"D4.1.4 – Design of FMI extensions for FMI TLM-based co-simulation"

"Sub WP 4.1: Theory, semantics and standardization of multi-mode DAE systems"

"Work Package 4: Systems with multiple operating modes"

MODRIO (11004)

Version 1.0

Date 2016-04-27

Authors

Robert Braun	SICSEast
Dag Fritzson	SKF

TLM FMI extensions

1. Abstract

Asynchronous co-simulation using transmission line modelling offers good numerical robustness. It also enables distributed step sizes and parallelism, which can improve simulation performance. A drawback with the current implementation is that it uses tool-specific wrapper code. It is desirable to improve compatibility with other tools. For this reason, the possibilities for combining the Functional Mockup Interface (FMI) with the framework have been investigated. Three possible approaches have been identified: using the existing FMI standard, modifying the existing FMI standard, and creating a new FMI standard for TLM. The main issue is that the intended use of the current FMI standard for co-simulation is fundamentally different from asynchronous co-simulation using TLM.

2. Introduction

The purpose of this report is to investigate the possibilities for supporting the Functional Mockup Interface (FMI) standard in asynchronous co-simulations using transmission line modelling (TLM). Previous experiments using synchronous communications have been successful (Braun & Krus, 2013). With the TLM method, external models in the co-simulation are numerically isolated by using physically motivated time delays. This ensures numerical stability, which is a very important property in simulation (Nakhimovski, 2006). It also enables parallel execution, which can greatly improve simulation performance (Braun, 2015). Prototypes for multi-model simulation using TLM has been successfully implemented and are described in MODRIO reports D4.2.14 and D4.2.15.

2.1. Transmission Line Modelling

Transmission line modelling is based on wave propagation theory, and can be derived from the telegrapher's equations (Auslander, 1968). Every physical element has a natural time delay, which depends on its capacitance and inductance. For mechanical elements, this equals compressibility and inertia, respectively. Forces F_1 and F_2 on the two ends of an element at time t can be computed as follows:

$$F_1(t) = c_1(t) + v_1(t) * Z_c$$
$$F_2(t) = c_2(t) + v_2(t) * Z_c$$

Here v_1 and v_2 are the velocities, Z_c the characteristic impedance of the element, and c_1 and c_2 the wave variables, which contains the delayed information from the other end of the element:

$$c_1(t) = F_2(t - \Delta t_{TLM}) + v_2(t - \Delta t_{TLM}) * Z_c$$
$$c_2(t) = F_1(t - \Delta t_{TLM}) + v_1(t - \Delta t_{TLM}) * Z_c$$

The solvers in the slaves compute velocity from the force using numerical integration. A multi-step solver can use the computed velocity to calculate a new force, used for the next step. Force can be obtained at any time instance by using interpolation of wave variables.

2.2. Co-simulation framework

The co-simulation framework consists of a manager executable, which handles communication between slaves. Both the manager and the slaves contains a TLM plugin object, which communicate with each other through network sockets, see Figure 1.

