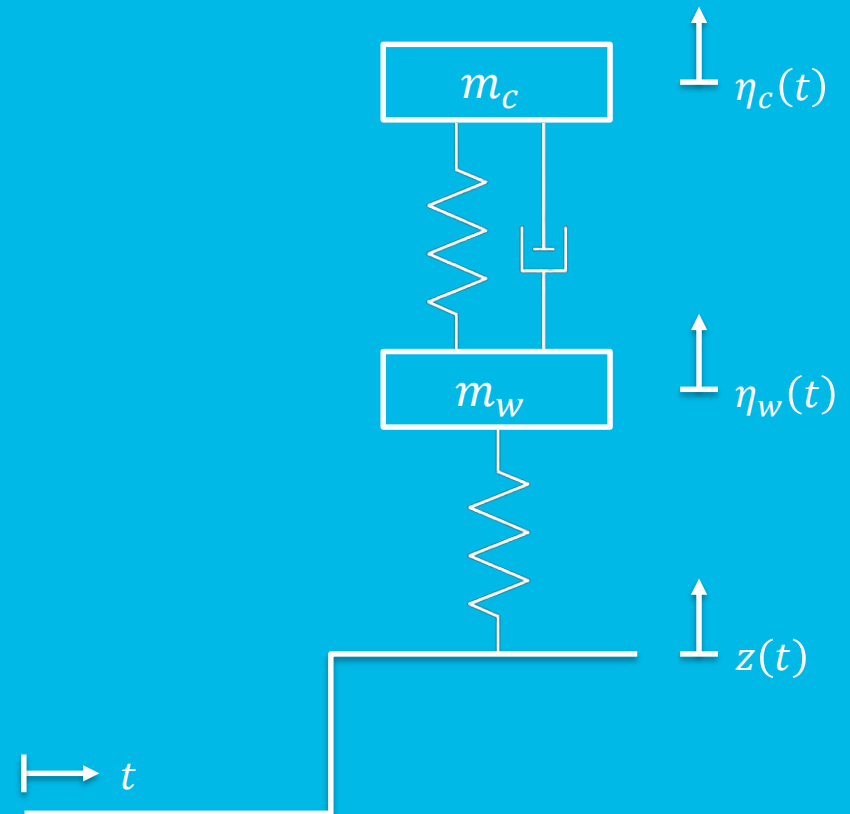
- Command-line interface
- Scripting interface
- Graphical interface

# Summary

- OMSimulator v2.0 is now available
  - OpenModelica v1.13.2
- Strongly-Coupled System
  - Direct communication schema
- Weakly-Coupled Systems
  - Input extrapolation
- TLM-Systems

