TLMSimulator

A Framework for Composite Modelling and Co-simulation

April 2017

List of Contents

Chapter 1

TLMSimulator Co-Simulation Framework

A general framework for composite modeling and co-simulations has previously been designed [9]. The design goals for the simulation part of the framework were portability, simplicity to incorporate new simulation tools, and computational efficiency. These goals were realized by defining the following concepts and interfaces:

External interface. A named point on a mechanical object where position and velocity can be evaluated and reaction load (force and moment) applied. To guarantee numerical stability when utilising different numerical solvers in the co-simulation, only interfaces based on the *transmission line modelling* method, see Section 2, are currently supported.

Simulation manager. The central simulation engine. It is a stand alone program that reads an XML definition of the coupled simulation. It then starts *external model simulations* and provides the communication bridge between the running simulations. The external models only communicate with the simulation manager which acts as a broker marshalling information between the external models. Simulation manager sees every external model as a black box having one or several external interfaces. The information is then forwarded between external interfaces belonging to different external models. Additionally the simulation manager opens a network port for monitoring all communicated data.

Interface plug-in. A small C++ library having a single abstract class representing external interface for a specific simulation tool. The interface plug-in can be seen by an external model simulator as an external load that depends on position, velocity and time. The implementation of the plug-in handles the necessary communications with the simulation manager. It also handles necessary coordinate system transformations into the global composite model inertial system. All positions and velocities are transformed from the external-model (simulation-tool specific) inertial system to the global composite model inertial system. All reaction loads are translated back into the local inertial system. This constant transformation is stored in the composite model and sent to the corresponding interface plug-in on simulation start up.

External model simulator. Any simulation program that has incorporated the interface plug-in as a part of its model. A small script that takes the general parameters as input and starts the specific simulation tool is an additional requirement. This intermediate step is necessary since the simulation manager needs a common way to start all the components and each tool might have some specific start procedures.

1.1 Requirements on External Model Simulators

External models are associated with specialised simulation tools. Even though many simulation tools have interfaces to external functions, the interfaces differ between tools. Therefore it is first necessary that a software developer who is familiar with the particular tool architecture, designs and implements the *external interface* for each tool. That is, to create a tool specific wrapper for the *interface plug-in* of the simulation framework.

The following functionality is required from all simulation tools that implement the interface plug-in and want to participate in co-simulation:

- Possibility to start simulation externally or in batch mode. The developer of the tool specific interface must provide a start up program. This is used by the simulation manager to set-up global simulation parameters, that is, start time, end time, and max time step for each co-simulation participant. Regarding the maximum time step length: The current co-simulation communication protocol is based on the *transmission line modelling* method, see Section 2 for details. This method requires a certain communication time control, i.e., time steps need to be within a physically motivated limit.

- Possibility to integrate the *interface plug-in* into the tool specific adaptor. Note that the tool independent part of the plug-in is implemented in C++. Some tools require external functions to be implemented in, e.g., C or Fortran. In such cases the C++ code can be invoked from a C or Fortran function.

- Ability to deliver position, orientation and velocity of a point to the *interface plug-in* and receive the reaction load (force and moment) to be applied at this point.

- Ability to send information about the taken solver steps to the interface. This is important for variable time step solvers. Data is send from one co-simulation party to the other when a time step is completed, that is, if the solver, after many iterations, decided what step it will take next. This information needs to be send to the TLMPlugin.

- Correct handling of shutdown signals coming from the tool specific wrapper. In some cases the *simulation manager* needs to take down the simulation tool in a controlled way. This can be achieved by a tool specific API (*Application Programming Interface*) call or simply handling of exit signals.

Additionally it is desirable that the simulation tool can export surface geometry (graphics) for visualisation in the composite model environment. Surface geometry is not required for correct composite modelling but beneficial for visual model verification.

Most of the commonly used simulation tools offer some kind of external connection either through inter process communication (IPC), e.g., network sockets or remote procedure calls, or an application programming interface (API). Both options are acceptable for implementation of the interface plug-in as long as they fulfill the requirements above. The main focus of this work is on transient simulations of mechanical systems. Simulation tools that are of interest for this work are pure mechanical system and multi physics tools. All of the tools that have been considered for integration into the co-simulation framework comply with the requirements, some of the tools are shown in Table 1.1. Interface plug-ins for SKF's BEAST, SKF's Orpheus, MSC.ADAMS, Matlab/Simulink, and Modelica have successfully been implemented and tested.

Table 1.1: *List of potential simulators considered for TLM co-simulation. Possible implementation and type of the interface plug-in are also shown.*

Simulator	Implementation	Interface
BEAST (SKF in-house)	C++ implementation	TLM enabled control points (coordinate-systems)
MSC.ADAMS	C wrapper DLL (dynamic link library)	General force with sub-routine call
Matlab/Simulink	C wrapper	S-function interface
Modelica	C or Fortran wrapper	External function interface
Simpack	Fortran wrapper	SIMPACK User routine

1.2 External Model startup

The requirements on the external model simulators, that is, the simulation tools that are supposed to participate in a co-simulation are defined above, see Section 1.1. One requirement specifies that the simulation tool should be executable in batch mode, this is, the simulation manager should be able to start the simulation tool and pass certain parameters to the program.

The developer of the simulation tool specific adapter (TLMPlugin) should provide a start-up program/script that accepts the following command line parameters:

- *Model* - the name of the sub-model as presented in the composite model definition. This name typically corresponds to the component specific input file name.
- FromTime - start time for the simulation.
- ToTime - end time for the simulation.
- Step - maximum time step allowed for the simulation. This depends on the minimum TLM delay associated with one of the TLM links connected to the sub-model.

- Server:port - name of the host machine running TLM manager application and the TCP/IP port where TLM server is listening. This information is required for TLM-plugin initialization. It is provided by the TLM manager as the last argument to the start script.

A sample OpenModelica Linux start-up script could look like this:

```
#!/bin/sh
# OpenModelica TLM start-up script
# Start with 6 arguments:
# 1 XModelName (XModel directory)
# 2 start-time
# 3 end-time
# 4 max-time-step
# 5 server-name:port
# 6 model-file

OpenModelicaPath=/opt/OpenModelica
TLMModelicaPath=/opt/OpenModelica/TLMPlugin/Modelica
OMC_Cmd="${OpenModelicaPath}/bin/omc"
TLMCONFIGFILE=tlm.config
LD_LIBRARY_PATH=${OpenModelicaLibPath}/lib

echo Writing caseID $1 and server name $5 to file $TLMCONFIGFILE
echo $1 > $TLMCONFIGFILE
echo $5 >> $TLMCONFIGFILE
echo $2 >> $TLMCONFIGFILE
echo $3 >> $TLMCONFIGFILE
echo $4 >> $TLMCONFIGFILE

MOSFILE=$1.mos
MODELNAME=‘basename $6 .mo‘
INTERVAL_STR="($3-$2)/($4)"
INTERVAL_STR="scale=8;${INTERVAL_STR/e/*10^}"
INTERVALS=‘echo $INTERVAL_STR | bc‘

echo Writing $MOSFILE
echo // Autogenerated modelica script for TLM cosimulation > $MOSFILE
echo "setEnvironmentVar( \"MODELICAUSERCFLAGS\", \\
                         \"-I${TLMModelicaPath} \\
                         -L${TLMModelicaPath}/${ABI}\");" >> $MOSFILE
echo "loadModel(Modelica);" >> $MOSFILE
echo "loadFile(\"${TLMModelicaPath}/TLM.mo\");" >> $MOSFILE
echo "loadFile(\"$6\");" >> $MOSFILE
echo "getErrorString();" >> $MOSFILE
echo "checkModel($MODELNAME);" >> $MOSFILE
echo "getErrorString();" >> $MOSFILE
echo "translateModel($MODELNAME);" >> $MOSFILE
echo "getErrorString();" >> $MOSFILE
echo "simulate($MODELNAME, startTime=$2, \\
                           stopTime=$3, \\
                           numberOfIntervals=$INTERVALS, \\
                           method=\"dassl\", \\
                           outputFormat=\"plt\");" >> $MOSFILE
echo "getErrorString();" >> $MOSFILE

echo Starting OpenModelica with: $OMC_Cmd $MOSFILE
$OMC_Cmd $1.mos > $1.simlog
```

A sample OpenModelica Windows batch start-up script could look like this:

```
echo off
REM OpenModelica TLM start-up script
REM Start with 6 arguments:
REM 1 XModelName (XModel directory)
REM 2 start-time
REM 3 end-time
REM 4 max-time-step
REM 5 server-name:port
REM 6 model-file

set OpenModelicaPath=C:/OpenModelica1.8.1
set OMC_Cmd=%OpenModelicaPath%/bin/omc.exe
set TLMCONFIGFILE=tlm.config

cd %1

echo Writing caseID %1 and server name %5 to file %TLMCONFIGFILE%
echo %1 > %TLMCONFIGFILE%
echo %5 >> %TLMCONFIGFILE%
echo %2 >> %TLMCONFIGFILE%
echo %3 >> %TLMCONFIGFILE%
echo %4 >> %TLMCONFIGFILE%

set MOSFILE=%1.mos
for /F %%I in ("%6") do set MODELNAME=%%~nI
for /F %%I in ("%Mofile%") do set MODELNAME_WITH_MO=%%I
calc (%3-%2)/%4
call result.bat
set INTERVALS=%res%

echo Writing %MOSFILE%
echo // Autogenerated modelica script for TLM cosimulation > %MOSFILE%
echo setEnvironmentVar("MODELICAUSERLFLAGS", \\
                       "-L/c/OpenModelica/sources/TLMPlugin/Modelica/WINDOWS32"); >> %MOSFILE%
echo loadModel(Modelica); >> %MOSFILE%
echo loadFile("%6"); >> %MOSFILE%
echo checkModel(%MODELNAME%); >> %MOSFILE%
echo simulate(%MODELNAME%, startTime=%2, stopTime=%3, \\
              numberOfIntervals=%INTERVALS%, \\
              tolerance=0.000001, \\
              method="euler", \\
              outputFormat="plt", \\
              variableFilter="x"); >> %MOSFILE%

echo Starting OpenModelica with: %OMC_Cmd% %MOSFILE%
%OMC_Cmd% %1.mos > %1.simlog
```

Note that the above approach requires that the Modelica TLMPlugin reads the file “$TLMCONFIGFILE” in order to get the TLM parameters.

1.3 Co-Simulation

A composite model simulation environment has been created that is based on the previously defined co-simulation framework. The system design of the environment is shown in Figure 1.1. The environment provides:

- Generality, due to the general framework that allows for integration of many different simulation tools.
- A generic method of co-simulation based on a composite model.
- A general method of external model execution, i.e., all simulation tools involved are executed. Generality is achieved by a platform independent start up command that takes care of possible actions, i.e., remote login, file transfer, or set up of a parallel simulation environment.

- Communication and data transfer between the different external models.
- Data monitoring for analysis and post processing.
- Controlled simulation termination where all external models are taken down in a correct way. This includes error handling due to external model failure or network problems.

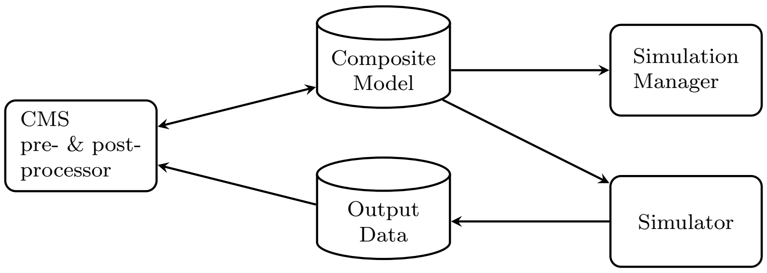


Figure 1.1: The system design for the composite model simulation environment.

In addition to the *simulation manager* a composite model *simulator* is included in the design that can start and monitor the composite model simulation. The simulation manager has been designed to act as a network server that allows several clients to connect and monitor the data flow between the interconnected external model simulators. The *simulator* can connect to this port. All data received by the simulation manager from the external model simulators is passed on to the simulator that stores results in the composite model output file. This client server architecture has some advantages over an integrated solution where the simulation manager is part of the simulator:

- The simulator and simulation manager can run on different machines, this is useful, for instance, if the simulation start up must be performed on a machine other than the user's local machine.
- Other processes can connect to the simulation manager's monitoring port. This allows, for instance, to visually monitor the co-simulation.

The simulation-manager manages start up of the different simulation tools (external model simulators) and communication between these tools, as defined by the simulation framework, see Figure 1.2. Each simulation tool implements the interface plug-in that handles the necessary communication with the simulation manager. The manager keeps an internal interconnection table for all connected external interfaces and forwards all received packages accordingly.

Simulation results can be analysed in the post-processing applications, i.e., a two dimensional plotting program and a three dimensional visualisation tool, i.e., the composite model editor (CME). Data animation of system dynamics is possible to a limited extent based on the data that is exchanged in the external interfaces.

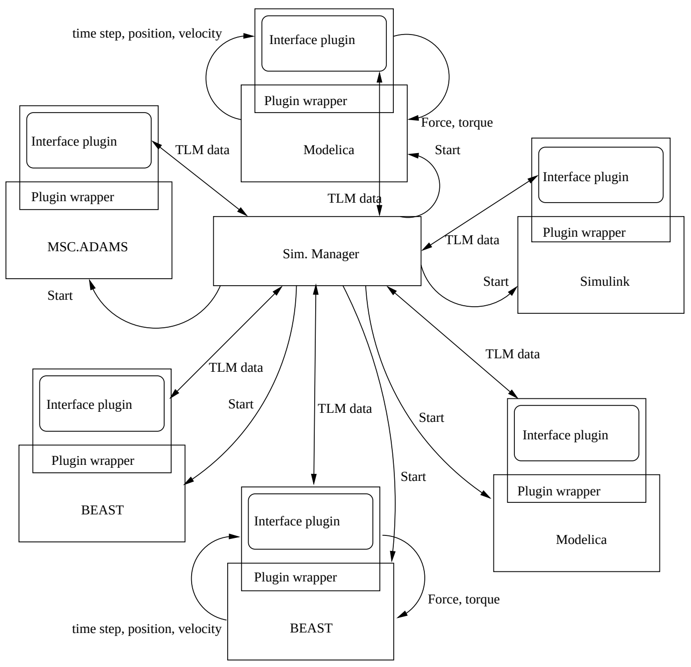


Figure 1.2: The simulation manager handles start up and communication between the tool specific simulators.