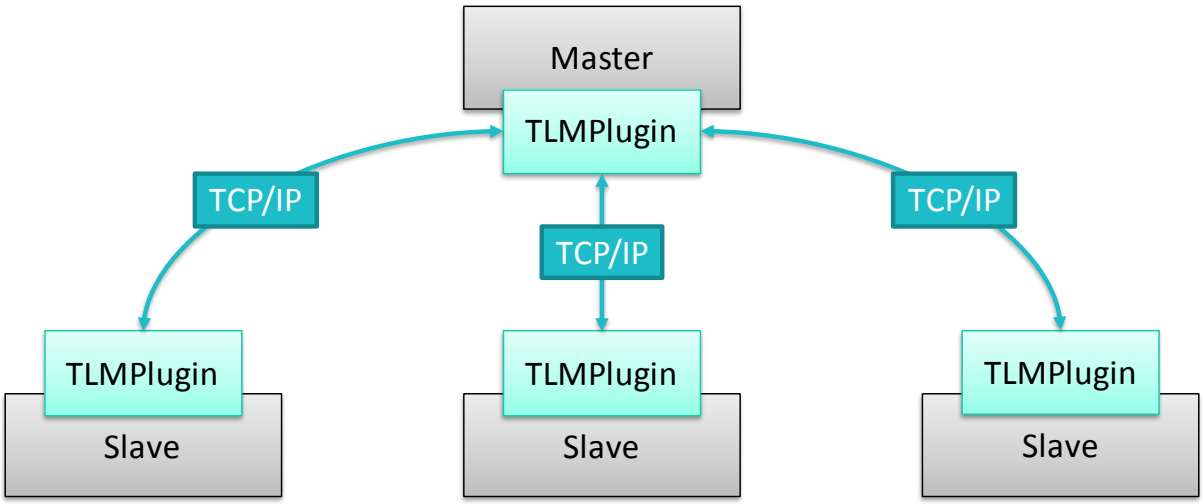

Figure 1: The master and the slaves communicate through network sockets.

The slaves notify the master of their motion (velocity, angular velocity, position and angle) at given time intervals. The master forwards this information to the connected slaves. Each slave has its own interpolation tables, which is populated with received variables. Whenever a force evaluation is requested, the wave variables are interpolated for the specified time instance. Slaves can read and write variables through the setMotion() and getForce() functions, see Figure 2. Arrows are directed from the caller to the callee. A demonstrator that uses the framework to simulate wind power model with a bearing will be described further in MODRIO document D8.7.4.

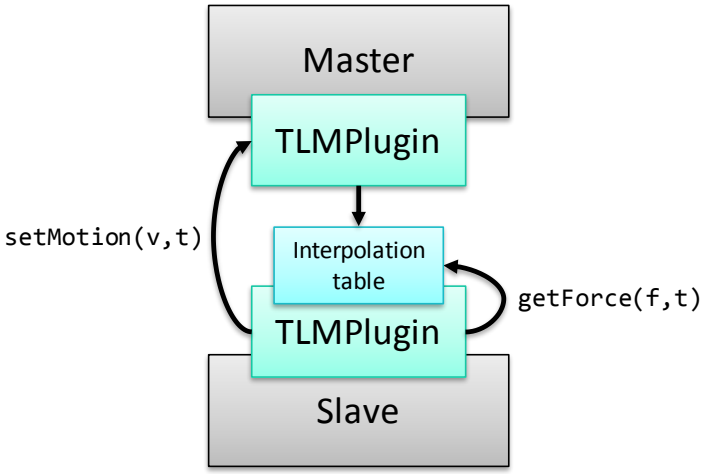

Figure 2: Slaves can read and write variables using the setMotion() and getForce() functions.

2.3. Functional Mock-up Interface

There are two versions of FMI, co-simulation and model exchange. Furthermore, it is also possible to either import FMUs from external models to the master simulation tool, or export generic FMUs that can be imported by the external models. This leaves four possible options:

- Importing FMUs for co-simulation
- Importing FMUs for model exchange
- Exporting FMUs for co-simulation
- Exporting FMUs for model exchange

2.4. General Difficulties

The main difference between the TLM co-simulation framework and the intended co-simulation methods for the FMI standard is the asynchronous communication pattern. With FMI, the master tool controls the simulation by calling functions in the slaves. Asynchronous communication, however, requires each slave to be in charge of communications and solver methods. Slaves must be able to set and get variables at arbitrary time instances in order enable distributed step sizes, multi-step solvers and implicit integration methods.

The different approaches result from the fact that TLM introduces physically motivated time delays. As a result, variables are sent from submodel A at time T is valid for submodel B at time $T + dT$. Variables are thus "sent into the future" due to the information propagation delay. Since a future variable is already known to the solver, it is possible to interpolate values for each integration step (major or minor). Since state-of-the-art co-simulation algorithms does not use physically motivated delays, they must instead rely on extrapolation of input variables.

Base Requirements	
1	Start-up methods for slave tools must be available.
2	It must be possible to control start time, stop time and maximum step size for external models.
3	It must be possible to specify network ports for communication with external models.
4	Each sub-model shall have an independent step-size, limited by half the communication interval.
5	Multi-step and implicit solvers in external models should be supported.
6	External models must be able to conduct force evaluation in any time step.
7	In between communication steps the forces must be interpolated.
8	Sub-models must return the motion to the master tool at certain time intervals.
9	One-step mode execution of should be supported for good resolution in interpolation tables.