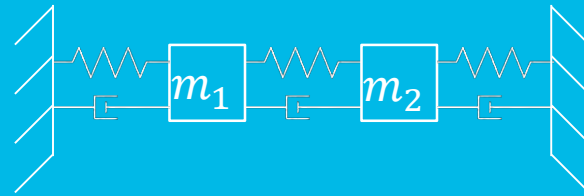
# Demo

## Quarter Car Model



# Exercise

## Dual Mass Oscillator

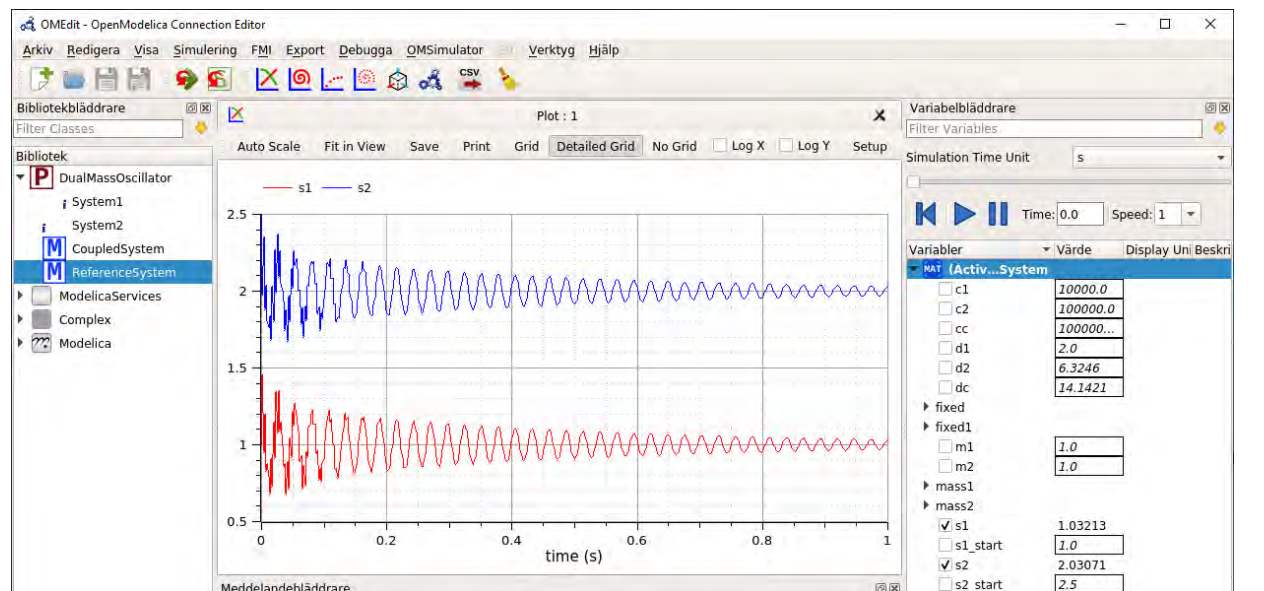


# Dual Mass Oscillator

- Splitting the mechanical (reference) model into two subsystems using force-displacement coupling
- Defining interfaces for the FMUs
- Creating a FMU-based composite model (CS/ME) in OMEdit
- Set start values
- Simulate the composite model

# Dual Mass Oscillator (I)

- Open DualMassOscillator.mo in OMEdit
- Simulate DualMassOscillator.ReferenceSystem
- Play with s1\_start and s2\_start to get the system into motion

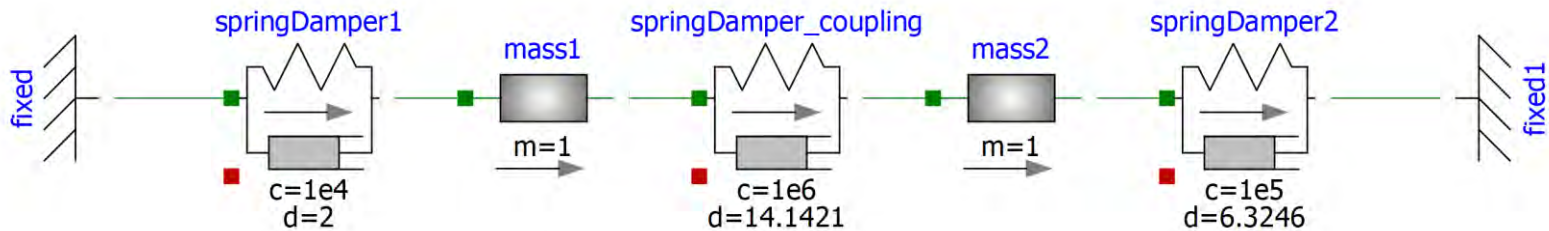


# Dual Mass Oscillator (II)

- Break the model `DualMassOscillator.ReferenceSystem` down into two FMUs
  - Note: Duplicate this model and delete the not needed components
- Define interfaces (inputs/outputs) by adding signal ports from `Blocks.Interfaces` and sensors e.g. from `Electrical.Analog.Sensors`
- Export the two models as FMUs