1.4 Composite Model XML Files

The TLM default implementation contains a XML composite model reader, that can be used by the simulation manager to read the composite model from an XML file.

A composite model contains the following XML nodes:

Model is the top composite model node that contains a list of `SubModels` and `Connections`.

SubModels is the node that contains the list of all `SubModel` nodes.

SubModel is used for the external models. There is a `SubModel` node for each external model that participates in the co-simulation. It defines a sub-model name used in the composite model and the start method and the simulation tool specific model files that needed to run the simulation. Each `SubModel` node also contains a list of `InterfacePoint` nodes.

InterfacePoint nodes are used to specify the TLM interfaces of each external model, that is, each `SubModel`. It defines the name of a certain interface in the external model.

Connections is the node that contains the list of all `Connection` nodes.

Connection nodes define connections between two connected `InterfacePoints`, that is, a connection between two TLM interfaces. TLM parameters are specified for each connection.

SimulationParams node defines global composite model simulation parameters. Total simulation start time, end time, and network port are defined in this node.

Here a sample composite model XML file for a Modelica with Beast co-simulation:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- The root node is the composite model -->
<Model Name="Pendulum">

  <!-- List of connected sub-models -->
  <SubModels>

    <!-- Each sub-model defines an external simulation model,
         the following is an Modelica model that is started with
         the StartTLMModelica command and the model file shaft1.mo -->
    <SubModel Name="shaft1"
              StartCommand="StartTLMModelica"
              ExactStep="0"
              ModelFile="shaft1.mo">

      <!-- TLM interface points for SubModel shaft1.
   For each interface one can define position and
          orientation vectors. These are mainly useful for
          3D modeling. Orientation is defined as three angles
          aroung x, y, and z axis "x,y,z" in radians. Angle321
          defines rotation order 321 (z,y,x) of the three angles.
   Position and orientation is defined with respect to the
   external models inertial system. -->
      <InterfacePoint Name="tlm"
      Position="0,-0.1,0"
      Angle321="0,0,0"/>
    </SubModel>

    <!-- SubModel brg1. This is a BEAST model. Also for sub-models
 one can define position and orientation vectors. These are
 useful for translations between the different models.
 Position and orientation is defined with respect to the composite
 models global inertial system. -->
    <SubModel Name="brg1"
              StartCommand="StartTLMBeast"
              ExactStep="0"
              ModelFile="dgbb"
              Position="0,0,0"
              Angle321="1.5708,0,0">

      <!-- TLM interface points for SubModel brg1 -->
      <InterfacePoint Name="bIR‘cs1"/>
      <InterfacePoint Name="bER‘cs1"/>
    </SubModel>

    <!-- SubModel shaft2. This is another Modelica model. -->
    <SubModel Name="shaft2"
              StartCommand="StartTLMModelica"
              ExactStep="0"
              ModelFile="shaft2.mo">

      <!-- TLM interface points for SubModel C -->
      <InterfacePoint Name="tlm"/>
    </SubModel>
  </SubModels>

  <!-- List of TLM connections -->
  <Connections>
    <!-- For each connections individual TLM parameters are defined.
 Note, that the maximum step size must be smaller than the
 shortest delay of all TLM connections for a single simulation
 tool. This is taken care of by the tlmmanager. -->
    <Connection From="shaft1.tlm" To="brg1.bER‘cs1"
 Delay="1e-3" Zf="1e4" Zfr="1e2" alpha="0.2"/>
    <!-- Each connections defines which interface of which models are
         connected.
 In these interfaces forces are exchanged, see TLM definition. -->
    <Connection From="shaft2.tlm" To="brg1.bIR‘cs1"
Delay="1e-3" Zf="1e4" Zfr="1e2" alpha="0.2"/>
  </Connections>

  <!-- "Global" parameters for the co-simulation.
       Typically the overall simulation time is defined here.
       This information is propageted to all simulation tools.
       Note, some additional parameters for network port and
       timeout can be defined here as well. But it is recommended
       to not define this "system dependent settings" here.
       Use some system setup instead that can be given or read by
       the tlmmanager.  -->
  <SimulationParams StartTime="0.0"
                    StopTime="1.0"/>

</Model>
```

Chapter 2

TLM background theory

The method that is used to enable interaction between dynamic models in the composite model simulation is *transmission line modelling* (TLM) [1] [5] [2] [11]. TLM uses physically motivated time delays to separate the components in time and enable efficient co-simulation. Only TLM connections between two external interfaces are currently supported by the composite model simulation environment because the TLM method gives numerical stability.

The TLM (*Transmission Line Modeling*) method, also called Bilateral Delay Line Method [1], exploites the fact that all physical interactions in nature have finite propagation speed.

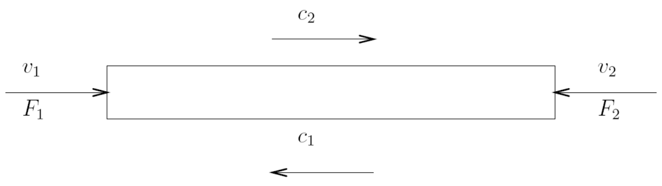


Figure 2.1: Delay line with the passing wave variables c_1 and c_2 and velocity variables v_1 and v_2.

A basic one-dimensional transmission line is shown in Figure ??. For the mechanical case the line is basically an ideal elastic medium with force waves c_1 and c_2 going between its ends. The input disturbances are velocities v_1 and v_2 and the forces from the transmission line F_1 and F_2.

Note that the springs in our implementation are assumed to be isotropic. That is no cross-term waves are generated when working in 2D and 3D. See [5] for further discussions.

If the line delay is set to T_{TLM} and its impedance to Z_F then the govering equations are:

$$\begin{aligned} c_1(t) &= F_2(t - T_{TLM}) + Z_F\, v_2(t - T_{TLM}) \\ c_2(t) &= F_1(t - T_{TLM}) + Z_F\, v_1(t - T_{TLM}) \\ \\ F_1(t) &= Z_F\, v_1(t) + c_1(t) \\ F_2(t) &= Z_F\, v_2(t) + c_2(t) \end{aligned} \tag{2.1}$$

The equations show that the two simulation systems are decoupled with the delay time T_{TLM}. Simulation framework can utilize this decoupling to enable efficient communications during co-simulation.

Representing the TLM connection with a simple model of a steel beam, the stiffness coefficient can be computed as (see [5]):

$$k = \frac{EA}{L_0} \tag{2.2}$$

where E is Young's modulus, A is the cross section area and L_0 the length of the beam.

The impedance Z_F has a relation to the spring constant k, $Z_F = kT_{TLM}$. The impedance factor can then be formulated as a function of the area and length of the steel rod according to

$$Z_F = \frac{EAT_{TLM}}{L_0} \tag{2.3}$$

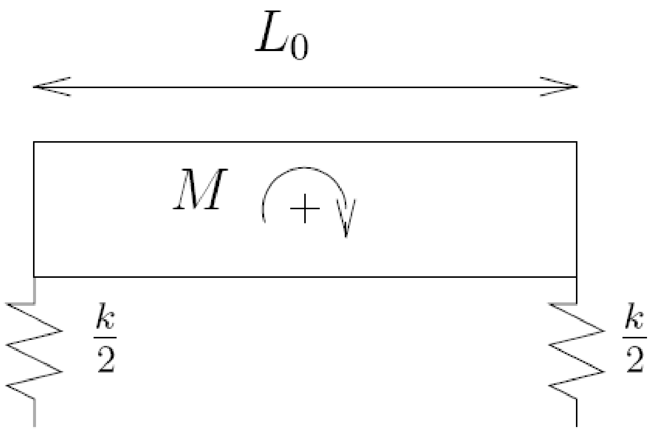


Figure 2.2: Estimating the rotational stiffness.

To get the stiffness and impedance for the rotational degrees of freedom one can use the already computed stiffness k. If the arrangement depicted in Figure ?? is assumed, then:

$$k_\phi = \frac{M}{\delta_\phi} = 2\frac{(k/2)\delta_\phi(L_0/2)^2}{\delta_\phi} = \frac{kL_0^2}{4} \tag{2.4}$$

and the impedance for the rotation:

$$Z_{FR} = \frac{1}{4}Z_F L_0^2 \tag{2.5}$$

The time constant T_{TLM} can be computed using the speed of sound for the medium:

$$T_{TLM} = \frac{L_0}{v_{medium}} \tag{2.6}$$

It can be shown that the TLM element also introduces a (*parasitic mass*) that can be viewed to be outside the simulated system [5]. The total mass for the combined systems must therefore also include the parasitic mass of the TLM element in order to make, e.g., the energy conservation formulas correct. This mass depends on the impedance factor and the time delay factor

$$m_p = Z_F T_{TLM} \tag{2.7}$$

This implies that if the impedance factor Z_F is increased, the parasitic mass will increase if the synchronization delay T_{TLM} is not decreased. If the parasitic mass is large it may influence the system behavior and can not be neglected. Note, that for the simple beam case when TLM parameters are computed according to Equations 2.3 and 2.6 the parasitic mass is equivalent to the mass of the beam ($\rho AL0$, where ρ is the material density).

For practical purposes (see [13]) one can use the parameters of a material cube with an edge given by *characteristic distance* L_0. Equations 2.2 and 2.4 can then be used to compute the translational and rotational stiffnesses:

$$\begin{aligned} k &= \tfrac{EL_0^2}{L_0} = EL_0 \\ k_\phi &= \tfrac{kL_0^2}{4} = \tfrac{EL_0^3}{4} \end{aligned} \tag{2.8}$$

To give a concrete example, let us assume that connection medium is steel and the characteristic length is $L_0 = 0.1$. Steel has Young's modulus $E = 210GPa$ and the speed of sound in steel is $v_{steel} = 5180m/s$. The TLM parameters then can be computed:

$$\begin{aligned} T_{TLM} &= L_0/v_{steel} \approx 2 * 10^{-5} \\ Z_F &= EL_0 T_{TLM} \approx 2 * 10^5 \\ Z_{FR} &= \tfrac{1}{4} Z_F L_0^2 \approx 500 \end{aligned} \tag{2.9}$$

Calculations like these give approximate values of the stiffness and the time delay of the TLM element. This gives a background for selecting the TLM line delay and impedance parameters. In cases when required T_{TLM} becomes a limiting factor, while the TLM link stiffness is much higher than the stiffnesses used in the sub-models, a lower stiffness and larger T_{TLM} may be considered.

The elastic medium that is modelled with the TLM element introduces oscillation frequencies (standing waves) given by:

$$f_{TLM,i} = \frac{i}{2\, T_{TLM}}, \quad i = 1, 2, 3, ... \tag{2.10}$$

The basic TLM model has no damping which can result in unwanted vibrations during simulation. In [5] a low pass filtering of the TLM charateristics is recommended:

$$c_{\text{filtered}}(t) = c_{\text{filtered}}(t - T)\ \alpha + c(t)\ (1 - \alpha) \tag{2.11}$$

The filtering is controlled by a damping constant α. The recommended value according to [5] is 0.2.

Chapter 3

TLMManager System Design

The TLMManager is responsible for managing the TLM based co-simulation. The main component used in the TLMManager is an instance of *ManagerCommHandler* that takes a composite model as input.

```
    //! Constructor.
    ManagerCommHandler(CompositeModel& Model):
                       MessageQueue(),
                       Comm(Model.GetComponentsNum(),
                            Model.GetSimParams().GetPort()),
                       TheModel(Model),
                       MonitorConnected(false),
                       CommMode(CoSimulationMode),
                       monitorInterfaceMap(),
                       monitorMapLock(),
                       runningMode(StartUpMode),
                       exceptionMsg(""),
                       exceptionLock()
    {
    };
```

A *CompositeModel* instance can be generated in different ways. The default TLM co-simulation implementation contains an XML composite model reader, see also Figure ??. For details about the XML composite model description see Section 1.4.

The *CompositeModelReader* parses the XML composite model file and initializes *CompositeModel* and *SimulationParams* data structures, that is, set-up co-simulation parameters. Simulation parameters define global co-simulation settings, e.g., start and end time. Composite model parameters define the co-simulation components (simulation models) and their interconnection.

The *ManagerCommHandler* is then started, that is, the *Run()* function is invoked. This function initialized the different threads that are used during the co-simulation:

```
// Run method executes all the protocols in the right order:
// Startup, Check then Simulate
void ManagerCommHandler::Run(CommunicationMode CommMode_In) {
    CommMode = CommMode_In;

    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr,  PTHREAD_SCOPE_SYSTEM);
    pthread_t reader, writer;

    // Start the minitoring thread
    pthread_t monitor;
    if(CommMode == CoSimulationMode) {
        pthread_create(&monitor, &attr, thread_MonitorThreadRun, (void*)this);
    }

    // start the reader & writer threads
    pthread_create(&reader, &attr, thread_ReaderThreadRun, (void*)this);

    pthread_create(&writer, &attr, thread_WriterThreadRun, (void*)this);

    // Wait until all threads are finished.
    if(CommMode == CoSimulationMode) {
        pthread_join(monitor, NULL);
    }

    pthread_join(reader, NULL);
    pthread_join(writer, NULL);
}
```

Note, that all communication between the co-simulation participants (simulators with models) is going through the TLM manager. This communication is handled by the threads that are started in the *Run()* function. The following threads are started:

The reader thread initialized the co-simulation, processes incoming messages, and creates messages to be sent. This threads first task is to run the *RunStartupProtocol()* that starts all the simulation tools that are participating in the co-simulation and runs the initiation protocol. After that the thread goes into message passing mode. Messages from one co-simulation participant to another are taken by the reader thread and marshaled to the receiver, that is, they are converted into outgoing messages.

The writer thread processes all outgoing messages. It loops through the queue of outgoing messages and distributes them to the correct receivers.

The monitor thread copies and forwards all outgoing messages to any connected monitoring process. This can be used to monitor the co-simulation. If no monitoring process is connected no copying and forwarding will take place.

The *ManagerCommHandler* collaborates with different classes to handle co-simulation start-up and communication, see also Figure 3.2. *TLMManagerComm* is responsible for all socket communications on the TLMManager side. The *TLMMessageQueue* handles the queue for outgoing messages.

3.0.1 Co-Simulation Start-up

During co-simulation start-up the TLMManager starts the different co-simulation components (external models) and then waits until all components have registered themselves. This is done with a simple initialization protocol.

First the components are started using `TLMComponentProxy::StartComponent(...)`. This is done using the OS specific execution method, for instance, with *fork()* and *execlp(...)* on Linux:

```
  execlp( StartCommand.c_str(), StartCommand.c_str(),
          Name.c_str(),
          startTime.c_str(),
          endTime.c_str(),
          strMaxStep.c_str(),
          serverName.c_str(),
          ModelName.c_str(),
          NULL );
```

The start command is the start-up script that must be provided for each specific simulation tool. It takes a couple of parameters as specified in the composite model XML file:

Name of the component as specified in the composite model XML file.

Start time of the simulation in seconds, typically 0.

End time of the simulation in seconds.

Max step for the solver to take. This is a limit set by the TLM interfaces of the specific component. The solver is not allowed to take larger steps.

Server name is typically the hostname or IP address that the TLMManager is running on and the port number that it is using. For instance *163.157.1.23:1111*.

Model name of the simulation model. Typically the input file needed to start the external model simulation.

3.0.2 External Model Initialization

After external model start-up, the TLMManager waits until all external models have registered themselves. This is performed in *ManagerCommHandler::RunStartupProtocol()*. Here an outline of the code:

```
void ManagerCommHandler::RunStartupProtocol() {
    ...
    // Start the external components forming "coupled simulation"
    TheModel.StartComponents();

    // Setup timer
    tTM_Info tInfo;
    TM_Init(&tInfo);
    TM_Start(&tInfo);

    while( (numToRegister > 0) ||
           ( numCheckModel < TheModel.GetComponentsNum()) ) {
        ...