3. Importing FMUs from Slaves to Master

Importing FMUs to the co-simulation framework requires a stand-alone executable wrapper. The wrapper imports the FMUs and controls the slaves through the FMI API, while communicating with the TLM manager through network sockets.

3.1. FMI for Co-simulation

FMI for co-simulation is supported by many tools and should offer good compatibility. Figure 1 shows how import of FMI for co-simulation can be implemented in the framework. A drawback is that it is difficult to evaluate the forces in between communication points. This is required for multi-step or implicit solvers. Using linear extrapolation will introduce numerical errors and reduce stability. FMI is based on the master calling functions in the slave, but the master cannot know when the slave requires a force evaluation.

One solution can be to letting each FMU handle interpolation of forces internally. However, this expects the FMU to be aware of the master algorithm, which is not desirable. It also requires a function for setting the time variable in the slaves, which currently does not exist for co-simulation.

Additional Requirements (alternative 1)	
1	FMUs must support interpolation internally
2	FMI for co-simulation must support setting time variables

An alternative solution is to provide the slaves with a callback function which can be used by the solver to obtain input variables at desired time instances. This requires a modification to the FMI standard.

Additional Requirements (alternative 2)

1	FMU must provide a callback function for slave to obtain force variable at a specified time.

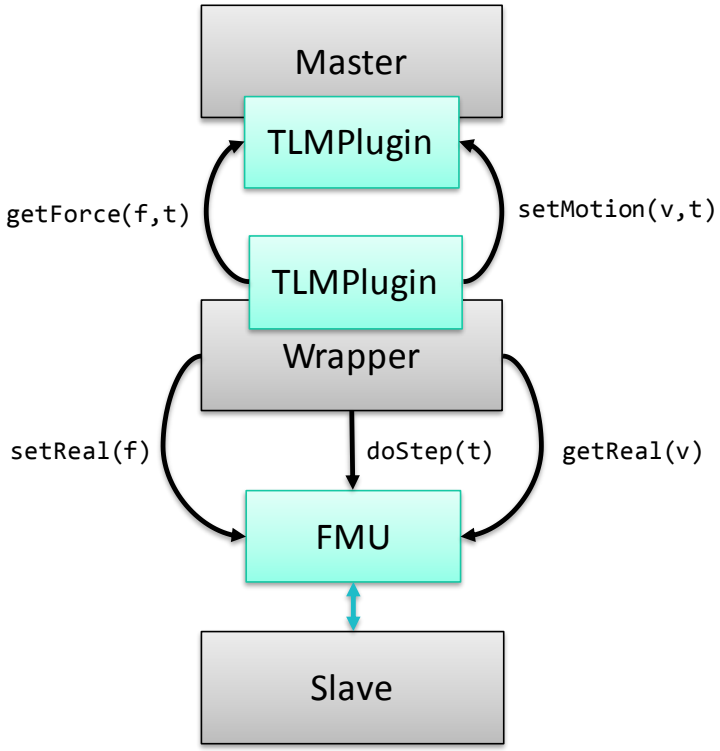

Figure 1: Importing FMUs for co-simulation to the framework requires a wrapper executable.

3.1. FMI for Model Exchange

Importing FMUs for model exchange requires a numerical solver in the wrapper executable, as shown in figure 2. In this way, interpolation of input variables for multi-step integration can be introduced directly in the wrapper. A drawback is that many slave tools require advanced or customized solvers and may thus not be compatible with this solution.

Newton iteration is possible due to the getDirectionalDerivatives() function in the model exchange API. Calling this function makes it possible to compute the Jacobian matrix. This of course requires the FMU to support this feature. Such additional requirements reduce compatibility.

Additional Requirements	
1	FMU must provide directional derivatives.
2	Numerical solvers must be implemented in the generic wrapper executable.

