Tool-Independent Distributed Simulations Using Transmission Line Elements And The Functional Mock-up Interface

Robert Braun and Petter Krus

Linköping University, Division of Fluid and Mechatronic Systems, SE-58183 Linköping , Sweden

Abstract. This paper describes how models from different simulation tools can be connected and simulated on different processors by using the Functional Mockup Interface (FMI) and the transmission line element method (TLM). Interconnectivity between programs makes it possible to model each part of a complex system with the best suited tool, which will shorten the modelling time and increase the accuracy of the results. Because the system will be naturally partitioned, it is possible to identify weak links and replace them with transmission line elements, thereby introducing a controlled time delay. This makes the different parts of the system naturally independent, making it possible to simulate large aggregated system models with good performance on multi-core processors. The proposed method is demonstrated on an example model. A suggestion of an XML extension to the FMI standard for describing TLM ports is also presented.

Keywords: Functional Mockup Interface (FMI), Functional Mockup Unit (FMU), Transmission Line Element Method (TLM), Parallelism, Co-Simulation

1 Introduction

If different parts of a simulation model can be run on different processor cores, execution time can be considerably shortened. By using the transmission line element method (TLM) with independent distributed solvers, it is possible to achieve natural parallelism. This paper investigates the possibilities the combine this with the Functional Mock-up Interface (FMI), a standardised interface for connecting different simulation environments. Using TLM and FMI together makes it possible to run distributed simulations with sub-models from different tools, which can fully exploit the benefits of multi-core processors.

First, the backgrounds of FMI, TLM, and the Hopsan simulation environment are explained. Then the implementations of import and export routines of FMI are presented. Finally, the validity of the method is confirmed by experiments on an example model.

2 Functional Mockup Interface

The Functional Mock-up Interface (FMI) is an open standardised interface for connecting simulation environments in a variety of ways. It is developed by the MODELISAR consortium, initiated by Dassault Systems [1]. There are four areas where FMI can be used:

- FMI for model exchange
- FMI for co-simulation
- FMI for applications
- FMI for PLM

The basic concept is to create a Functional Mockup Unit (FMU) from a model in one tool and then import it and use it in a target environment. An FMU consists of a compressed ZIP file with the FMU file extension. It contains an XML description of the contents and the simulation code as a set of C functions, either as binary files and/or compilable source code, which can be used by the host program. It can also contain documentation and a graphical icon. The use of a plain C interface makes FMUs compiler independent. They do, however, still depend on platform and architecture. With FMI for co-simulation the solver is included in the FMU, as opposed to FMI for model exchange, where the solver must be provided by host program. This paper focuses only on model exchange, mainly because it is supported by a larger number of vendors. In the upcoming FMI 2.0 standard, the difference between co-simulation and model exchange will be reduced [2]. It is, however, not yet released and is not used in this paper.

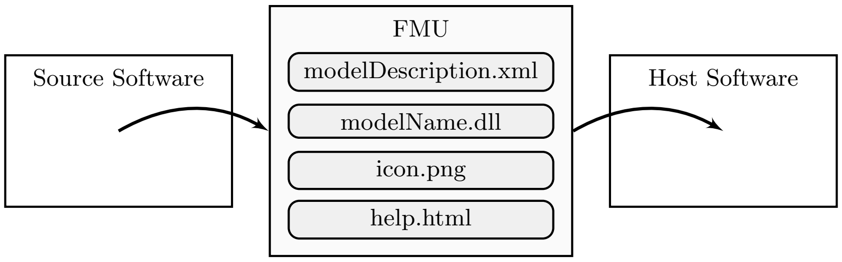

Fig. 1: With the FMI standard, Functional Mockup Units (FMUs) can be used to exchange models among simulation tools.

3 Transmission Line Element Connections

The transmission line element method (TLM) is a method for partitioning models by introducing physically motivated time delays. It is related to the method of characteristics [3] and to transmission line modelling [4]. In physical systems, information propagation is always delayed by capacitances. The concept with transmission line elements is to replace these capacitances in the model with transmission line elements, modelled as characteristic impedances. This method makes it possible to maintain accurate wave propagation, which is not possible by using only pure time delays.