        // Check all the registered components if they send
        // interface registration messages.
        for(int iSock =  TheModel.GetComponentsNum() - 1; iSock >= 0 ; --iSock) {
            ...
            TLMMessage* message = MessageQueue.GetReadSlot();
            message->SocketHandle = hdl;
            TLMCommUtil::ReceiveMessage(*message);

            if(message->Header.MessageType == TLMMessageTypeConst::TLM_CHECK_MODEL) {
                // This component is done with registration. It's will wait for others
                TLMErrorLog::Log(string("Component ") + comp.GetName() + " is ready to simulation");;

                comp.SetReadyToSim();
                numCheckModel++;
            }
            else if(message->Header.MessageType == TLMMessageTypeConst::TLM_REG_PARAMETER) {
                TLMErrorLog::Log(string("Component ") + comp.GetName() + " registers parameter");

                Comm.AddActiveSocket(hdl);
                ProcessRegParameterMessage(iSock, *message);
                MessageQueue.PutWriteSlot(message);
            }
            else {
                TLMErrorLog::Log(string("Component ") + comp.GetName() + " registers interface");;

                Comm.AddActiveSocket(hdl); // expect more messages
                ProcessRegInterfaceMessage(iSock, *message);
                MessageQueue.PutWriteSlot(message);
            }
        }

        // Check if a new connection is waiting to be accepted.
        if((numToRegister > 0) && Comm.HasData(acceptSocket)) {
            int hdl = Comm.AcceptComponentConnections();
            TLMMessage* message = MessageQueue.GetReadSlot();
            message->SocketHandle = hdl;

            if( !TLMCommUtil::ReceiveMessage(*message) ){
                TLMErrorLog::FatalError("Failed to get message, exiting");
                abort();
            }

            ProcessRegComponentMessage(*message);

            MessageQueue.PutWriteSlot(message);
            numToRegister --;
            if(numToRegister == 0)
                TLMErrorLog::Log("All expected components are registered");

            Comm.AddActiveSocket(hdl);
        }

        if(numToRegister)  // still more connections expected
            Comm.AddActiveSocket(acceptSocket);

    }
}
```

The above method has two main tasks, first to check for new connections from any external model. Second to check if already registered components need to register any TLM interfaces.

The initialization protocol looks the following way:

1. An external model send a TLMMessageTypeConst::TLM_REG_COMPONENT message to the TLMManager.
2. The TLMManager answers with the same message but also sets the data size of the message to 0.
3. The external model send an interface registration message (TLMMessageType-Const::TLM_REG_INTERFACE).
4. The manager answers with the same message head and adds the connection parameters as stored in the composite model XML file to the data section of the message.
5. The external model repeats the interface registration for all its external (TLM) interfaces.

6. Finally the external model sends a TLMMessageTypeConst::TLM_CHECK_MODEL that tells that it is ready to simulate.
7. The TLMManager puts the component into *ready for simulation* mode.

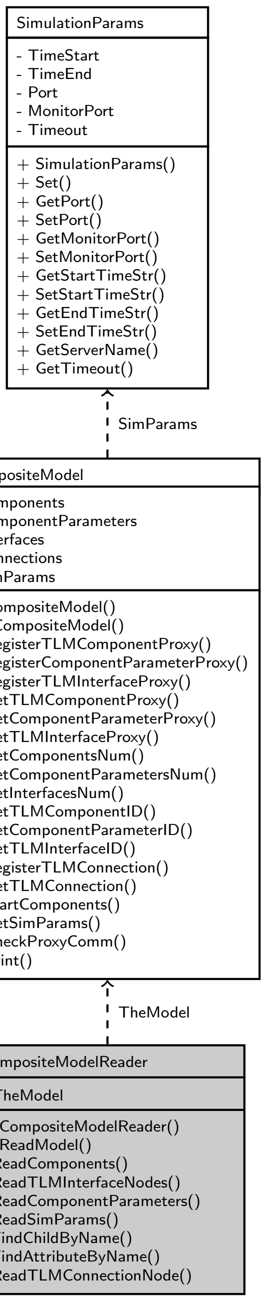


Figure 3.1: UML collaboration diagram of the CompositeModelReader class. The CompositeModelReader initialized the CompositeModel data structure.

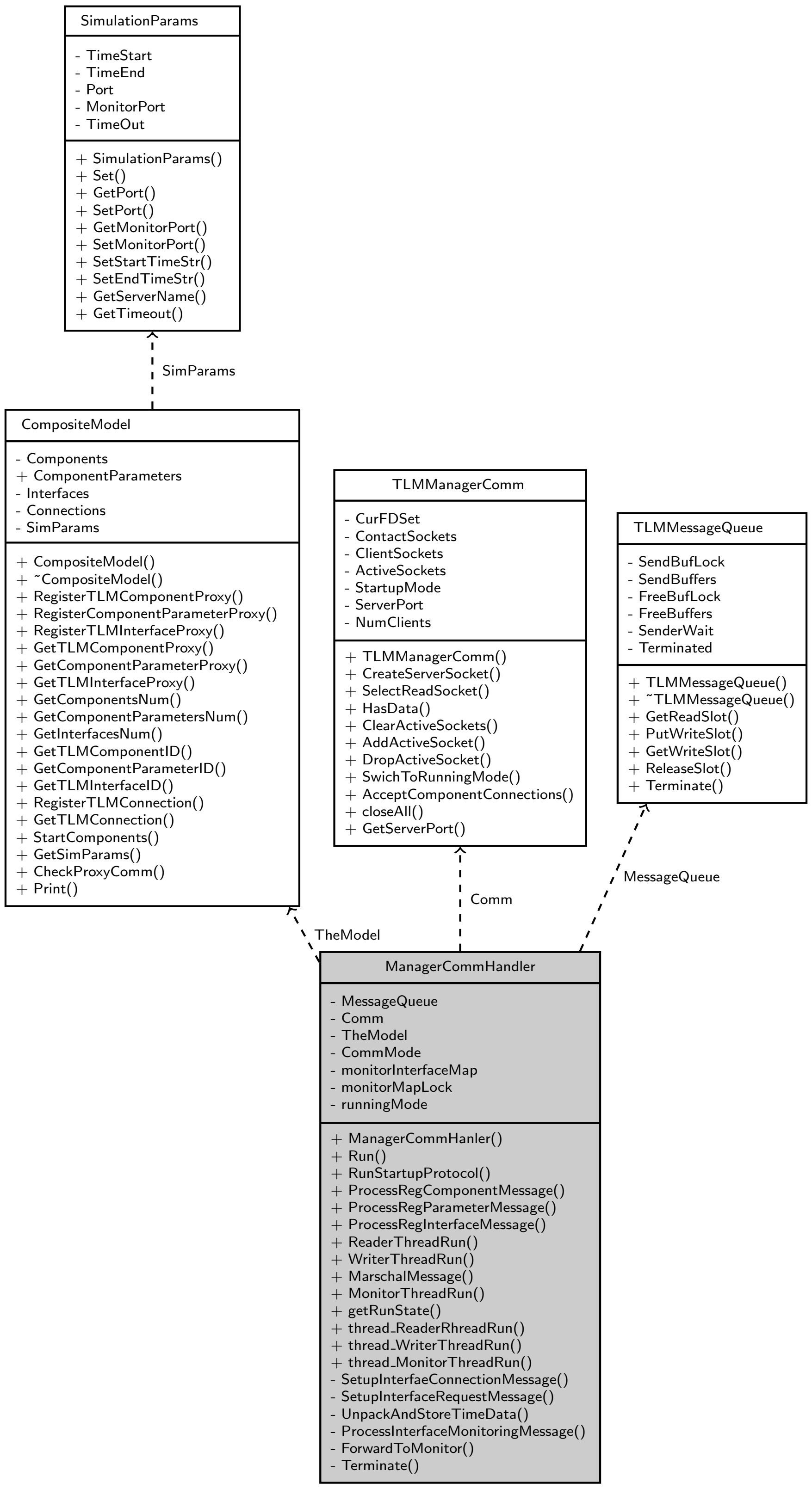


Figure 3.2: ManagerCommHandler collaboration and class diagram.

Chapter 4

Simulink TLM Plugin

The Matlab/Simulink TLMPlugin implementation is based on a Matlab S-function interface.

```
/*
 * tlmforce.c: Based on 'C' template for a level 2 S-function.
 *
 *  -------------------------------------------------------------------------
 *  | See matlabroot/simulink/src/sfuntmpl_doc.c for a more detailed template |
 *  -------------------------------------------------------------------------
 *
 * Copyright 1990-2000 The MathWorks, Inc.
 * $Revision$
 */
```

TLM delay is set in the initialization of the S-Function interface function *mdlInitializeSampleTimes(...)*.

```
/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    double sTime, eTime, timeStep;
    TLM_InterfaceReg::GetInstance(false)->GetSimParameters(sTime, eTime, timeStep);
    // true or false in GetInstance(...) enables/disables debug output

    TLMErrorLog::Log("Set sample time to " + ToStr(timeStep));

    /* Set TLM delay here! */
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);

    /* Set TLM delay here! */
    ssSetSampleTime(S, 1, timeStep);
    ssSetOffsetTime(S, 1, 0.0);
}
```

Note, that the first call to *TLM_InterfaceReg::GetInstance(...)* initialized the TLM plugin, that is, it reads the TLM configuration parameters form the config file.

Communication with Simulink is based on the C structure *SimStruct*. Initially we need to setup the size (of the data that we want to communicate in the structure. This is done in the S-Function interface function *mdlInitializeSizes(...)*.

```
/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl_doc.c for more details on the macros below */

    ...

    /* input ports are: */
    /* position[3]      Interface position data */
    /* orientation[3x3] Interface rotation matrix */
    /* speed[3]         Interface translational velocity */
    /* ang_speed[3]     Interface angular velocity */

    if (!ssSetNumInputPorts(S, 4)) return;
    /* Set size of input arrays */
    ssSetInputPortWidth(S, 0, 3);
    ssSetInputPortWidth(S, 1, 9);
    ssSetInputPortWidth(S, 2, 3);
    ssSetInputPortWidth(S, 3, 3);

    ...

    /* output ports are: */
    /* force[3]  Output force */
    /* moment[3] Output moment */

    if (!ssSetNumOutputPorts(S, 4)) return;
    ssSetOutputPortWidth(S, 0, 3);
    ssSetOutputPortWidth(S, 1, 3);
    ssSetOutputPortWidth(S, 2, 3);
    ssSetOutputPortWidth(S, 3, 9);

}
```

The force calculation in the TLM interface takes place in the S-Function interface function *mdlOutputs(...)*. This function is called by Simuink for a given time instance. The motion of the interface can be extracted from the *SimStruct* structure that has been setup before, see *mdlInitializeSizes* above. The resulting load (force and moment) is stored in the same structure.

```
/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    /* inputs: */
    /* position[3]      Interface position data */
    /* orientation[3x3] Interface rotation matrix */
    /* speed[3]         Interface translational velocity */
    /* ang_speed[3]     Interface angular velocity */
    double *R = (double*)ssGetInputPortSignal(S,0);
    double *A = (double*)ssGetInputPortSignal(S,1);
    double *vR = (double*)ssGetInputPortSignal(S,2);
    double *Omega = (double*)ssGetInputPortSignal(S,3);

    const char* name = ssGetPath(S);
    real_T time = ssGetT(S);

    int ifID = TLM_InterfaceReg::GetInstance()->GetInterfaceID(name);
    /* output */
    double force[6];
    TLMTimeData CurTimeData;

    if( ifID >= 0 ) {
        TLM_InterfaceReg::GetInstance()->GetPlugin()->GetForce(ifID,
                                                               time,
                                                               R,
                                                               A,
                                                               vR,
                                                               Omega,
                                                               force);

        /* Get Position and Orientation */
        TLM_InterfaceReg::GetInstance()->GetPlugin()->GetTimeData(ifID,
                                                                  time,
                                                                  CurTimeData);

    }
    else {
        /* Not connected */
        for( int i=0 ; i<6 ; i++ ) {
            force[i] = 0.0;
        }
    }

    /* ------- store the result ------- */

    /* Force & Moment */
    real_T  *f = ssGetOutputPortRealSignal(S,0);
    real_T  *m = ssGetOutputPortRealSignal(S,1);

    for( int i=0 ; i<3 ; i++ ){
        f[i] = force[i];
        m[i] = force[i+3];
    }

    /* Position & Orientation */
    real_T  *R_TLM = ssGetOutputPortRealSignal(S,2);
    real_T  *A_TLM = ssGetOutputPortRealSignal(S,3);

    for( int i=0 ; i<3 ; i++ ){
        R_TLM[i] = CurTimeData.Position[i];
        A_TLM[i] = CurTimeData.RotMatrix[i];
        A_TLM[i+3] = CurTimeData.RotMatrix[i+3];
        A_TLM[i+6] = CurTimeData.RotMatrix[i+6];
    }

}
```

TLM::SetMotion(...) is called when we want to update the states for a "final" integration step. In the S-Function interface this is handled in the function *mdlUpdate(...)*. This triggers the communication with the connected simulation tools in order to propagate the states.