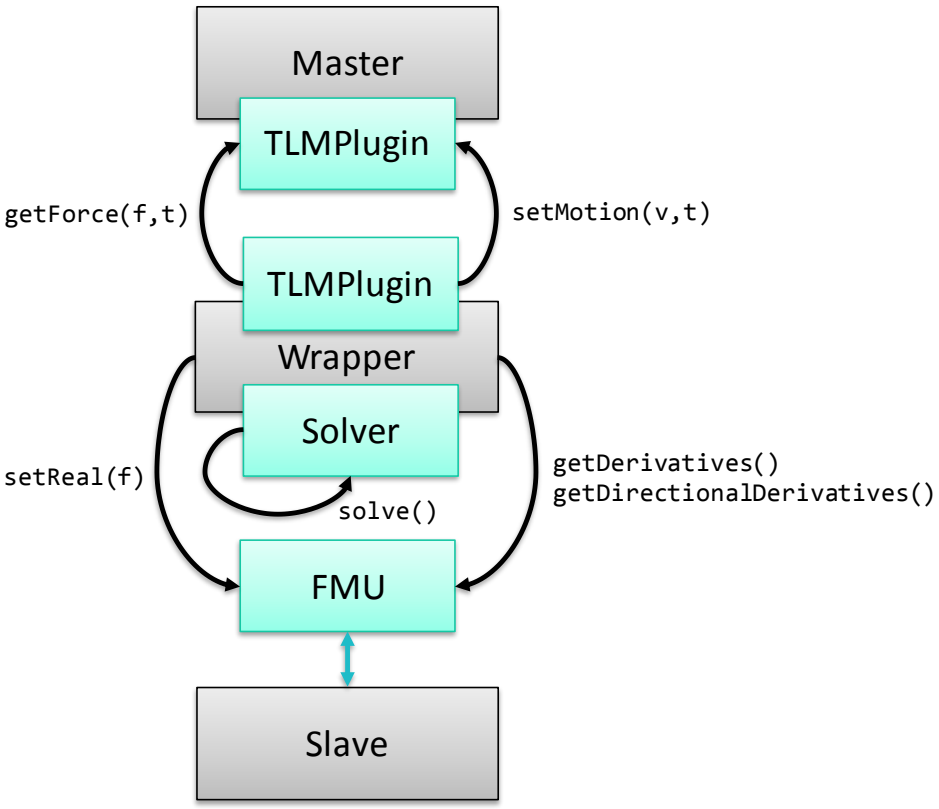

Figure 2: Importing FMUs for model exchange requires a wrapper executable with a numerical solver.

4. Exporting Generic FMUs from Master to Slaves

A different approach is to let the participating external tools import generic FMUs, which handles the socket communication with the framework. With this method, no wrapper executable is required. A drawback is that it contradicts the intended working principle with the FMI standard.

4.1. FMI for Co-simulation

One option is to export an FMU for co-simulation, as shown in Figure 4. The solver in the slave tool can then use setReal(), setTime() and getReal() functions to set and get variables for desired time instances. A problem is that no setTime() function for FMI for co-simulation currently exists. A possible solution could be to use doStep() functions in combination with getFMUstate() and setFMUstate() to step forward and backward in time, see Listing 1. This is possible since the FMU will not contain any equations and only handle interpolation of variables. The main drawback is that the slave tool must be designed for this, which requires the solver to call several FMI-specific functions. This would be facilitated by adding a setTime() function to the FMI for co-simulation standard.

Additional Requirements	
1	FMI for co-simulation must support setting time variables
2	Solver in slave must call FMI functions for taking minor integration steps