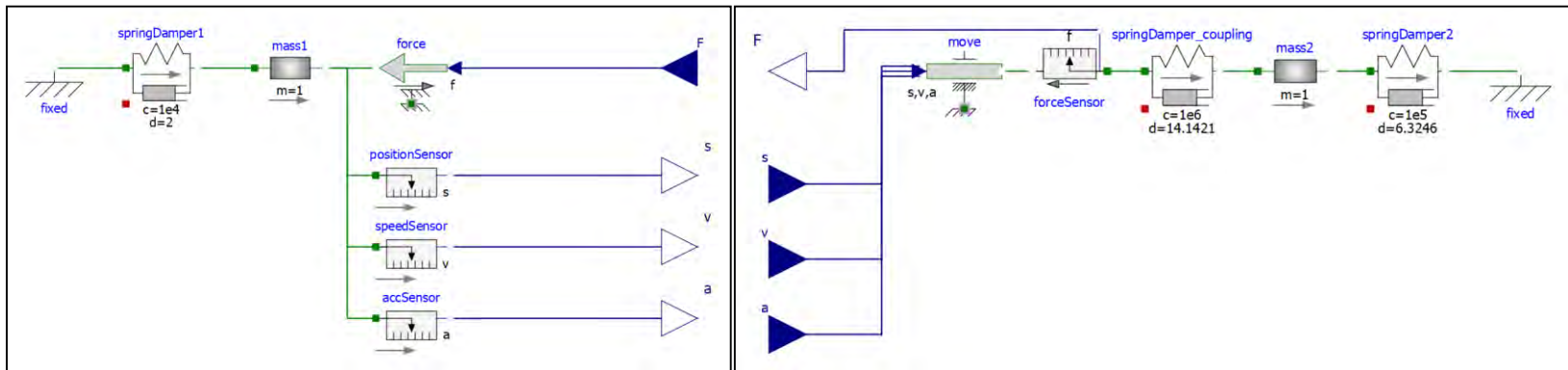


# Dual Mass Oscillator (II)



FMU1

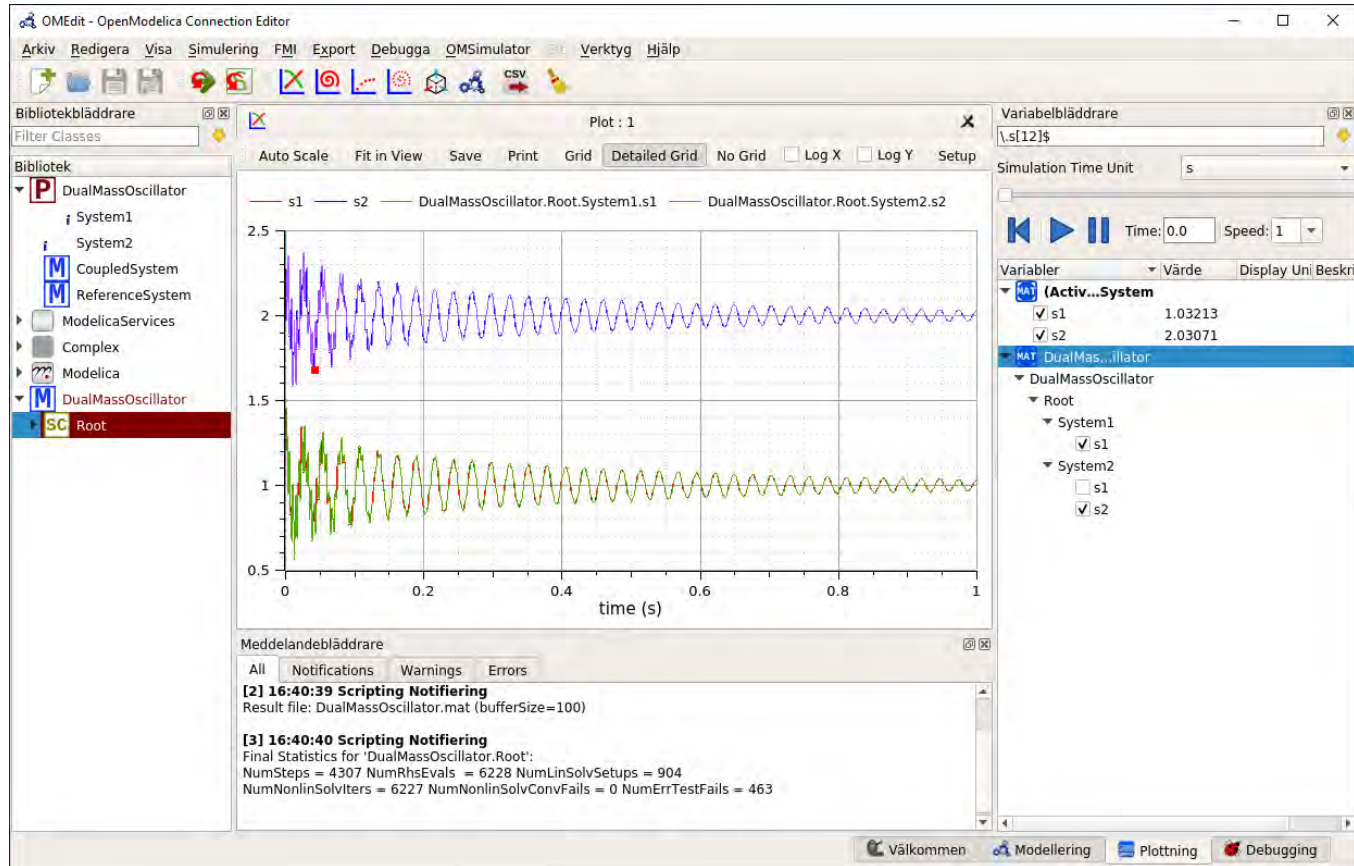
FMU2



# Dual Mass Oscillator (III)

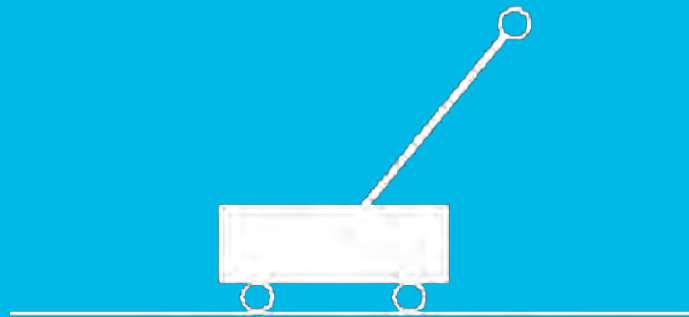
- Create a new OMSimulator model in OMEdit
- Import the FMUs just created
- Instantiate, set start values, and simulate the model
- Do the same for both, ME-FMUs and CS-FMUs