```
/* Function: mdlUpdate ======================================================
 * Abstract:
 *    This function is called once for every major integration time step.
 *    Discrete states are typically updated here, but this function is useful
 *    for performing any tasks that should only take place once per
 *    integration step.
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    if( ssIsSampleHit(S, 1, tid) ){
        double *R = (double*)ssGetInputPortSignal(S,0);
        double *A = (double*)ssGetInputPortSignal(S,1);
        double *vR = (double*)ssGetInputPortSignal(S,2);
        double *Omega = (double*)ssGetInputPortSignal(S,3);

        const char* name = ssGetPath(S);
        real_T time = ssGetT(S);

        int ifID = TLM_InterfaceReg::GetInstance()->GetInterfaceID(name);

        if( ifID >= 0 ){
          // Send data to the Plugin
          TLM_InterfaceReg::GetInstance()->GetPlugin()->SetMotion(ifID,
                                                                   time,
                                                                   R,
                                                                   A,
                                                                   vR,
                                                                   Omega);
        }
    }
}
```

Chapter 5

MSC.Adams TLM Plugin

Important note:
The current version of the co-simulation environment does not propagate delay time (maximum time step) from the composite model to the MSC.ADAMS models automatically. Instead these values must be set in the Adams Command File (.acf). See Section 8.3 for details.

MSC.Adams supports a C subroutine application programming interface (API) that can be integrated with your C/C++ code. It it is based on some C header files, for instance:

```
// portability header from ADAMS
#include "userPortName.h"

// C-callable subroutines from ADAMS
#include "utilCcallable.h"
```

The MSC.Adams TLMplugin implementation is based on the Adams *GFOSUB* subroutine. This subroutine can be called from a general force element in Adams.

```
// File gfo_wrapper.c
//
// GFOSUB ADAMS/Solver user subroutine interface implementation
//

// GFOSUB - is an interfacec function for general 6-component force tensor
// that is 3-component force & 3-component torque.
//
// id -  id of this external function
// time - current time. Note that adaptive step solver is used in ADAMS
//    Unsuccessful steps are possible. TIMGET function can be used to check
//    is the last call was a final RHS call.
// par - parameters to the function - for GFOSUB expected:
//    a marker, i.e. one parameter
// nPar - number of parameters - should be 1 or 2. If second parameter exists, there will be debug output.
// dflag - true if Jacobian calculation is in progress, i.e. small change of inputs
//      For TLM purposes such calls can be made much faster since TLM force
//       is dependent only on time and not on the states.
// iflag - (init flag) true if solver is only after dependencies between vars
//      We need to make sysary calls to the motion variables that TLM is dependent on.
// result - array of 6 elements with the resulting force
VOID_FUNCTION GFOSUB(int *id, REAL *time, REAL *par, int *nPar, BOOL *dflag, BOOL *iflag, REAL *result){

...

}
```

The *c_sysary* function lets a user-written subroutine read information from ADAMS. Here we need it to get the displacement and velocity.

```
    //  c_sysary(char *fncnam, int *ipar, int nsize, REAL *states, int *nstate,
    //      BOOL *errflg);
    // Input:
    //  fncnam - name of the function, "DISP" for displacement
    //  ipar, nsize - list of markers and the number of markers
    //    DISP requires 1 to 3 markers where the distance is measured
    //    to marker 1 origin from marker 2 (reference)
    //    in coordinate system of marker 3 (basis) Setting markers
    //     2 and 3 to "0" gives global inertia system as reference.
    //     Markers 2 & 3 are optional.
    //    For our case we use the marker submitted to GFOSUB in parameters
    //    and ground.
    // Output:
    //   states, nstate - variable values and the number of variable returned
    //   For "DISP" 6 variables giving position and orientation are returned
    //   The orientation is given by Psi, Theta, Phi (ADAMS/Solver Euler angles)
    //   c_rcnvrt(char *sys1, REAL *coord1, char *sys2, REAL *coord2, int *istat);
    //   can be used to convert those to Euler parameters by specifying
    //   sys1 = 'EULER' and sys2 = 'EULPAR' or directly to rotation matrix
    //   (columnwise) by specifying sys2 = 'COSINES'.
    //   errflg - returns TRUE on error.

    // Get displacement
    c_sysary ("DISP", markers, 3, disp, &ns, &errflg);

    // Convert the angles from ADAMS "standard" Euler to rotation matrix (9-components)
    c_rcnvrt("EULER",disp+3,"COSINES",rot,&errflg);

    // Get velocity
    c_sysary ("VEL",markers,4,vel,&ns,&errflg);
```

Adams also supports unit scales. This is useful since we assume SI units in the TLM interface. Thus, we can convert into the correct unit.

```
    // Units scaling. See gtunts in ADAMS manual
    BOOL existsUnits;
    double scales[4];
#define UNIT_SCALE_TIME 0
#define UNIT_SCALE_LENGTH 1
#define UNIT_SCALE_FORCE 2
#define UNIT_SCALE_MASS 3
    char units[3 * 4];

    // gtunts retunes the unit scales to MKS as used in the model
    c_gtunts(&existsUnits, scales, units);

    if(existsUnits ) {
        // Scale transitional measures with length units + speeds with time units
        if(scales[UNIT_SCALE_LENGTH] != 1.0) {
            disp[0] *= scales[UNIT_SCALE_LENGTH];
            disp[1] *= scales[UNIT_SCALE_LENGTH];
            disp[2] *= scales[UNIT_SCALE_LENGTH];

            vel[0] *= scales[UNIT_SCALE_LENGTH];
            vel[1] *= scales[UNIT_SCALE_LENGTH];
            vel[2] *= scales[UNIT_SCALE_LENGTH];
        }
        if(scales[UNIT_SCALE_TIME] != 1.0) {
            for( i = 0; i < 6; ++i) {
               vel[i] /= scales[UNIT_SCALE_TIME];
            }
        }
    }
```

Finally we can call *calc_tlm_force(...)* that invokes both, *SetMotion(...)* and *GetForce(...)* in the TLM interface. In order to handle "final" or "converged" solver steps we invoke the Adams function *c_timget(...)* that returns the last converged time step. This information is passed on to the TLM interface for the correct TLM parameter communication with the connected simulation tools, that is, data is send for converged steps only.

```
    // TIMGET returns simulation time at the end of the last successful step
    c_timget(&lastConvergedTime);

    calc_tlm_force(*iflag,  // init flag. No result is needed.
                   *dflag,  // If set, it is derivatives calculation.
                            // Time stands still.
                   dbgOut,     // should we print debug messages?
                   markers[0], // The calling marker ID
                   *time,      // Current simulation time
                   lastConvergedTime, // Last converged time
                   disp,    // Marker position data
                   rot,     // Marker rotation matrix
                   vel,     // Marker translational velocity
                   vel + 3, // Marker angular velocity
                   result);  // Output 6-component force
```

The calculated load (force and moment) response needs to be converted back to Adams units.

```
    if(existsUnits ) {
        // Scale force & torque
        if(scales[UNIT_SCALE_FORCE] != 1.0) {
            for( i = 0; i < 6; ++i) {
                result[i] /= scales[UNIT_SCALE_FORCE];
            }
            if(scales[UNIT_SCALE_LENGTH] != 1.0) {
                result[3] /= scales[UNIT_SCALE_LENGTH];
                result[4] /= scales[UNIT_SCALE_LENGTH];
                result[5] /= scales[UNIT_SCALE_LENGTH];
            }
        }
    }
};
```

In the ADAMS TLM plugin implementation we keep track of the last converged time step in order to send data only if needed, that is, when we have a converged step. This is implemented in the *TLM_force::GetForce(...)* function that is invoked by *calc_tlm_force(...)*, see File *tlmforce.c* in the TLMPlugin/ADAMS directory.

What happens is that we always store the motion data of the last time step that ADAMS invoked *calc_tlm_force(...)* for. For each step we check if the last step was a converged step (based on the information we got from the ADAMS solver and the data that we stored). If it was a converged step we send the data that we have cached, that is, the last time step.

```
void TLM_force::GetForce(bool derCalc,
                         int markerID,
                         double lastConvergedTime,
                         MarkerMotionData& param,
                         double* force) {
    if(getMode() == 0) {
        // init has been called for all
        setMode();
        SwitchToRunMode();
    }

    MarkerID& mID = MarkerIDmap[markerID];

    int interfaceID = mID.ID; // interface force ID in TLM manager

    MarkerMotionData& lastParam = LastMarkerMotion[mID.index];

    if(!derCalc) { // if it's a normal call (not Jacobian)
        if( (lastParam.Time >= 0 ) // there's data
            && (lastParam.Time != param.Time ) //not a repeated call
            ) {

            if(lastConvergedTime == lastParam.Time) { // that was a converged step
                map<int, MarkerID>::iterator it;
                for(it = MarkerIDmap.begin(); it != MarkerIDmap.end();++it) {
                    int curID = it->second.ID;
                    int index = it->second.index;
                    MarkerMotionData& toSend = LastMarkerMotion[index];
                    Plugin->SetMotion(curID,          // Send data to the Plugin
                                  toSend.Time,
                                  toSend.Position,
                                  toSend.Orientation,
                                  toSend.Speed,
                                  toSend.Ang_speed);
                    // invalidate time to avoid resend
                    toSend.Time = param.Time;
                }
            }
       }

       lastParam = param; // store the current motion data
    }

    // Call the plugin to get reaction force
    Plugin->GetForce(interfaceID,
                     param.Time,
                     param.Position,
                     param.Orientation,
                     param.Speed,
                     param.Ang_speed,
                     force);
}
```

Chapter 6

BEAST TLM Plugin

BEAST integrates the TLM plugin directly into the source code. Any coordinate system in BEAST has an TLM-enable flag to turn this coordinate system into a TLM interface.

```
class CtlPoint : public NamedCoordSystem , public MBSTreeComponent {

...

    //! The TLM activation flag.
    int tlmEnabledFlg;

public:

    // Check for tlmEnabled CtlPoint
    bool isTLMEnaled(){ return tlmEnabledFlg != 0; }

...

};
```

The BEAST TLM co-simulation part is based on two C++ classes:

TLMInterfaceHandler takes care of creating and initializing all additional components for the TLM co-simulation, this is, a global (cB) coordinate system, necessary connection instances, and the necessary TLMTies.

TLMTie A TLMTie is created for each TLM interface, that is, for each TLM enabled coordinate system. It ties the interface to a global coordinate system for correct motion computation. The *TLMTie* functions as the communication port between the actual TLM interface in the co-simulation manager.

6.1 The TLMInterfaceHandler

The *TLMInterfaceHander* stores a list of all TLM-enabled coordinate systems in the Beast model. In the initial phase of the co-simulation it creates all necessary *TLMTies* for the communication with the TLM manager. This includes creation of a global control point that is needed for the *TLMTie* and creation of all necessary *cBBodyConnections*.

```
void TLMInterfaceHandler::EnableTLMCtlPoint(CtlPoint* ctl)
{
    TLMTie* tie=0;
    ModelcBBodyConnection* connection=0;

    // Now, everything seems fine.
    // Let's create the global ctl-point needed for the tie
    if( globalCtl == 0 ){
        assert(topModel!=0);

        doRegister = false;

        globalCtl = new FixedCtlPoint(topModel, "ctTLMglobal" );
        assert(globalCtl!=0);

        doRegister = true;
    }

    // Create the connection if needed
    Connection* tlmConn = 0;
    if( topModel->ConnectionExistQ("cB:"+body->get_FName()) ){
        tlmConn = topModel->GetConnectionPtr("cB:"+body->get_FName());
    }
    else {
        connection = new ModelcBBodyConnection(topModel, body);
        assert(connection!=0);
        tlmConn = connection;
    }

    int idx = tlmConn->Get_ListSize(TIE_category)+1;
    // Now create the TLM tie
    tie = new TLMTie(tlmConn,
                     "TLM" + ToStr(idx),
                     globalCtl,
                     ctl,
     0, 0,
     true);
}
```

6.2 The TLMTie

During the co-simulation the *TLMTie* is responsible for the communication with the TLM co-simulation manager and takes care of force evaluation and motion propagation in the TLM interface. There are three phases. First phase is the preparation of the force evaluation. This is done in *TLMTie::ComputeMasterBefore()*:

```
// Evaluate the data needed for the current time step.
void TLMTie::ComputeMasterBefore()
{
    if (!NonZeroFlg) return;
    assert(ModelMode != SlaveMode);

    // Get the time data for the specified time
    TLMlink->GetTimeData(ForceID, SimTime, CurTimeData);
}
```

TLMlink-¿GetTimeData(...) makes sure that force and moment data for the current time step is available in the interface. If it is not yet available it waits until the data has been received though the TLM manager from the connected simulation tool.

Second phase is to evaluate force and moment and update the internal states of the coordinate system. This happens in *TLMTie::calcChildForceMoment(...)*:

```
void TLMTie::calcChildForceMoment(const MotionVar& ctl2_M_ctl1_ctl1)
{
    if (!NonZeroFlg) return;

    // Get the motion.
    ctl2_M_ctl1_ctl1.Get(ctl2_R_ctl1_ctl1,
 ctl2_A_ctl1,
 ctl2_vR_ctl1_ctl1_ctl1,
 ctl2_Omega_ctl1_ctl1);

    if(RHSFinalFlg) {
ctl2_M_ctl1_ctl1_Final =  ctl2_M_ctl1_ctl1;
    }

    double forceOut[6];