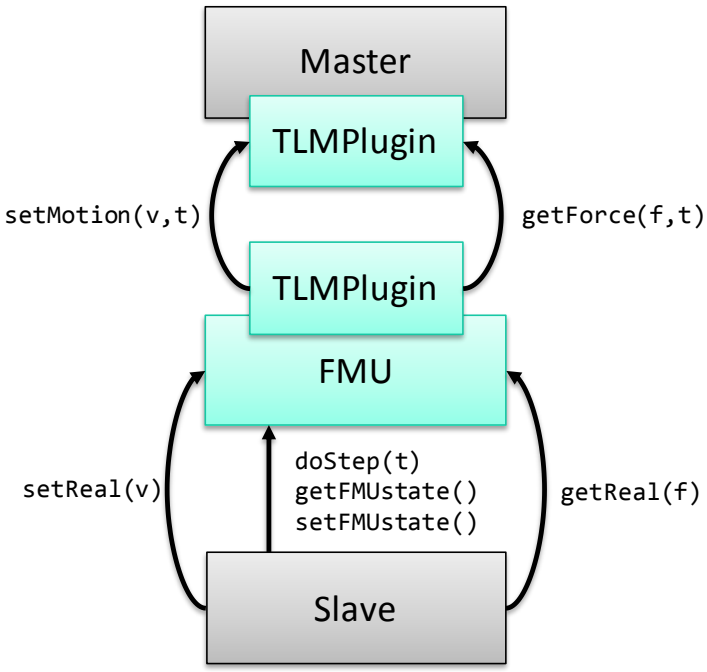

Figure 3 : Exporting a generic FMU for co-simulation to the slave makes it possible for the slave to control the simulation

```
void *orgState;
fmi2GetFMUstate(fmu, orgState);        //Store state for time t
fmi2DoStep(fmu, T_end, dt1, fmi2True);
fmi2GetReal(s1, ..., 1, &force);       //Force at t+dt1
fmi2SetFMUstate(fmu, orgState);        //Reset time to t
fmi2DoStep(fmu, T_end, dt2, fmi2True);
fmi2GetReal(s1, ..., 1, &force);       //Force at t+dt2
```

Listing 1: The slave can obtain variables for a specific time by saving and restoring the FMU state.

4.1. FMI for Model Exchange

Finally, it is also possible to let each slave tool import generic FMUs for model exchange, see figure 4. This gives each slave complete control of the equation solving. Requesting input variables for certain time instances is possible by iteration, which is handled inside the FMU. A drawback is that slaves must provide a numerical solver, which is not always possible. It also requires the slave tool to support import of FMUs for model exchange, which reduces compatibility.

Förklara noggrannare, liknar tlmpllugin, varför problem, förväntyar sig bara data ,ellan major steps

Additional Requirements	
1	Slave must have a numerical solver

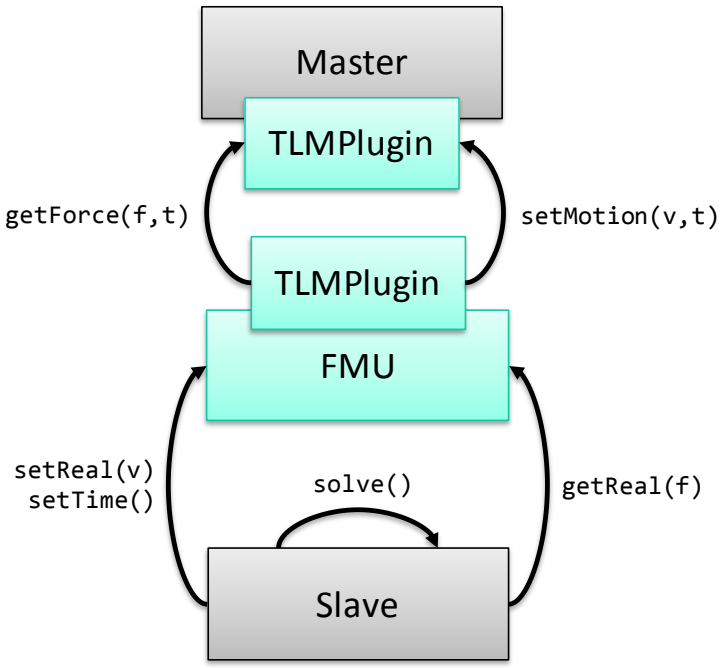

Figure 3: Exporting generic FMUs for model exchange makes it possible to use the numerical solver in the slaves.

5. Existing FMI Standard

With the current FMI standard for co-simulation, the time variable in the FMU is only updated implicitly by the doStep() command. As a consequence, it is only possible to set input variables at the beginning of each step. Hence, interpolation using time delays is not possible. This removes the main benefit of TLM: numerical stability. If linear extrapolation is sufficient, however, import of FMI for co-simulation will at least increase compatibility with other tools.