# Dual Mass Oscillator (III)



# Exercise

Inverted Pendulum

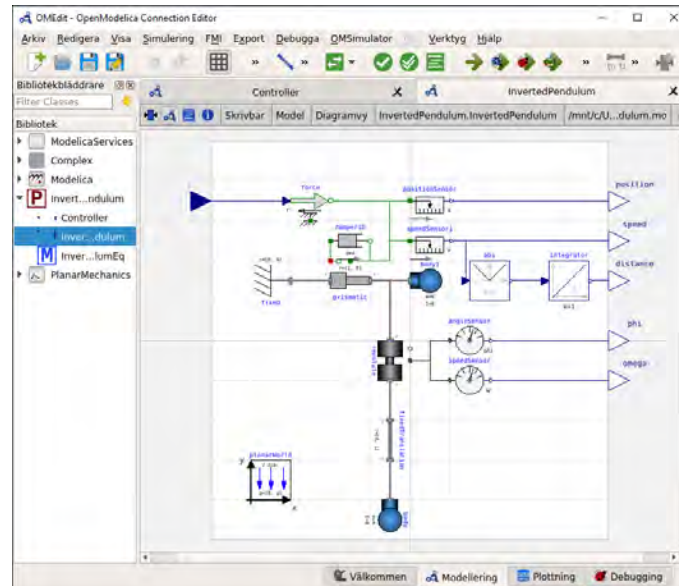
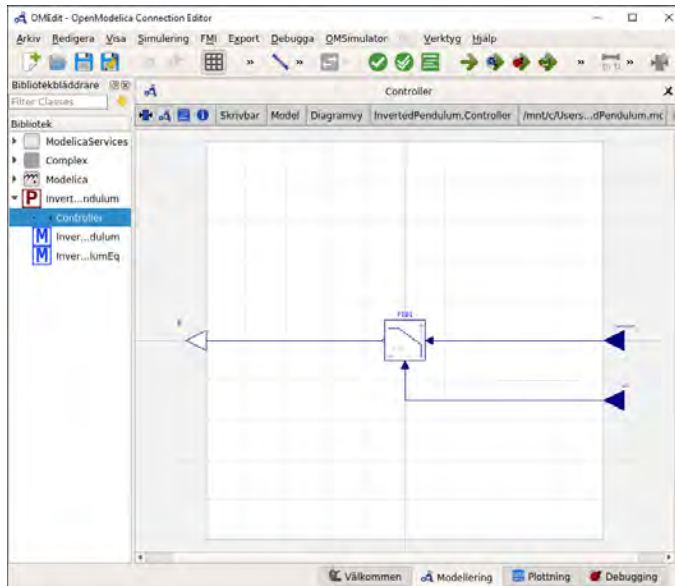


# Inverted Pendulum

- Create a physical model of an inverted pendulum on a cart and export it as FMU
- Create a controller model and export it as FMU
- Create a Python script to connect and simulate both FMUs
- Use the Python interface to optimize the parameters of the controller