    // Note that the static function is used here.
    // This is necessary since the force might be
    // evaluated several times and on a slave.
    TLMPlugin::GetForce(&ctl2_R_ctl1_ctl1(1),
&ctl2_A_ctl1(1,1),
&ctl2_vR_ctl1_ctl1_ctl1(1),
&ctl2_Omega_ctl1_ctl1(1),
CurTimeData,
Params,
forceOut);

    // pc is equal ctl2 for now, i.e. ConLoc12 =1.0
    pc_ctl1_ctl1 = ctl2_R_ctl1_ctl1;;

    double3 F_pc_ctl1_tmp(forceOut[0], forceOut[1],forceOut[2]);
    double3 M_pc_ctl1_tmp(forceOut[3], forceOut[4],forceOut[5]);

    // Transform to system ctl1.
    M2_pc_ctl1 = M_pc_ctl1_tmp;
    M2_ctl1_ctl1 = M_pc_ctl1_tmp + Cross(pc_ctl1_ctl1, F_pc_ctl1_tmp);
    F2_ctl1 = F_pc_ctl1_tmp;


    ctl2_P_ctl1 = F2_ctl1*ctl2_vR_ctl1_ctl1_ctl1 + M2_ctl1_ctl1*ctl2_Omega_ctl1_ctl1;
}
```

Note, that in a parallel BEAST simulation *TLMTie::calcChildForceMoment(...)* is invoked on the slaves. Slaves do not have a TLMPlugin instance but use the static *TLMPlugin::GetForce(...)* instead that requires the TLM parameters as input. Time-data and TLM parameters are therefore send to all the slaves using the standard packing mechanism.

The third phase is to send the necessary response to the TLM interface. This is done for final/converged solver steps only. In BEAST we can check this the *RHSFinal* flag:

```
void TLMTie::ComputeMasterAfter()
    // Set the data - send out the force used in this Evaluate the data needed for the current time step.
{
    if (!NonZeroFlg) return;

    if(RHSFinalFlg) {
// Get the motion.
ctl2_M_ctl1_ctl1_Final.Get(ctl2_R_ctl1_ctl1,
   ctl2_A_ctl1,
   ctl2_vR_ctl1_ctl1_ctl1,
   ctl2_Omega_ctl1_ctl1);

// Set it in TLM Plugin, socket communication might happen
TLMlink->SetMotion(ForceID,
   SimTime,
   &ctl2_R_ctl1_ctl1(1),
   &ctl2_A_ctl1(1,1),
   &ctl2_vR_ctl1_ctl1_ctl1(1),
   &ctl2_Omega_ctl1_ctl1(1));
    }
}
```

Chapter 7

Modelica TLM Plugin

A Modelica TLM library has been implemented for the purpose of co-simulation. The library consists of TLM functions, sensors and the TLM interface for 1D and 3D modeling. In this report we will illustrate how to use the TLM interface together with the Modelica Multi-Body Library. The Library has been tested and verified in *Wolfram SystemModeler*, *Dymola*, and *OpenModelica*.

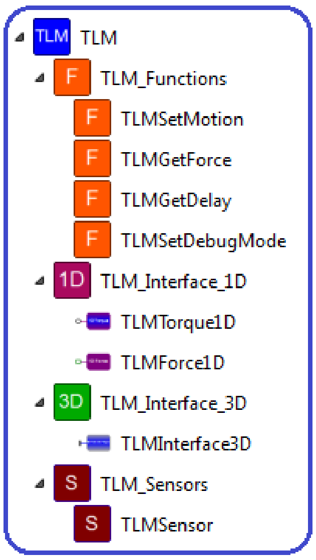
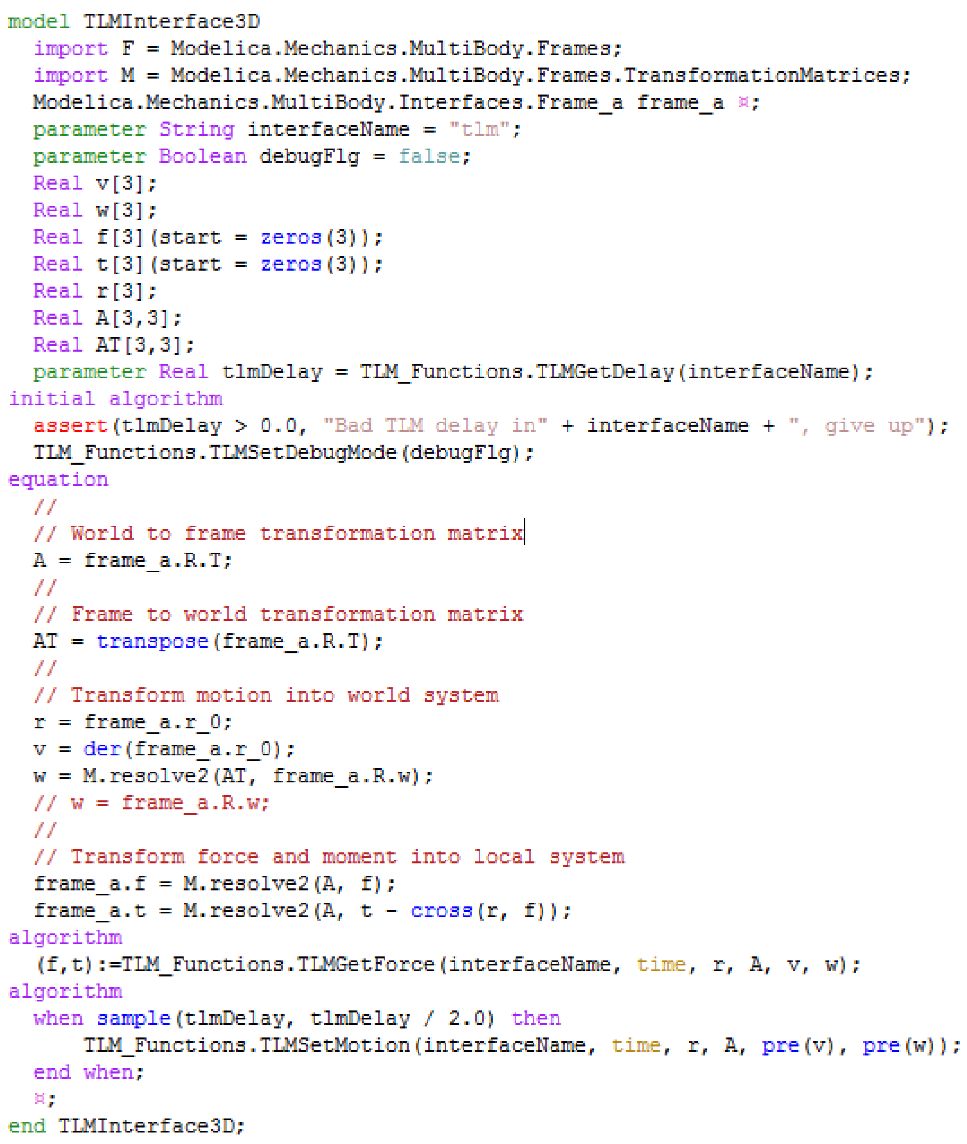


Figure 7.1: Modelica TLM Library

In this chapter we will describe the implementation of the TLM interface for Modelica. The component of interest is the 3D TLM interface from the TLM Modelica library. The 3D TLM component is the interface for co-simulation between Modelica multi-body models and external models. The design of the 3D TLM interface is given in Figure: (7.2).

The main functions in the 3D multi-body TLM component are the C-functions *TLMSetMotion()* and *TLMGetMotion()*. These two functions receive and send the model-name, time, position, orientation, velocity and the angular-velocity



```
model TLMInterface3D
  import F = Modelica.Mechanics.MultiBody.Frames;
  import M = Modelica.Mechanics.MultiBody.Frames.TransformationMatrices;
  Modelica.Mechanics.MultiBody.Interfaces.Frame_a frame_a ¤;
  parameter String interfaceName = "tlm";
  parameter Boolean debugFlg = false;
  Real v[3];
  Real w[3];
  Real f[3](start = zeros(3));
  Real t[3](start = zeros(3));
  Real r[3];
  Real A[3,3];
  Real AT[3,3];
  parameter Real tlmDelay = TLM_Functions.TLMGetDelay(interfaceName);
initial algorithm
  assert(tlmDelay > 0.0, "Bad TLM delay in" + interfaceName + ", give up");
  TLM_Functions.TLMSetDebugMode(debugFlg);
equation
  //
  // World to frame transformation matrix
  A = frame_a.R.T;
  //
  // Frame to world transformation matrix
  AT = transpose(frame_a.R.T);
  //
  // Transform motion into world system
  r = frame_a.r_0;
  v = der(frame_a.r_0);
  w = M.resolve2(AT, frame_a.R.w);
  // w = frame_a.R.w;
  //
  // Transform force and moment into local system
  frame_a.f = M.resolve2(A, f);
  frame_a.t = M.resolve2(A, t - cross(r, f));
algorithm
  (f,t):=TLM_Functions.TLMGetForce(interfaceName, time, r, A, v, w);
algorithm
  when sample(tlmDelay, tlmDelay / 2.0) then
      TLM_Functions.TLMSetMotion(interfaceName, time, r, A, pre(v), pre(w));
  end when;
  ¤;
end TLMInterface3D;
```

Figure 7.2: 3D TLM Modelica component text view

between a Modelica model and an external model. The *TLMSetMotion()* sends the above mentioned variables between each sub-model within a user-defined TLM Time Delay. The TLM Time Delay is defined in an XML file, which describes the relationship between each Modelica sub-model.

The cut-force and cut-torque acting on the mechanical component the TLM component is connected to is calculated based on the data received from the *TLMGet-Force()* function. The force and torque calculation takes place in the C-functions.

On Figure: (7.3) a flow chart of the Modelica TLM interface co-simulation is given. The first step in the TLM Modelica co-simulation is to split a model into two or more sub-models (in cases where the co-simulation is undertaken within two Modelica tools). A TLM interface component has to be connected to each sub-model.

The second step is to prepare the files for the simulation. The first step in the co-simulation process is to write an XML file. In the XML file the user defines the TLM tie relationship between each sub-model that has to be included in the co-simulation in Modelica. See Figure: (11.5) for an example of an XML file. The XML file contains the model connection description, TLM Time Delay, physical connection parameters, and the simulation time.

When the XML file has been created a simulation using the TLM Manager can now be undertaken. When the user executes the TLM manager the TLM manager executes the BAT file, which contains the information about the simulation, i.e. simulation solver, variables to be saved in the result file, step time, numbers of interval etc., and generates a MOS file needed for the Modelica Engine to start a simulation. Based on the MOS file (which is generated for each sub-model) a simulation is started. When the simulation is finished a result file is written, currently a MAT file (the user can chose the type of result file from the simulation in the BAT file). The result file can now be loaded into a Modelica GUI and an analysis of the results can be undertaken.

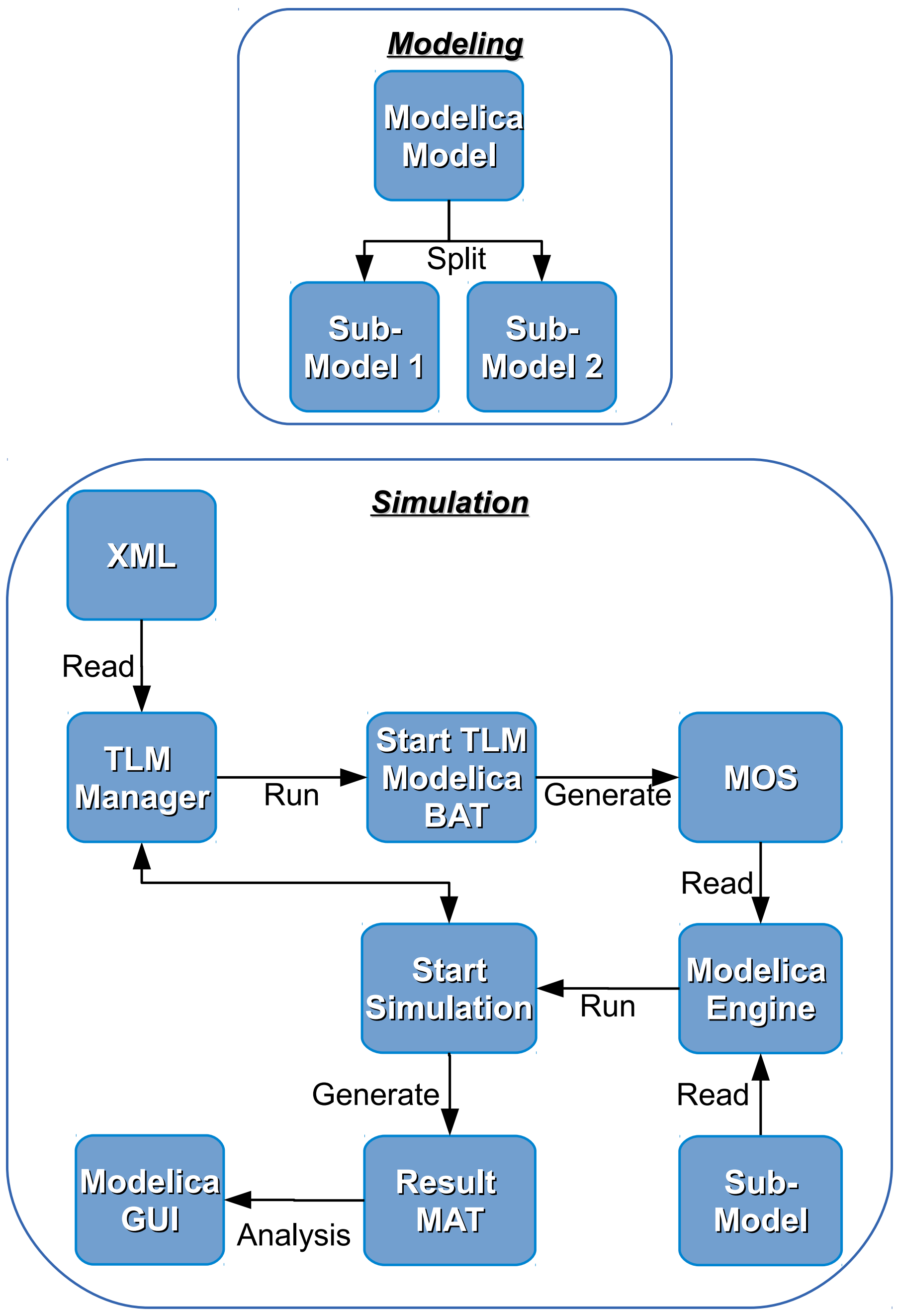


Figure 7.3: Modelica TLM co-simulation flow chart

Chapter 8

External MSC/Adams Models

8.1 Preparing the Model File

The TLM Plugin interface was implemented for ADAMS 2005. It has been tested with `C++` solvers only. It is installed with the BEAST installation and located in the BEAST bin directory.

ADAMS simulation is performed in batch mode, therefore the solver data set (adm file) needs to be generated. Before you can export the file from ADAMS/View you need to introduce TLM Interfaces in your model. TLM interfaces in ADAMS models are represented by GFORCE elements acting between a marker on the action part and a marker on ground. Note that different coordinate systems are often used in Beast and ADAMS. Typically, the gravity points in the X-negative direction for BEAST models and Y-negative for ADAMS.

8.2 Modifying an ADAMS model step-by-step

The example is an ADAMS-BEAST co-simulation where one of the revolute joints in an existing ADAMS model will be substituted with a BEAST bearing.

Start by locating a revolute joint to be replaced by the real bearing model. Print out the information about the joint and about the markers it connects. A typical information is presented in Figure 8.1. Write down the names of the markers and their IDs. The markers will become TLM interfaces in the resulting component. For the presented examples the names would be `M4` for the shaft interface and `M10` for the housing interface.

The revolute joint should be excluded from the model, e.g., by using the object activate/deactivate dialog as presented in Figure 8.2.

Next step is creating a reference marker on ground that will give the correct orientation of the bearing. Use the Command Navigator - marker - create and specify the orientation so that the Z-axis is the axis of rotation and gravity has negative X direction. An example is presented in Figure 8.3. Write down the ID of the newly created marker.

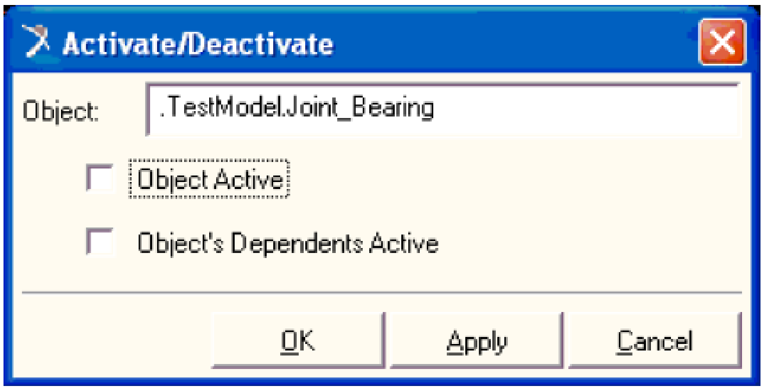
Next step is the creation of general forces representing the TLM connections. Use the Command Navigator-force-create-direct-general_force. The Figures 8.4 and 8.5 presents an example of the dialog.