FMI for model exchange does not have the same limitations. On the other hand, it either requires the solver to be implemented in the master framework, or that the framework exports FMUs to the slaves. While the latter is possible, it also requires the solver to understand how to use getReal() in conjunction with setTime() in order to obtain the intensity variable at a specific time.

6. Suggested Extensions to the FMI Standard

The FMI standard can be modified in two ways. First, it may be possible to extend the current standard with minor extensions to enable asynchronous TLM co-simulation. Second, another possibility is to create a new FMI standard for TLM, alongside the existing standards for co-simulation and model exchange.

6.1. Modifying the existing standard

The framework is intended for importing and connecting external models co-simulation with distributed solvers. With the current standard, the most suitable solutions are therefore either export of FMI for model exchange or import of FMI for co-simulation. The model exchange method would require a getReal() function with a specified time instance, to avoid having to use a combination of setTime() and getReal(), see Listing 2.

```
fmi2Status fmiGetReal (fmi2Component c,
                       const fmi2ValueReference vr[],
                       size_t nvr,
                       const fmi2Real value[],
                       const fmi2Real time);
```

Listing 2: Exporting model exchange to the slave could be facilitated by adding a getReal() function with a time argument.

Importing FMI for co-simulation requires support for interpolation of input variables, in order to ensure numerical stability. As suggested above, this can be implemented in two ways; either by letting the FMU handle interpolation internally, or by interpolation tables in the wrapper executable. Forcing the FMUs to support interpolation is an additional requirement which will likely reduce compatibility. Thus, it is advisable to let the wrapper handle interpolation and extend the FMI standard with a callback function, which returns the force for a specified time instance. Using this feature in the FMU will be optional and dependent on the tool capabilities.

6.1.1. Suggestion 1: Interpolation in the wrapper

If the wrapper handles the interpolation, the slave solver must be able request an interpolated input value at any specified time instance. This requires a callback function. It could be implemented by assigning the function for a specified variable through a function in the API, see Listing 3. In case interpolation is not possible, the Boolean parameter in the callback function should be set to false. The slave must then keep requesting the variable until it is available.

```
fmi2Status fmiSetRealCallbackPtr(fmi2Component c,
                                 const fmi2ValueReference vr,
                                 const fmi2Real (*fnc)(fmi2Real, *bool));
```

Listing 3: Interpolation in the wrapper requires a callback function for the slave requesting interpolated variables

It is up to the solver in the slave how and if this pointer is used. It should not require any modification to the XML description. For the sake of compatibility, an FMU should always work to simulate without receiving callback pointers from the master. Should the master send a callback pointer to an FMU that does not support it, the FMU should also still be able to run. By default, the function should do nothing and maybe give a warning message. Input variables should still be updated before every major integration step.

6.1.1. Suggestion 2: Interpolation in the FMU

If callback functions are too complicated, another option can be to add a time argument to the setReal/Integer/Boolean/String functions, as shown in Listing 4. In this way it would be possible for an FMU to handle interpolation of input variables internally.

```
fmi2Status fmi2SetReal (fmi2Component c,
                        const fmi2ValueReference vr[],
                        size_t nvr,
                        const fmi2Real value[]
                        const fmi2Real time);
```

Listing 4: Interpolation inside the FMUs requires the possibility of setting variables at a specified time instance.

This can either replace the existing set functions, or be added as new functions in parallel with the existing ones. It should still be required to update input variables at the beginning of each major integration step. No changes to the XML model description should be necessary. Should the FMU not support internal interpolation, it should still be possible to simulate it as usual using extrapolation. The

function should always exist but by default do nothing (and maybe give a warning).

An alternative way of setting variables at a specified time is by letting the master algorithm control the time in an FMU for co-simulation, see Listing 5. This requires a function exactly similar to the existing one in FMI for model exchange:

```
fmi2Status fmi2SetTime(fmi2 Component c, fmi2Real time);
```

Listing 5: If the master algorithm can control the time in an FMU for co-simulation, it can also assign variables for specific time instances

This is an alternative solution to suggestion 2. By controlling the time externally it is possible to use the regular setReal() function for assigning variables at a specific time instance. It is, however, important that the time is reset again before calling doStep(). This feature would require the FMU to inform the master that it supports it through an XML attribute:

allowsExternalTimeControl	
type	xs:boolean
default	false

Figure 4: Externally controlling the time variables requires an XML attribute that informs of this capability.