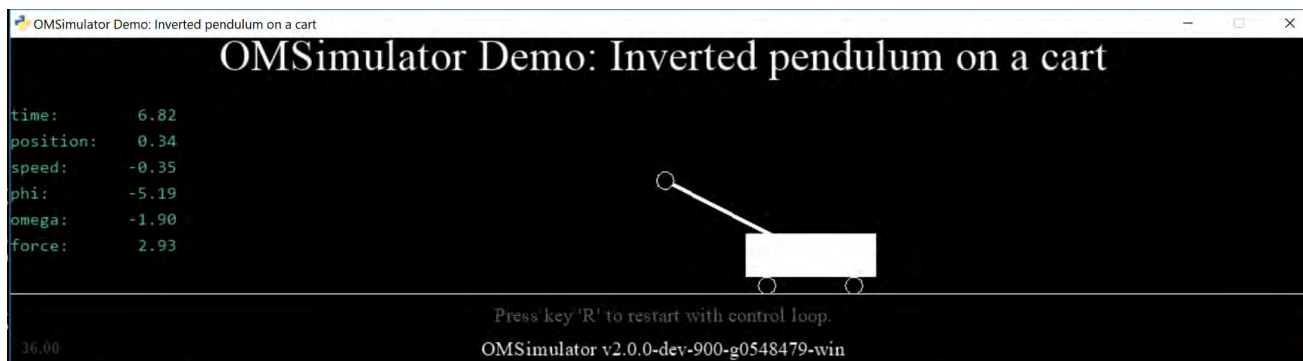
# Inverted Pendulum (I)

- Open InvertedPendulum.mo in OMEdit
- Export the models as FMUs



# Inverted Pendulum (II)

- Copy the FMUs to the Exercise folder:  
InvertedPendulum.fmu and Controller.fmu
- Open Start->cmd.exe and run the OMSimulator python script
  - > set PATH=c:\python64\;c:\%OPENMODELICAHOME%\bin;%PATH%
  - > OMSimulatorPython DemoInvertedPendulum.py

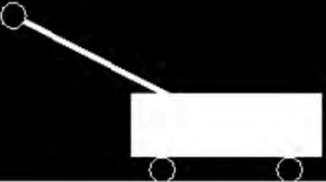


# Inverted Pendulum (II)

OMSimulator Demo: Inverted pendulum on a cart

OMSimulator Demo: Inverted pendulum on a cart

time:	6.82
position:	0.34
speed:	-0.35
phi:	-5.19
omega:	-1.90
force:	2.93



Press key 'R' to restart with control loop.

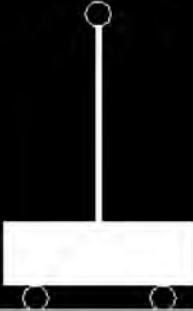
36.00 OMSimulator v2.0.0-dev-900-g0548479-win

Now press R key to restart with controller

OMSimulator Demo: Inverted pendulum on a cart

OMSimulator Demo: Inverted pendulum on a cart

time:	3.76
position:	-0.01
speed:	0.11
phi:	0.00
omega:	-0.01
force:	2.17



Press key 'R' to restart with control loop.

40.00 OMSimulator v2.0.0-dev-900-g0548479-win



# Inverted Pendulum (III)

- The Python script (`OptimizeInvertedPendulum.py`) is used to find the best set of parameters for the PID controller. The optimization minimizes the total distance that the cart is moving within 10s. The idea is that the cart should not move if the pendulum is stable.
- The current PID controller is only looking at the angle of the pendulum and therefore still moving with the optimal parameters according to the optimization.

# Inverted Pendulum (III)

- Optimize the Controller so that the cart moves as little as possible. Open `OptimizeInvertedPendulum.py` and play with the parameters, then run
  - > `set PATH=c:\python64\;c:\OMSimulator\bin;%PATH%`
  - > `OMSimulatorPython OptimizeInvertedPendulum.py`
- Implement a better controller in Modelica for the inverted pendulum and optimize it again.
- One idea: Combine two PID controllers (one for  $\phi$  and one for speed) by adding their outputs together. Therewith get a controller that manage to keep the pendulum stable without the cart moving sideways.

[www.liu.se](http://www.liu.se)

# Transmission Line Method (TLM) for Numerically Stable Co-simulation

Robert Braun  
Linköping University

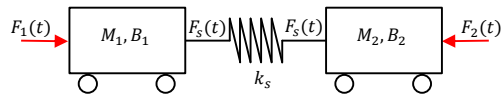


## Introduction

- TLM
  - Wave propagation in physical systems
  - Physically motivated time delays
- Examples of use:
  - **Atlas Copco**: Simulation of wave propagation in rock drills
  - **SKF**: Co-simulation of bearing models
  - **Hopsan**: Wave propagation in fluid power systems
  - **Hopsan/Modelica**: Parallel simulation

## Background

Simulation models consist of equation systems, solved at every time step. Example:



$$\begin{cases} \ddot{x}_1(t)M_1 + \dot{x}_1(t)B_1 = F_1(t) - F_s(t) \\ F_s(t) = k_s(x_2(t) - x_1(t)) \\ \ddot{x}_2(t)M_2 + \dot{x}_2(t)B_2 = F_s(t) - F_2(t) \end{cases}$$

Important to solve all variables at same time ( $t$ )!