```
 Object Name     :  .TestModel.Joint_Bearing
Object Type     :  Revolute Joint
Parent Type     :  Model
Adams ID        :  1
Active          :  NO_OPINION
I Marker          : .TestModel.Shaft.Marker_Joint_Shaft
J Marker          : .TestModel.ground.Marker_Joint_Housing
Initial Conditions
  Angular Displacement : NOT SET
  Angular Velocity     : NOT SET


Object Name     :  .TestModel.Shaft.Marker_Joint_Shaft
Object Type     :  Marker
Parent Type     :  Part
Adams ID        :  4
Active          :  NO_OPINION
Local  :
    Location    : -150.0, 150.0, 0.0 (mm, mm, mm)
    Orientation : 0.0, 0.0, 0.0 (deg)
Global :
    Location    : -150.0, 150.0, 0.0 (mm, mm, mm)
    Orientation : 0.0, 0.0, 0.0 (deg)

Object Name     :  .TestModel.ground.Marker_Joint_Housing
Object Type     :  Marker
Parent Type     :  Part
Adams ID        :  10
Active          :  NO_OPINION
Local  :
    Location    : -150.0, 150.0, 0.0 (mm, mm, mm)
    Orientation : 0.0, 0.0, 0.0 (deg)
Global :
    Location    : -150.0, 150.0, 0.0 (mm, mm, mm)
    Orientation : 0.0, 0.0, 0.0 (deg)
```

Figure 8.1: Information print-out for a Revolute Joint and its Markers

The ADAMS solver dataset can now be exported via the export dialog as shown in Figure 8.6. Note that "Verify the model" check box is off. Verification will result in an unsuccessful call to TLMManager that crashes the ADAMS/View.

TLM co-simulation is sensitive to the consistence of the velocity initial conditions. The simulation will start and run even if the initial velocities are not consistent. However, high amplitude vibrarions in the TLM element with the frequency of ω_{TLM} as defined in Equation 2.10. Section 2 will show up in the system. Therefore it is recommended to start the simulation with zero velocities in all the components and simulate a smooth transition to the working velocities.

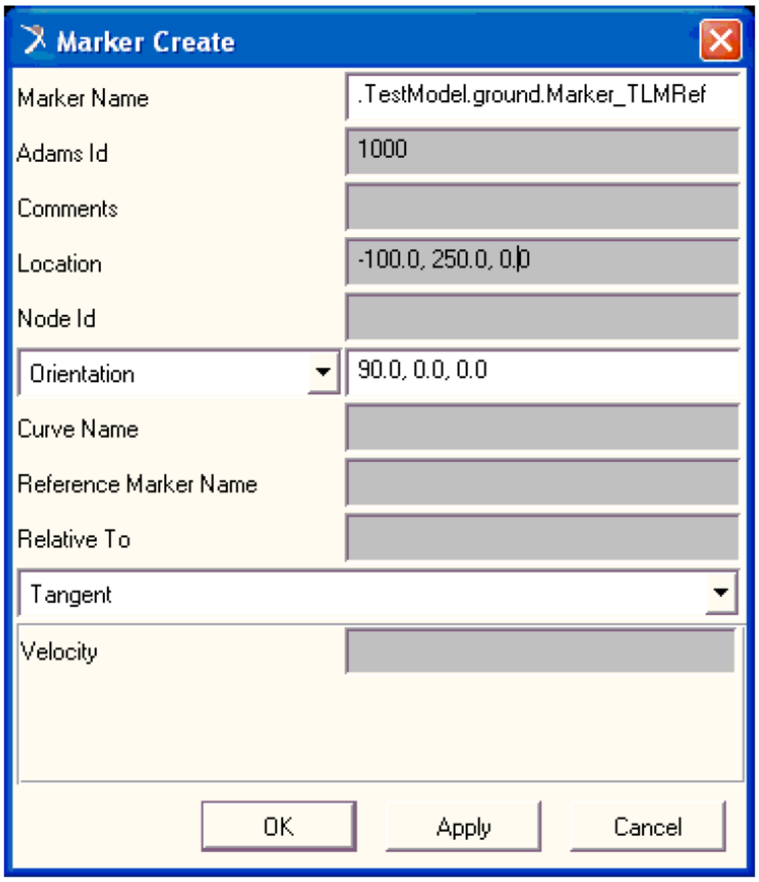


Object: .TestModel.Joint_Bearing

☐ Object Active

☐ Object's Dependents Active

Figure 8.2: Joint deactivation dialog



Marker Name: .TestModel.ground.Marker_TLMRef

Adams Id: 1000

Comments:

Location: -100.0, 250.0, 0.0

Node Id:

Orientation: 90.0, 0.0, 0.0

Curve Name:

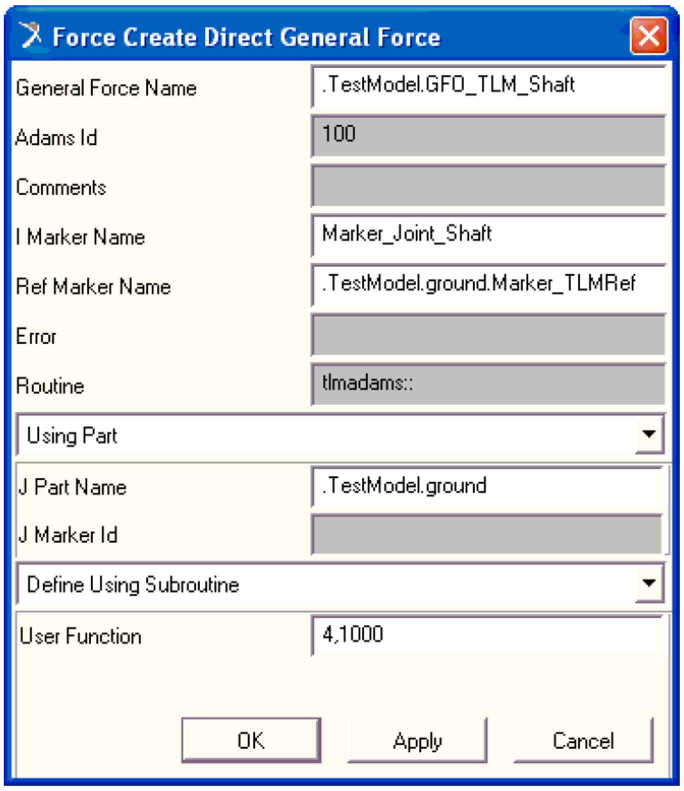
Reference Marker Name:

Relative To:

Tangent

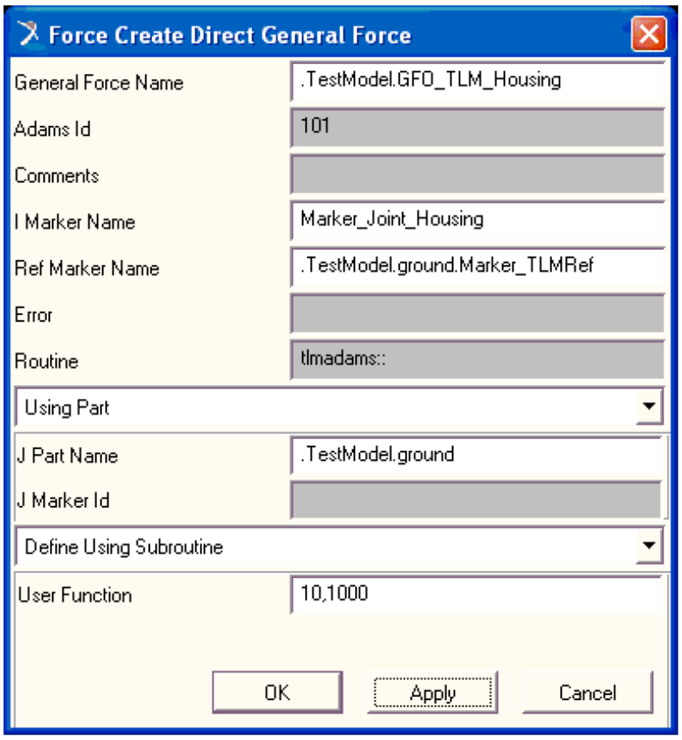
Velocity:

Figure 8.3: Create TLM reference marker dialog


Force Create Direct General Force

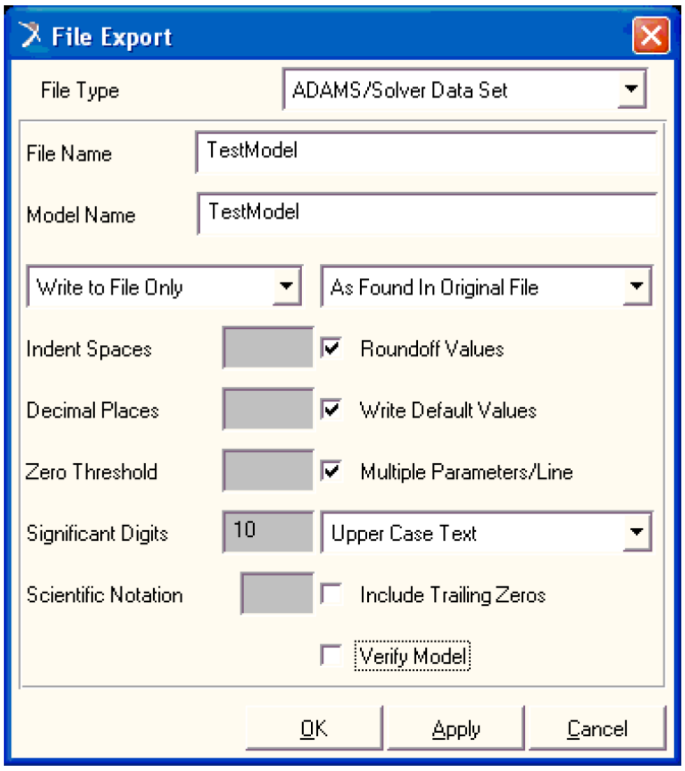
General Force Name	.TestModel.GFO_TLM_Shaft
Adams Id	100
Comments	
I Marker Name	Marker_Joint_Shaft
Ref Marker Name	.TestModel.ground.Marker_TLMRef
Error	
Routine	tlmadams::
Using Part	
J Part Name	.TestModel.ground
J Marker Id	
Define Using Subroutine	
User Function	4,1000

Figure 8.4: Create general force on shaft for TLM dialog



General Force Name	.TestModel.GFO_TLM_Housing
Adams Id	101
Comments	
I Marker Name	Marker_Joint_Housing
Ref Marker Name	.TestModel.ground.Marker_TLMRef
Error	
Routine	tlmadams::
Using Part	
J Part Name	.TestModel.ground
J Marker Id	
Define Using Subroutine	
User Function	10,1000

Figure 8.5: Create general force on housing for TLM dialog



File Type	ADAMS/Solver Data Set
File Name	TestModel
Model Name	TestModel
Write to File Only	As Found In Original File
Indent Spaces	✓ Roundoff Values
Decimal Places	✓ Write Default Values
Zero Threshold	✓ Multiple Parameters/Line
Significant Digits 10	Upper Case Text
Scientific Notation	☐ Include Trailing Zeros
	☐ Verify Model

Figure 8.6: Create general force on housing for TLM dialog

8.3 Preparing the Scripts

The next step is creation of an appropriate Adams Command File (.acf). The file should start a transient simulation of your model.

Important note:
The current version of TLMSimulator does not propagate time-settings from the composite model to the MSC.ADAMS models automatically. Therfore users need to check that ”time” and ”endTime” correspond to the times in the composite model. Check that the integrator parameter Hmax is set to be less than TLM Delay/2 (use INTEGRATOR command).

An example ACF file is given in Figure 8.7.