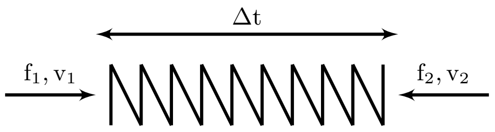

Fig. 2: An example of a transmission line element is the linear mechanical spring.

An example of a TLM element is the linear mechanical spring, as shown in figure 2. As can be seen it is subjected to two forces, f_1 and f_2, from the left and the right side, respectively. According to equation 1 each force is defined by a function of the velocity at the same side and the delayed force and delayed velocity from the other side. This implies that one end is always independent of the opposite end at the same point of time. This enables the use of distributed solvers, where each sub-component in the model solves its own equations. This approach is very suitable for co-simulation and parallel execution [5].

$$\begin{aligned} f_1(t) &= F(v_1(t), f_2(t-\Delta t), v_2(t-\Delta t)) \\ f_2(t) &= G(v_2(t), f_1(t-\Delta t), v_1(t-\Delta t)) \end{aligned} \tag{1}$$

When communicating between different programs using TLM connections, it is important that the variables in each connection are clearly specified. This can be done manually by the user when importing the model to the host environment, although this can be quite cumbersome. An alternative solution would be to include this information in the XML specification in the FMU. A TLM connection is defined as four variables; intensity, flow, wave variable, and characteristic impedance. Some additional variables may also be necessary depending on the physical domain, such as position and equivalent inertia for mechanical connections. Even though similarities between different domains exist, it is unfortunately not possible to use a general definition; the set of variables will always need to be hard-coded depending on the physical type of the connection. For example, an incompressible fluid needs only one flow variable, while a compressible fluid might need variables for both mass flow and volume flow. It is also necessary to provide information of whether the FMU is a resistive component (Q-type) or a transmission line connection (C-type). In order to transfer this information, the following addition to the XML description is suggested, see listing 3

```
<tlmConnections>
  <tlmConnection type="q" domain="hydraulic">
    <q>q1</q>
    <p>p1</p>
    <c>c1</c>
    <Zc>Zc1</Zc>
  </tlmConnection>
  <tlmConnection type="c" domain="mechanic">
    <F>F2</F>
    <x>x2</x>
    <v>v2</v>
    <me>M2</me>
    <c>c2</c>
    <Zc>Zc2</Zc>
  </tlmConnection>
</tlmConnections>
```

Fig. 3: An addition to the FMU XML description, describing TLM ports, is suggested.

4 Importing Functional Mock-up Units

All experiments in this paper are conducted in Hopsan, a distributed simulation environment developed at Linköping University [6]. It is based on the transmission line element method and uses distributed solvers. Components in Hopsan are quite similar to FMUs; they consist of a pre-compiled shared library file and an XML description file. The library file, however, is linked against Hopsan from where it inherits classes and can thus not be used standalone.

The process of importing an FMU to Hopsan is implemented in the following way, see figure 5. First the files are extracted to a temporary directory. Then the XML is parsed, including TLM specifications if present. Then the source file for the component library (`fmuLib.cc`) and the Hopsan component (`fmuName.hpp`) are generated and compiled to a shared library. A simple solver that uses the forward Euler method is also included, in order to be able to solve the equations. This would not be required with FMI for co-simulation, where the solver is included in the FMU. Finally, the XML description (`fmuName.xml`) is generated. The component can then be loaded from Hopsan and will then be available from the component library.

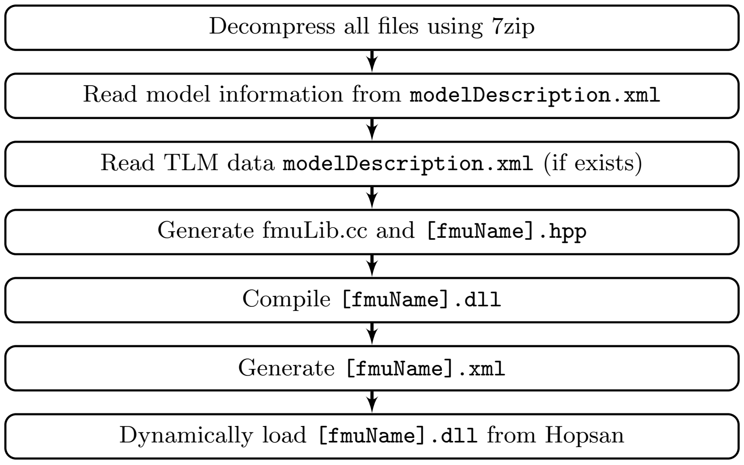
Decompress all files using 7zip
Read model information from modelDescription.xml
Read TLM data modelDescription.xml (if exists)
Generate fmuLib.cc and [fmuName].hpp
Compile [fmuName].dll
Generate [fmuName].xml
Dynamically load [fmuName].dll from Hopsan

Fig. 4: FMUs are imported to Hopsan by compiling a wrapper library.