## Background

- Often desired to split up equation system
  - Parallell simulation on multi-core processors
  - Co-simulation between different simulation tools
  - Linear relationship between simulation time and model size

## Background

Decoupling will delay certain variables:

$$\text{Subsystem 1: } \begin{cases} \ddot{x}_1(t)M_1 + \dot{x}_1(t)B_1 = F_1(t) - F(t) \\ F(t) = k(x_2(t-T) - x_1(t)) \end{cases}$$

$$\text{Subsystem 2: } \ddot{x}_2(t)M_2 + \dot{x}_2(t)B_2 = F(t-T) - F_1(t)$$

Hence, values from previous iteration must be used.

⇒ Reduced accuracy

⇒ Risk for instability



## Background

~50 years ago, two research groups came up with the same idea:

*"In reality, information propagation speed is always limited by speed of sound (or light)"*

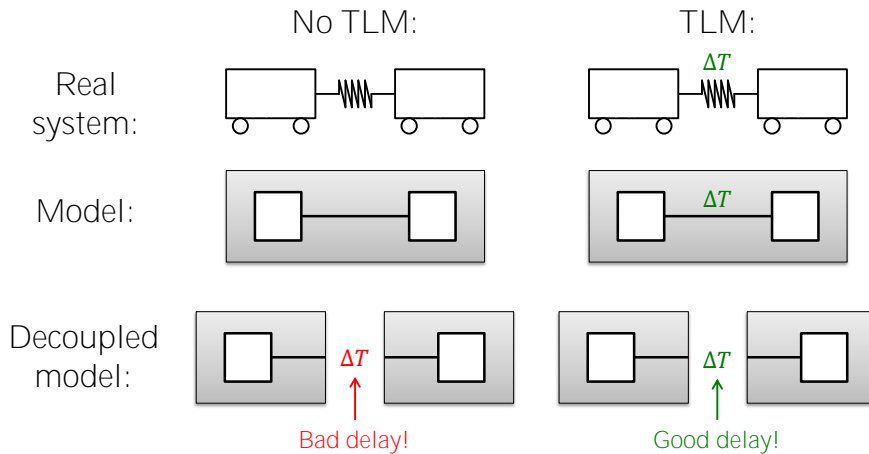
- (Auslander, 1968) (Johns & O'Brian, 1971)

- Examples:
  - Hydraulic pipes
  - Mechanical springs,
  - Electrical transmission lines



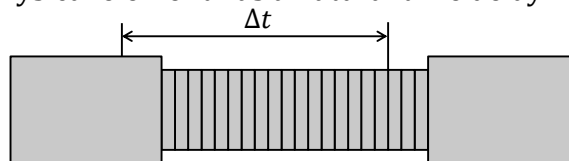
# Transmission Line Modelling - TLM

"Physically Motivated Decoupling"



## Transmission Line Modelling (TLM)

Every physical element has a natural time delay:



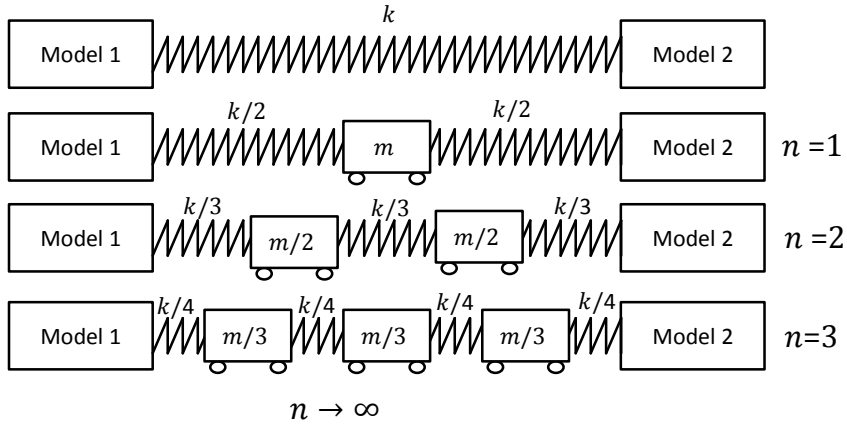
Physically motivated decoupling

→ numerical stability

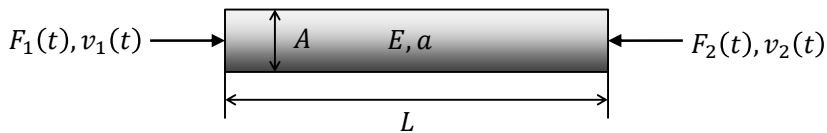
Asynchronous communication

→ independent time steps

## Transmission Line Modelling (TLM)



## Transmission Line Modelling (TLM)



TLM equations:

$$F_1(t) = F(t - \Delta t) + Z_c v_1(t) + Z_c v_2(t - \Delta t)$$

$$F_2(t) = F(t - \Delta t) + Z_c v_2(t) + Z_c v_1(t - \Delta t)$$

Capacitance (stiffness):

$$C = Z_c / \Delta t$$

Inductance (inertia):

$$L = Z_c \Delta t$$



## Two-mass example again

Spring equation is replaced by the TLM equations:

$$\begin{cases} \ddot{x}_1(t)M_1 + \dot{x}_1(t)B_1 = F_1(t) - F_{s2}(t) \\ F_{s1}(t) = Z_C \dot{x}_1(t) + F_{s2}(t - T) + Z_C \dot{x}_2(t - T) \\ F_{s2}(t) = Z_C \dot{x}_2(t) + F_{s1}(t - T) + Z_C \dot{x}_1(t - T) \\ \ddot{x}_2(t)M_2 + \dot{x}_2(t)B_2 = F_{s2}(t) - F_1(t) \end{cases}$$



## Two-mass example again

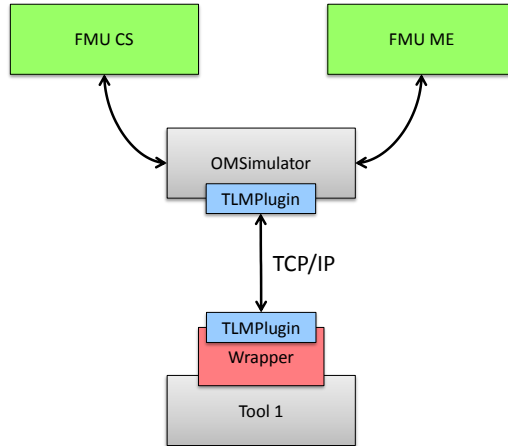
Splitting up the system now yields:

$$\begin{aligned} \text{System 1: } & \begin{cases} \ddot{x}_1(t)M_1 + \dot{x}_1(t)B_1 = F_1(t) - F(t) \\ F_{s1}(t) = Z_C \dot{x}_1(t) + F_{s2}(t - T) + Z_C \dot{x}_2(t - T) \end{cases} \\ \text{System 2: } & \begin{cases} F_{s2}(t) = Z_C \dot{x}_2(t) + F_{s1}(t - T) + Z_C \dot{x}_1(t - T) \\ \ddot{x}_2(t)M_2 + \dot{x}_2(t)B_2 = F_{s2}(t) - F_2(t) \end{cases} \end{aligned}$$

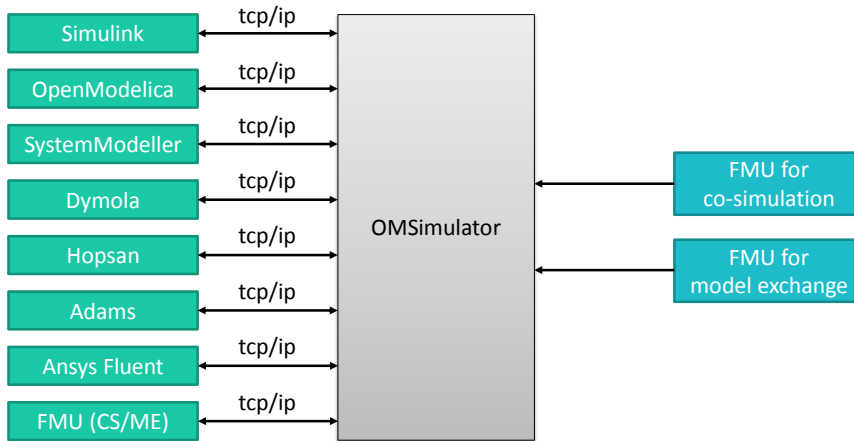
No *numerical* time delays!