```
Pendulum
Pendulum_out
INTEGRATOR/SI2, GSTIFF, ERROR=1.0E-6,
HMAX=0.25e-5, HINIT=1.0E-6, LIST
sim/dyn, end=1e-3, steps=10
stop
```

Figure 8.7: An example ACF (Adams Command File)

A startup script for running the ADAMS simulation needs to be created as well. A default script is distributed with TLMSimulator. It is called StartTLMAdams.bat.

Note: The default `StartTLMAdams.bat` Windows script needs to be checked. This should probably be done together with a system administrator and a member of the BEAST team. However below is a short discussion of the script for advanced users.

Figure 8.8 presents a template for the start script. The `StartTLMAdams.bat` should first generate a file `tlm.config` that will contain the parameter send to it by TLM manager. Only the line giving start command for ADAMS needs to be changed. For instance, for ADAMS car the line should read:

```
set ADAMSSCRIPT=mdi acar ru-so
```

Finally, make the `tlmadams.dll` library is accessible for the ADAMS solver. The simplest way is to copy the library file to your working directory.

```
@echo off
rem If other parameters are used for ADAMS, change the next line

set ADAMSSCRIPT=mdi ru-s

rem ***************************************************************
rem Changes below should not be necessary

if not a%1==a goto startadams
echo This script is used by TLM manager to start ADAMS simulations
echo It should result in a call to the mdi script
echo for running a simulation according to the specified ACF file
exit 1

:startadams

echo Simulation directory is %1
cd %1

echo Starting an ADAMS simulation with input file: %6.acf
echo Make sure that:
echo time = %2
echo timeEnd = %3
echo MaxTimeStep "<"= %4
echo Writing server configuration to file tlm.config

echo %1 > tlm.config
echo %2 >> tlm.config
echo %3 >> tlm.config
echo %4 >> tlm.config
echo %5 >> tlm.config

echo Starting ADAMS

%ADAMSSCRIPT% %6 > %6.simlog
```

Figure 8.8: A template for the *startadams.bat* script

Chapter 9

External Simulink Models

The TLM plugin has been implemented for Simulink. This section describes how to use and install the plugin to be able to integrate Simulink models into composite models and co-simulations.

9.1 TLM Simulink Library

A Simulink library exists that contains different Simulink blocks, see also Figure 9.1. The main block is the `TLMForce` block that is a Simulink S-Function implementation of the TLM plugin. The `TLMDriver` and `TLMInterface` are sub-systems that are used to connect SimMechanics components to the `TLMForce`. Additionally, some useful Simulink blocks are collected in the `TOOLS` sub-system, for instance, blocks for matrix transformations.

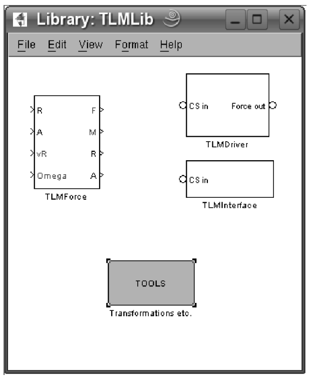


Figure 9.1: The Simulink TLM library for coupled simulations.

The (`TLMForce`) block takes four input variables: position (`R`), orientation matrix (`A`), velocity (`vR`), and rotational velocity (`Omega`). TLM calculates output force (`F`) and moment (`M`) from velocity and rotational velocity. From a TLM point of view, the most important variables are `vR` and `Omega`.

Position and orientation are mainly used for graphics in the composite model editor (CME) and for possible coordinate transformations. All input data must be expressed relative the inertial coordinate system of the simulation model. All output data is expressed relative the inertial coordinate system of the simulation model.

Position, velocities, force, and moment, are vectors of dimension three: [X Y Z]. Whereas the orientation matrix is a vectors of dimension nine defining the 3x3 matrix A(row, column): [A(1,1) A(1,2) A(1,3) A(2,1) A(2,2) A(2,3) A(3,1) A(3,2) A(3,3)]

9.2 Test Case: Pendulum

A pendulum with three TLM connected shafts has been implemented as a test model, see Figure 9.2. Each shaft is modeled as a separate Simulink model using SimMechanics.

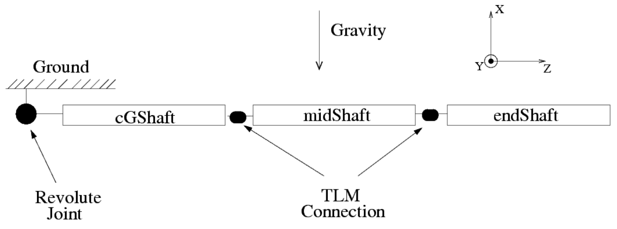


Figure 9.2: Three shaft pendulum with TLM connections between the shafts and a revolute joint connected to the ground.

The Simulink model of the three shaft pendulum has been implemented with SimMechanics and the `TLMInterface` from the TLMLib, see also Section 9.1. cGShaft is connected to the ground using a `Revolute` joint from the SimMechanics library, see Figure 9.3(a). The other shafts (midShaft and endShaft) are based on a Simulink model with a free body using a `Six-DoF` joint, see Figure 9.3(b). All three shafts are connected in their TLM interfaces.

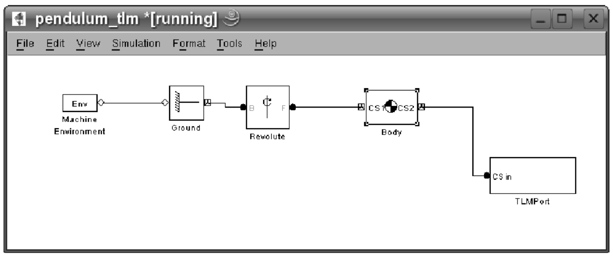
Each of the shafts has a mass of `3Kg`, a length of `160mm`, and a diameter of `40mm`. Assuming that the characteristic length L_0 is equal to the diameter of the shaft we can calculate the TLM parameters for the model according to Section 2, as follows:

$$
\begin{aligned}
&T_{TLM} = 0.04/5180 \approx 8 * 10^{-6} \\
&Z_F = 210 * 10^9 * 0.04 * 8 * 10^{-6} \approx 68000 \\
&Z_{FR} = \tfrac{1}{4} * 68000 * 0.04^2 \approx 30
\end{aligned}
\tag{9.1}
$$

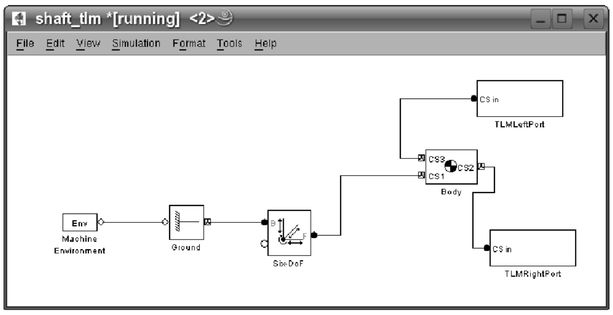
The composite model describes how the shafts are connected and what TLM parameters should be used in the different TLM connections. It is created using the composite model Editor.

The pendulum model is simple and has all important characteristics to test the TLM plugin:

- Multiple simulation models with single and multiple TLM interfaces and connections.



(a) SimMechanics shaft connected to ground with revolute joint.

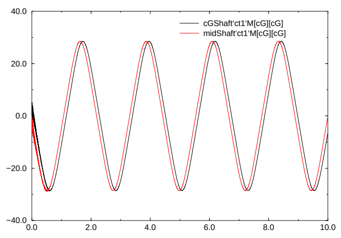


(b) Free SimMechanics shaft model.

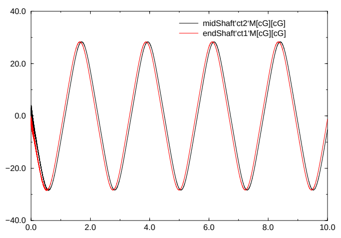
Figure 9.3: Simulink SimMechanics models for the three shaft pendulum.

- Open (not connected) TLM interface in the endShaft.
- Translational displacement of the midShaft and endShaft.
- Rotational and translational motion.

Results from the co-simulation can be verified by plotting the movement of the TLM interfaces.



(a) Moment in TLM connected coordinate systems cGShaft-midShaft.



(b) Moment in TLM connected coordinate systems midShaft-endShaft.

Figure 9.4: The difference in Moment between the TLM connected coordinate systems due to TLM delay.

Figure 9.4, for instance, shows the moments created in the TLM interfaces of the connected Simulink models. One can nicely see the time delay in the connected interfaces due to TLM delay.

9.3 Installation

Generally the following steps need to be conducted for a correct installation of the plugin:

1. Install the plugin files on your system.

2. Add the installation directory or directories to the Matlab `Search Path`.

3. Possibly adjust the StartTLMSimulink start-up script to your needs.

The Simulink TLM-plugin consists of three files:

- A tlmforce S-Function, or tlmforce.mex* file.

- A TLM library file, TLMLib.mdl.

- A start-up script, StartTLMSimulink

The start script is located in the `$BEAST/bin` directory. The tlmforce S-Function mex and the TLMLib.mdl files are placed in the `$BEAST/bin/$ABI/Simulink` directory. There is a Linux and a MS-Windows version of the plugin available.

It is important that the TLMLib.mdl and tlmforce.mex* file are within the Matlab `Search Path`. This can be achieved with the `Select Path` dialog that can be opened from the `File` menu in the Matlab main window.

Changes to the StartTLMSimulink script are typically not necessary. Note that there are two versions of the script, a MS Windows batch script and a c-shells script for Linux/Unix systems. In case the system requires a special setup for running Matlab/Simulink it is recommended to make changes to the script together with member of the BEAST team. Here, however, the c-shell version of the script:

```
#!/bin/csh -f

# Set the parameters to the beast script in the line below:

setenv PATH ${PATH}:/home/alex/bin/matlab2006b/bin
set SIMCOMMAND="matlab -nosplash -nodesktop -nojvm -r"

##############################################################
#------ Changes below this line should not be needed     ----

if("a$1" == "a") then
echo This script is used by TLM manager to start simulations
echo It should result in a call to matlab/simulink
exit 1
endif

cd $1

echo Starting a simulation with input file: $6
echo Make sure that:
echo time = $2
echo timeEnd = $3
echo MaxTimeStep "<"= $4
echo Writing component name $1 and server name $5 to file tlm.config
echo $1 $5 > tlm.config
echo $2 $3 $4 >> tlm.config

echo Starting matlab/simulink
echo $SIMCOMMAND

echo Writing execution methods to runtlm.m
echo "$6;" > runtlm.m
if( "$4" != "0" ) then
 echo "opts = simset('MaxStep', $4);" >> runtlm.m
 echo "sim( '$6', [$2 $3], opts );" >> runtlm.m
else
 echo "sim( '$6', [$2 $3] );" >> runtlm.m
endif
echo "save; quit force;" >> runtlm.m

exec $SIMCOMMAND runtlm > ${6}.simlog
```

9.4 Creating External Simulink Models

To enable an existing Simulink model for co-simulation is fairly simple:

1. Open the model and the TLM library TLMLib.mdl, see also Figure 9.1.
2. Drag the TLMForce (pure Simulink) or TLMInterface (SimMechanics) block into the model.
3. Create the necessary connections to the TLM block, see also Section 9.1 for details about signal dimensions.
4. Send all signals of interest to the Matlab workspace or to a file.

Remember, that the TLM plugin requires velocities to calculate output force and moment. Remember also, that all data must be expressed relative the global inertial system. Necessary coordinate transformations should be implemented in the Simulink model.

All external models are automatically executed on co-simulation start-up and terminated when the simulation is finished. Data that is not written to a file is lost when the simulation terminates. The Simulink plugin writes Matlab workspace content to the file `matlab.mat` when the simulation is finished. All simulation data that should be analyzed should either be written to a file during the simulation or stored in the workspace. The latter can be achieved by using a Simulink `ToWorkspace` sink, by logging signals, or by logging Simulink `Scope` data.

The next step is to create an external model from the TLM prepared Simulink model. It is created using the composite model editor. The composite model describes how different external models are connected and what TLM parameters should be used in the different TLM connections. Creating external Simulink models is not different from creating any other external model. Only the correct start-up method, i.e., StartTLMSimulink needs to be selected.

NOTE: Neither Simulink nor SimMechanics can export surface graphics for visualization of the composite model. Graphics files, i.e., VRML or STL files, can be created from a CAD tool or with the Virtual-Reality authoring tool of the Matlab Virtual Reality toolbox.

Results from the co-simulation can be verified in CME by plotting the movement of the TLM interfaces.

Chapter 10

External BEAST Models

The TLM-Plugin has been implemented for BEAST. This section describes how to integrate BEAST models into composite models and co-simulations.

10.1 TLM Enabled BEAST models

A BEAST model needs to be "TLM enabled" to participate in a co-simulation. The TLM-plugin has been integrated into the BEAST code and is thus always available in BEAST models.

Control-points are the interfaces for TLM connections in BEAST. One can enable any fixed and flexible control-point in BEAST for TLM connections. For TLM connections it is, however, recommended to use control-points that are not connected to a tie.

Any BEAST model that has at least one TLM enabled control-point can participate in a co-simulation. To enable a control-point for TLM communication one needs to set the *TLMEnabledFlg* for this control point in the following way:

- Right-click on the control point in the model-browser
- From the pop-up menu select *Edit Variables...* and then *TLM*
- In the dialog set the *TLMEnabledFlg* to *Yes*.

The BEAST model can then be saved and integrated into as an external model into a composite model.

10.2 BEAST Startup Script

A startup script for running the BEAST simulation needs to be created as well. A default script is distributed with the BEAST/CME installation. It is called StartTLMBeast.bat.

Note: The default `StartTLMBeast.bat` Windows script needs to be checked. This should probably be done together with a system administrator and a member of the BEAST team. However below is a short discussion of the script for advanced users.

Figure 10.1 presents a template for the start script. The `StartTLMBeast.bat` should first generate a file `<CaseName>.tlm` that will contain the parameter send to it by TLM manager. Only the line giving start command for BEAST needs to be changed.