7. Creating a new FMI standard for TLM

Asynchronous co-simulation using TLM is fundamentally different from the intended use of FMI for co-simulation. Thus, a possible solution could be to create a separate standard for this application. The most important difference is that asynchronous co-simulations are controlled directly by the slaves. The master algorithm only handles communication protocols and data logging. As a consequence, there is no `doStep()` function called by the master. Each slave controls their own step sizes, and takes a step whenever the required variables are available. Maximum step size is limited by half the TLM time delay, which should be provided from the master.

The typical use case is that models are exported from a simulation tool to an FMU which shall participate in a co-simulation. Hence, it is preferable that FMUs are imported to the framework through a wrapper executable. This required callback functions for data exchange during simulation. Only physical variables are allowed, either of type intensity or flow. This depends on the physical domain. Intensities can for example be force, voltage or pressure, while flows can be velocity, current and volume flow. Each TLM interface has one intensity and one flow variable. A possibility could be to replace "value reference" with "interface reference". A drawback with this is that it is not harmonized with the current FMI standards.

Each slave must at any time be able to either write back flow variables to the master or request a calculation of intensity variables. The exchange of wave variables between slaves are handled by the master framework, and must not be visible to the slaves. Wave variables are stored in interpolation tables, which are used to obtain the wave variables corresponding to the time instances when the slave requests a force evaluation. Should the variables required for interpolation not exist in the interpolation table, the slave must be informed so that it knows that it must wait.

The most important interface functions are to control the simulation parameters, assign intensity callback pointer and assign flow callback pointer, see Listing 6. Other general FMI functions, such as fmi2Instantiate, fmi2FreeInstance, fmi2Terminate and fmi2Reset are also required.

```
fmi2Status fmi2SetupExperiment(fmi2Component c,
                               fmi2Real      startTime,
                               fmi2Real      maxStepSize,
                               fmi2Real      stopTime);

fmi2Status fmiSetIntensityCallbackPtr(fmi2Component c,
                                      const fmi2ValueReference vr,
                                      const fmi2Real (*fnc)(fmi2Real));

fmi2Status fmiSetFlowCallbackPtr(fmi2Component c,
                                 const fmi2ValueReference vr,
                                 const void (*fnc)(fmi2Real,fmi2Real));
```

Listing 6: Minimum set of functions required for a new FMI standard for TLM

8. Conclusions

Asynchronous co-simulation using TLM is fundamentally different from the intended use of FMI for co-simulation. The main difference is that simulation step control is handled individually by each slave and not by the master algorithm. However, it may still be possible to use the FMI standard also for asynchronous communication.

With the current standard, FMI for model exchange and co-simulation can both be used. It is also possible to use either import or export FMUs. FMI for model exchange always require a numerical solver, either in the master tool or in the slave. This is not always feasible, and therefore not desirable. FMI for co-simulation can be used but with the current implementation there is not support for interpolation of variables.

Interpolation can be supported by minor modifications to the current standard. If interpolation is handled by the master tool, callback functions to the slaves are required. Interpolation can also be handled by the slaves, if input variables can be assigned for a specified time instance.

A more general solution is to create a new FMI standard for TLM. It would require functions for assigning callback functions, in order to enable interpolation individual step control by the slaves.

These results are preliminary, and so far no prototyping have been done. This work will continue in the ITEA3 project OPENCPS.

9. References

Auslander, D. M. (1968, June). Distributed System Simulation with Bilateral Delay-Line Models. *Journal of Basic Engineering*, 195-200.

Braun, R. (2015). *Distribued System Simulation Methods: For Model-Based Product Development.* Linköping: Linköping University.

Braun, R., & Krus, P. (2013). Tool-Independent Distributed Simulations Using Transmission Line Elements And The Functional Mock-up Interface. *SIMS 54th Conference.* Bergen, Norway.

Nakhimovski, I. (2006). *Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis.* Linköping: Linköping University.