## Co-simulation framework

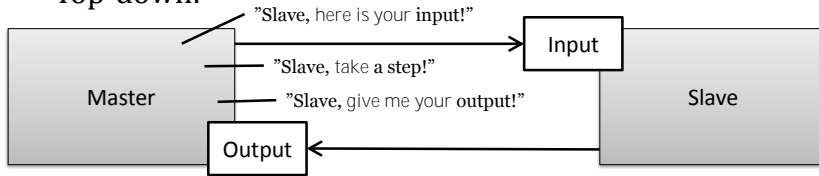


## OMSimulator Overview

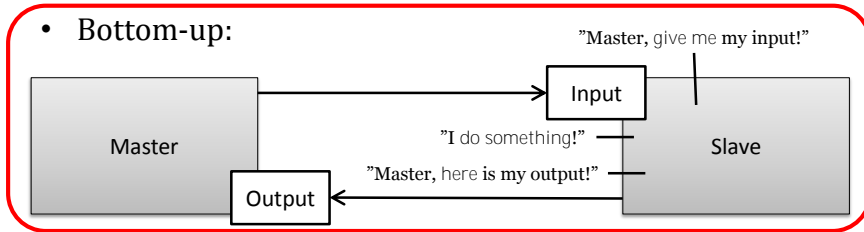


## Execution models

- Top-down:

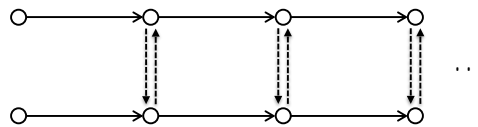


- Bottom-up:



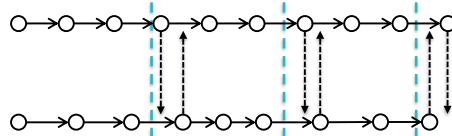
## Data exchange

- Synchronous communication



- ∴ Simple to implement
- ∴ Requires no interpolation

- Asynchronous communication



- ∴ Variable step-size can be used!

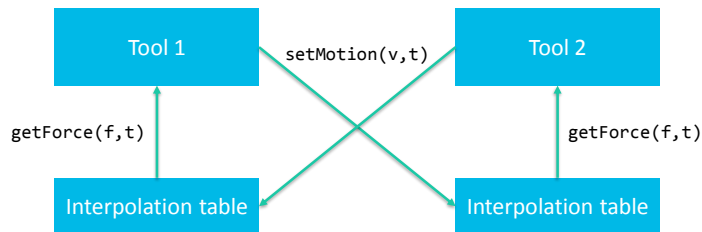
Communication points



## Data exchange

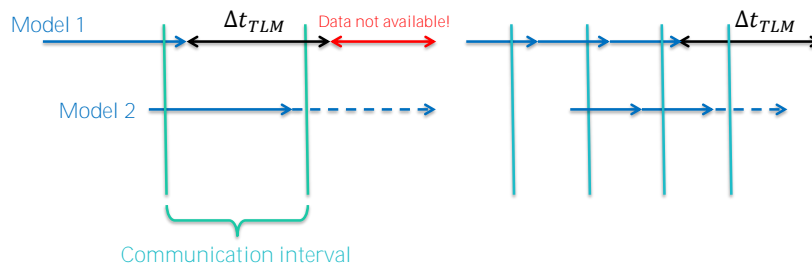
- `setMotion()` – only at communication points
- `getForce()` – any time during step

∴ Implicit and multi-step solvers can be used!



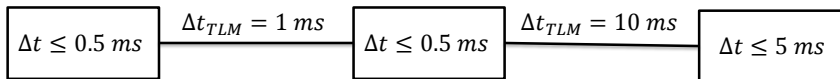
## Data exchange

- Requirement:  $\Delta t_{model} \leq 0.5\Delta t_{TLM}$



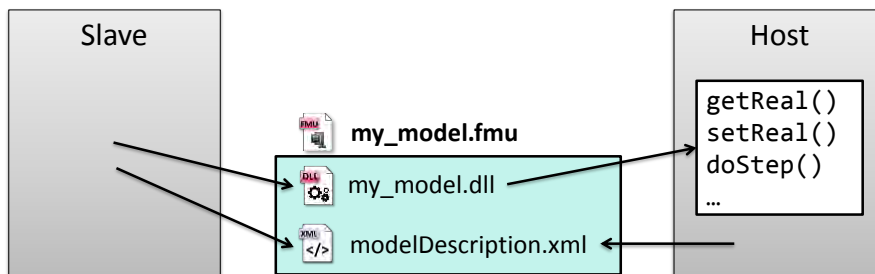
## Data exchange

- Requirement:  $\Delta t_{model} \leq 0.5 \Delta t_{TLM}$



## Functional Mockup Interface (FMI)

- Standardized interface for tool coupling



Robert Braun

[www.liu.se](http://www.liu.se)



## Exercises:

- Hydraulic system and "motor"
  - Two FMU for co-simulation (win64)
  - Lua-script