Hopsan is an object-oriented simulation environment where everything is pre-compiled. Components are objects that can be connected by node objects [7]. During the simulation, each component solves itself independently. Therefore no compilation prior to simulation is required. The main simulation uses fixed time steps for communication between components and nodes. It is, however, up to each component to decide how to perform its calculations. Thus, it possible to use variable time steps inside a certain component. Theoretically, it is possible to use variable time steps also for the whole model, but previous experiments show that this is generally not worth the effort [8].

5 Exporting Models to Functional Mock-Up Units

Hopsan does not contain any equations or numerical solvers. The natural method would thus be to export an FMU for co-simulation. For compatibility with other software, however, the model exchange interface is used. The exported FMU does, however, actually solve itself and is basically working as an FMU for co-simulation.

According to the FMI specifications, only one shared library file is allowed. It is thus not possible to link against a pre-compiled Hopsan library file; the simulation core must be compiled into the FMU library. The model file is also included as a string variable in a header file and is loaded during initialisation.

Another factor which must be considered is that the FMI standard requires a plain C interface to ensure cross-compiler compatibility. Because the Hopsan simulation core is written in C++, a wrapper file is used to allow access to all required functions without using C++ features, such as classes and objects.

The export process begins with parsing the model and generating the XML description, together with the TLM extension if required. All source code files are then generated, including the model header file and the wrapper files. These files are then compiled along with the source code from the Hopsan simulation core and the required FMI source code. Finally, the binary and the XML description are compressed using 7zip. It is important to use the "deflate" compression method, to ensure compatibility.

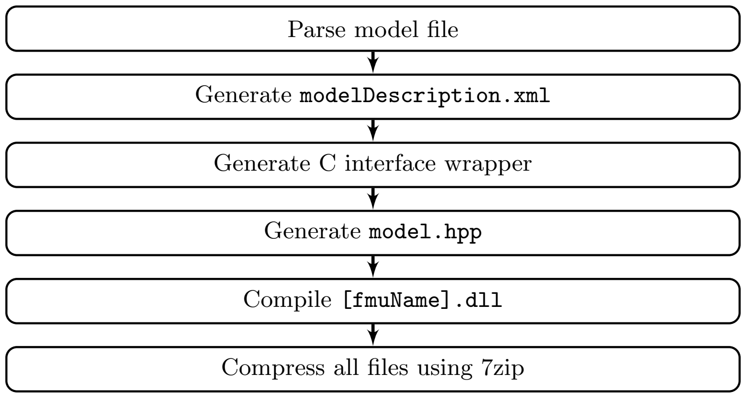
Parse model file
Generate modelDescription.xml
Generate C interface wrapper
Generate model.hpp
Compile [fmuName].dll
Compress all files using 7zip

Fig. 5: FMUs are exported from Hopsan by compiling the simulation core with a C interface using a wrapper.

FMUs exported from Hopsan are successfully validated by importing them into OpenModelica, Dymola, FMU SDK, and also back into Hopsan itself. They are verified with FMU Checker version 1.0.2. No errors or warnings are reported.

6 Example Simulation

In order to demonstrate the proposed method, an example model is created, see figure 6. It consists of a four-wheel vehicle with an engine, a mechanical gearbox, and a hydraulic transmission. The hydraulic system is modelled using built-in pre-compiled C++ components in Hopsan. The engine is modelled as a PI-controlled torque source with velocity feedback. As a demonstration, it is exported from Hopsan to an FMU and then imported back into Hopsan. The brake component is created in the same way. Models for the vehicle, the wheels, and the gearbox are all equation-based models created in OpenModelica, an open-source Modelica-based simulation tool [9]. They are then exported to Hopsan as FMUs. The vehicle consists of a linear inertia with a drag coefficient parameter. Wheels are modelled as rotating inertias with ports for drive shaft, brakes, and attachments in the vehicle. The gearbox is modelled as a rotating inertia with changeable gear ratio.

All components are connected through transmission line elements, representing the shafts and mechanical connections. This means that stiffness is replaced by characteristic impedances and a time delay, which results in a decoupled system with good wave propagation accuracy. Pre-defined components in Hopsan are used for this purpose.

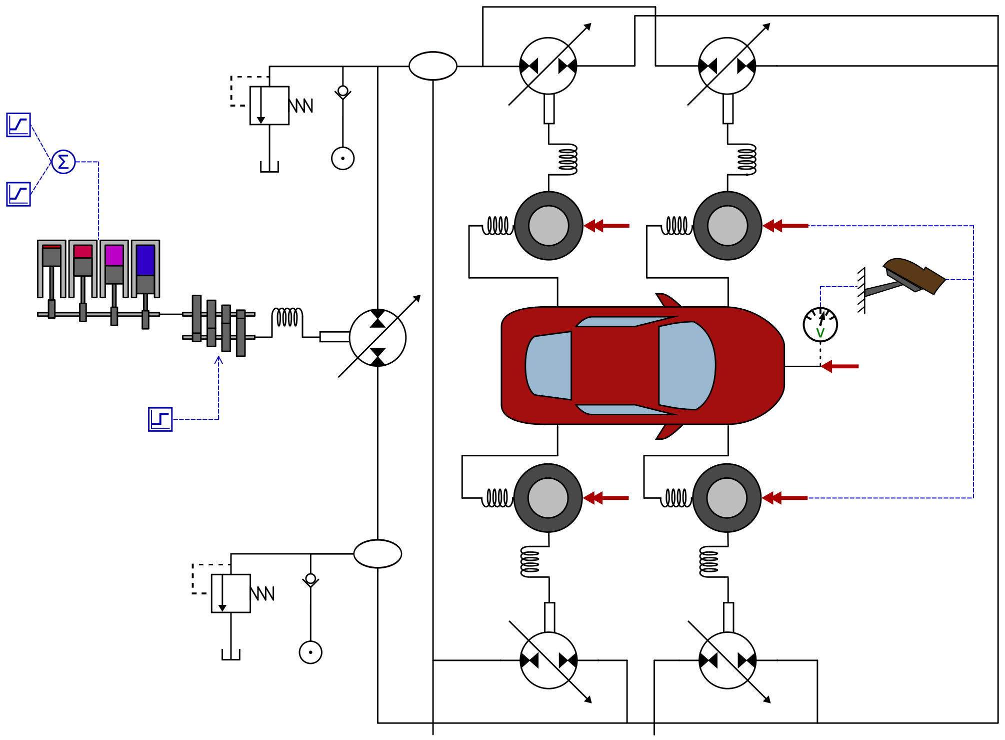

Fig. 6: An example model that describes a four-wheel vehicle with a simple hydraulic transmission is used to verify the proposed method.

In order to verify the functionality of the model, a simple drive cycle is simulated. The vehicle is first accelerated to 50 km/h and then to 70 km/h. It is then slowed down to 30 km/h and finally comes to a stop, see figure 7.

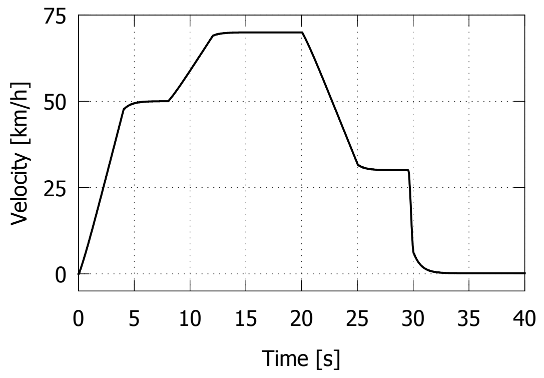

Fig. 7: A simple drive cycle is used to verify the functionality of the example model.

Load balancing is an essential aspect in parallel programming. If the work is not equally distributed over the threads, the speed-up will be limited by the slowest thread. In Hopsan this is solved by an automatic algorithm that measures the simulation time for each component over a few time steps before the actual simulation. This information is in turn used to distribute the components evenly over the simulation threads [5]. The average measured time per iteration for each sub-component type in the example model is shown in table 1. Note that these measurements are made on different sub-models and can thus not be used to compare the performance of different simulation tools.

Sub-model	**Time/iteration**
Wheel (OpenModelica, FMU)	1.192 μs
Gearbox (OpenModelica, FMU)	1.083 μs
Vehicle (OpenModelica, FMU)	0.574 μs
Brake (Hopsan, FMU)	0.476 μs
Engine (Hopsan, FMU)	0.189 μs
Relief Valve (Hopsan, built-in)	0.121 μs
Pump (Hopsan, built-in)	0.115 μs
Volume (Hopsan, built-in)	0.024 μs

Table 1: The simulation time for each sub-component is measured before the simulation, to achieve good load balancing.

The resulting distribution is shown in table 2. Components of C-type are generally much faster than those of Q-type. In this case they only required 5.8% of the total time. For this reason, only the threads for Q-type components are analysed.

	Thread 1	**Thread 2**	**Thread 3**	**Thread 4**
	Wheel 1	Wheel 2	Wheel 3	Wheel 4
	Gearbox	Vehicle	Pump 1	Pump 2
			Relief Valve 1	Relief Valve 2
			Pump 3	Pump 4
			Pump 5	Check Valve 1
				Check Valve 2
Total time:	2.275 μs	1.765 μs	1.644 μs	1.606 μs

Table 2: Sub-components are automatically distributed over the simulation threads.

As can be seen, a decent although not perfect load balancing is achieved. There are also overhead time costs from time measurements and thread synchronisation. Simulation time is, however, still more than twice as fast with four threads compared to with one thread. See table 3 for simulation times for 10,000 time steps with different numbers of processors. The time reduction from parallel simulation will increase when larger models are used. Theoretical maximum of speed-up is limited by the number of processor cores [5].

Threads:	**Simulation time:**
1	3307 ms
2	2091 ms
4	1466 ms

Table 3: Parallel execution reduces simulation time.

7 Conclusions

This paper shows that it is possible to combine the FMI standard with the transmission line element method. This makes it possible to simulate large aggregated models, consisting of sub-models from different modelling tools, in parallel on multi-core processors. Simulation time can then be significantly reduced. An interesting continuation could be real-time applications, where simulation performance is a critical aspect. Other possible future work could be to investigate higher level modelling methods for describing aggregated FMI models. Experiments were performed with FMI for model exchange using a simple solver. FMI for co-simulation would be more suitable, but is so for not supported by many simulation tools. Such difficulties will be easier to overcome with the FMI 2.0 standard, where co-simulation and model exchange will be harmonised.

8 Acknowledgements

This work was supported by ProViking research school and the Swedish Foundation for Strategic Research (SSF).

References

[1] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

[2] T Blochwitz, M Otter, J Akesson, M Arnold, C Clauß, H Elmqvist, M Friedrich, A Junghanns, J Mauss, D Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference, Munich*, 2012.

[3] D.M. Auslander. Distributed system simulation with bilateral delay-line models. *Journal of Basic Engineering*, pages 195–200, June 1968.

[4] P.B. Johns and M.A. O'Brian. Use of the transmission line modelling (T.L.M) method to solve nonlinear lumped networks. *The Radio And Electronic Engineer*, 50(1/2):59–70, 1980.

[5] R. Braun, P. Nordin, B. Eriksson, and P. Krus. High Performance System Simulation Using Multiple Processor Cores. In *The Twelfth Scandinavian International Conference On Fluid Power*, Tampere, Finland, May 2011.

[6] M. Axin, R. Braun, A. Dell'Amico, B. Eriksson, P. Nordin, K. Pettersson, I. Staack, and P. Krus. Next Generation Simulation Software Using Transmission Line Elements. In *Fluid Power and Motion Control*, Bath, England, October 2010.

[7] B. Eriksson, P. Nordin, and P. Krus. Hopsan NG, A C++ Implementation Using The TLM Simulation Technique. In *The 51st Conference On Simulation And Modelling*, Oulu, Finland, 2010.

[8] A. Jansson, P. Krus, and J-O Palmberg. Variable time step size applied to simulation of fluid power systems using transmission line elements. In *Fifth Bath International Fluid Power Workshop*, Bath, England, 1992.

[9] OpenModelica website. https://www.openmodelica.org/, July 2013.