```
@echo off
set BeastCmd="%BEAST%/bin/%ABI%/FORMAT-9.3/Beast_Serial"

echo execution directory is %1
cd %1

echo Starting a Beast simulation with input file: %6
echo Make sure that:
echo time = %2
echo timeEnd = %3
echo MaxTimeStep "<"= %4
echo Writing caseID %1 and server name %5 to file %6.tlm
echo %1 > %6.tlm
echo %5 >> %6.tlm
echo %2 >> %6.tlm
echo %3 >> %6.tlm
echo %4 >> %6.tlm

echo Starting beast
echo %BeastCmd% %6.in

%BeastCmd% %6.in > %6.simlog
```

Figure 10.1: A template for the *startadams.bat* script

Chapter 11

External Modelica Models

In this chapter a description of how to design and co-simulate Modelica models with the TLM interface is given.

11.0.1 Build and link TLM Manager to a Modelica tool

The first step is to install the TLM manager. But please make sure you have Microsoft Visual Studio installed and the MinGW compiler for OMC and WSM (the MinGW compiler comes with OMC and WSM by default).

Depending on which Modelica tool you are using, do as follows:

In OpenModelica

Run the script *InstallTlmForOmc.bat* in MS DOS.

In Wolfram SystemModeler

Run the script *InstallTlmForWsm.bat* in MS DOS.

In Dymola

Run the script *InstallTlmForDymola.bat* in MS DOS.

The above mentioned script files are located in *\TLMPlugin\Modelica.*

Note: The default Windows TLM installation BAT script may need some modifications. The main issue is to modify the script file so the Modelica tool, Microsoft Visual Studio and the TLMPlugin folder paths are correct. Furthermore, it is important to add the TLMPlugin/bin folder path to the Windows System Environment Path.

The installation script will set all the necessary paths in order to use the TLM interface. Thereafter, a library is generated and copied to the Modelica tool installation folder (for Dymola) or to the resource folder located in the TLMSimulator folder for OMC and WSM). For Dymola a tlmforce.h file is copied to the installation folder and for OMC and WSM the tlmforce.h file is copied to the resource folder. Next, the TLM Manager is compiled and built.

In order to run the installation script simple start a commando prompt with administrator rights and execute the script.

Before running the TLM Manager please check the log after executing the installation script for any errors. If the installation was a success restart the computer in order to update the System Environment paths.

11.1 How to create a model with a TLM interface

In this example we will split a model of a double pendulum and run a simulation with the TLM interface. The first step is to split the full model of the double pendulum and create two models, which are saved in two separate folders.

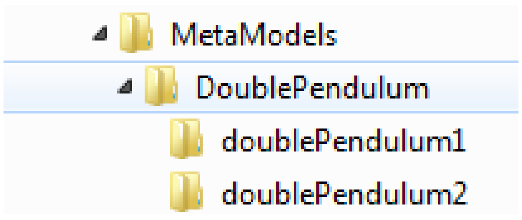


Figure 11.1: figuretext

In this example we will call the two new models 'DoublePendulum1' and 'DoublePendulum2'. At the end of each model (where the two models need to be connected) a TLM component is connected from the TLM Modelica library.

Full model

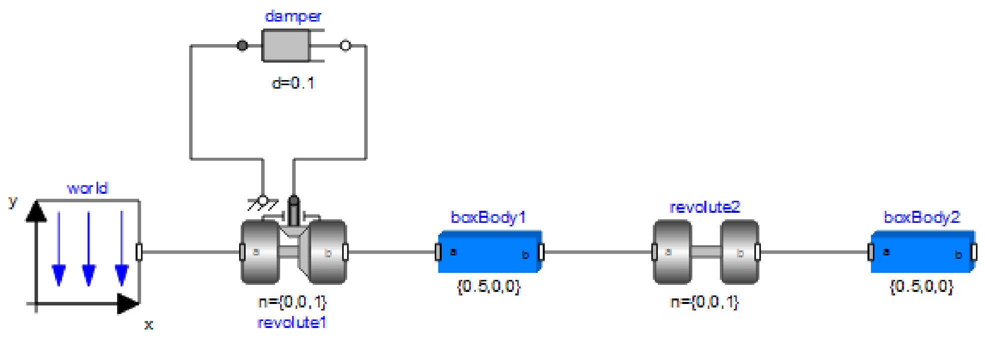


Figure 11.2: figuretext

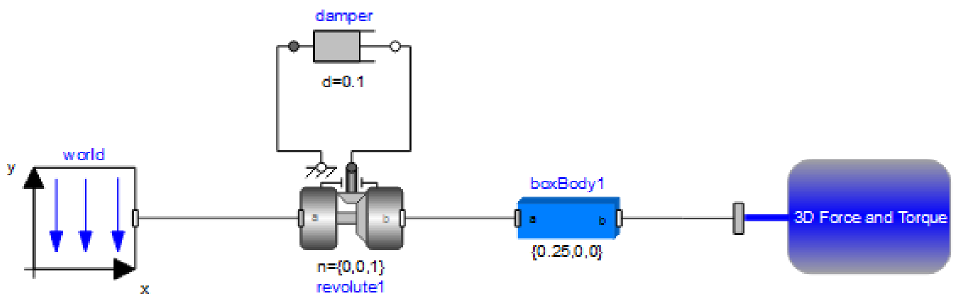


Figure 11.3: Sub-model 1 of a double Pendulum

Sub-model 1 (DoublePendulum1)

Sub-model 2 (DoublePendulum2)

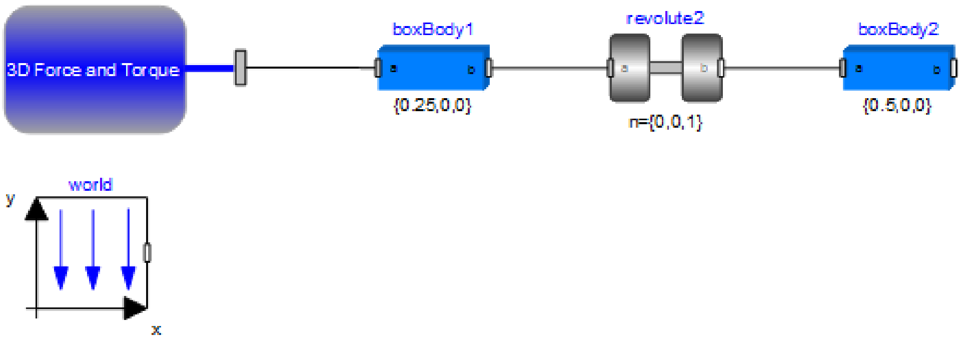


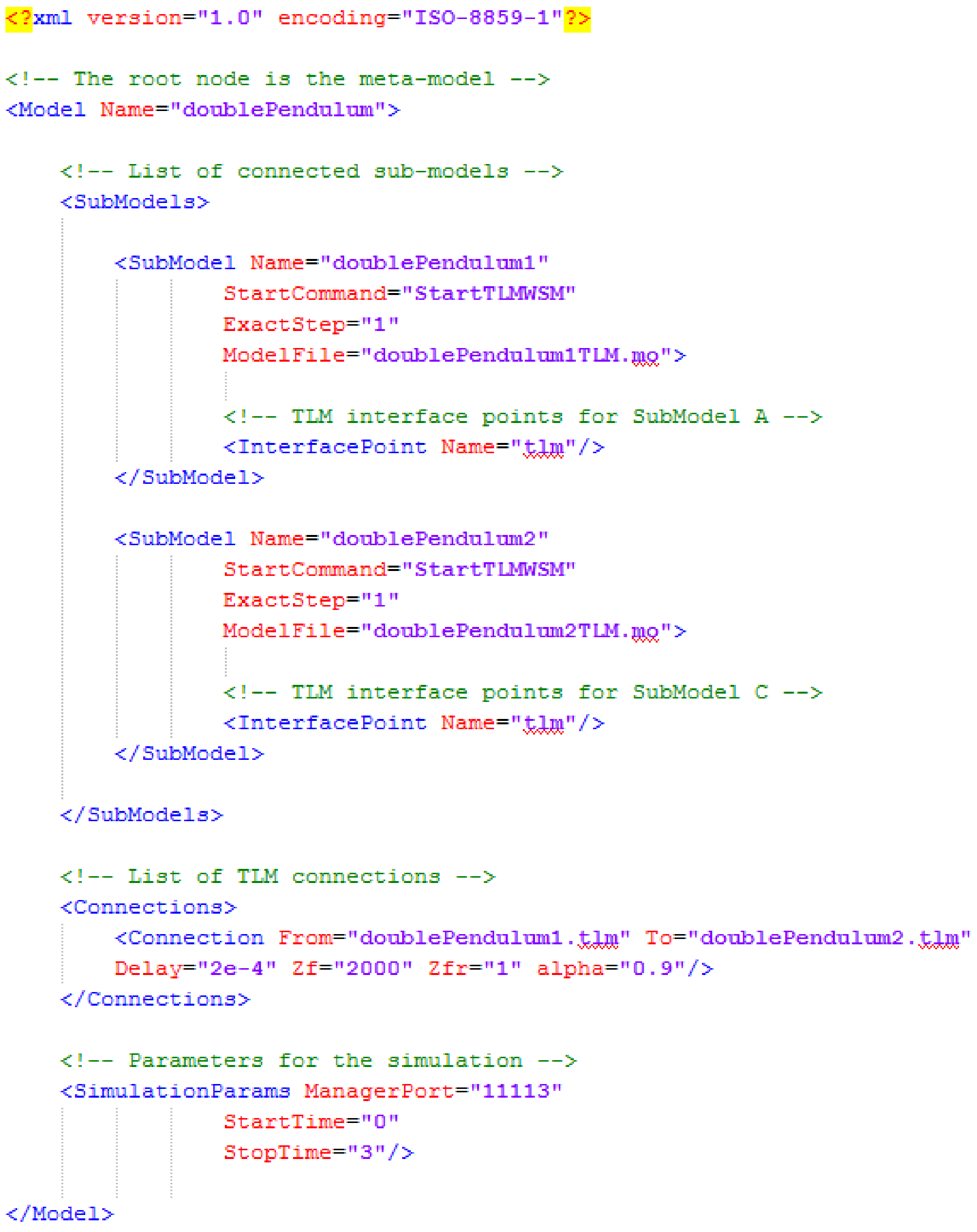
Figure 11.4: Sub-model 2 of a double Pendulum

When using the TLM interface in Modelica it is important to note that the TLM connection has to be connected to a body! Therefore, if you compare DoublePendulum2 with the full model, you can see that an extra body has been added. The extra body (boxBody1) is half of the boxbody1 from the full model (half mass and length). Furthermore, the boxBody1 from DoublePendulum1 is half (mass and length) of the boxbody1 from the full model. It is important that the initial conditions for the objects are correct, i.e. the boxbody1 from DoublePendulum2 will have an initial position (r_0[3] = 0.25,0,0) for the given example.

11.1.1 Prepare the XML composite model file for simulation

In the XML file we have to define how the models are connected and which Modelica tool is used. Furthermore the simulation time is defined. The XML file for the double pendulum:

Save the file to the top folder (\CompositeModels \DoublePendulum).



```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- The root node is the meta-model -->
<Model Name="doublePendulum">

    <!-- List of connected sub-models -->
    <SubModels>

        <SubModel Name="doublePendulum1"
                StartCommand="StartTLMWSM"
                ExactStep="1"
                ModelFile="doublePendulum1TLM.mo">

                <!-- TLM interface points for SubModel A -->
                <InterfacePoint Name="tlm"/>
        </SubModel>

        <SubModel Name="doublePendulum2"
                StartCommand="StartTLMWSM"
                ExactStep="1"
                ModelFile="doublePendulum2TLM.mo">

                <!-- TLM interface points for SubModel C -->
                <InterfacePoint Name="tlm"/>
        </SubModel>

    </SubModels>

    <!-- List of TLM connections -->
    <Connections>
        <Connection From="doublePendulum1.tlm" To="doublePendulum2.tlm"
        Delay="2e-4" Zf="2000" Zfr="1" alpha="0.9"/>
    </Connections>

    <!-- Parameters for the simulation -->
    <SimulationParams ManagerPort="11113"
                StartTime="0"
                StopTime="3"/>

</Model>
```

Figure 11.5: XML file for Modelica co-simulation

11.1.2 Set up simulation settings

The next step is to set up the simulation settings. For OpenModelica and Dymola all the simulation settings have to be modified in a BAT script. The BAT script is located in the bin folder: (*\TLMPlugin\bin\*).

- For openModelica the file is called: *StartTLMModelica.bat*.
- For Wolfram SystemModeler the file is called: *StartTLMWSM.bat*.
- For Dymola the file is called: *StartTLMDymola.bat*.

The first step is to modify the Modelica tool installation path so it matches the user's installation folder on his computer. Following, the simulation function in the BAT script has to been updated. The function defines the simulations settings, i.e. tolerance, number of intervals, solvers etc. When using Wolfram SystemModeler the user doesn't need to modify the simulation settings. Using the Simulation Center, the user can follow the simulation 'live'.

11.2 How to start a TLM simulation.

To start a TLM simulation, follow these 5 steps (please note the paths below are relative and needs to be fully specified according to the TLMPlugin installation folder).

- Start a commando prompt.
- Go to your CompositeModels folder, e.i. *cd /TLMPlugin/CompositeModels/OmcOmcDoublePendu*
- To start the simulation run the following: *tlmmanager.exe doublePendulum.xml*

When the simulation is done the result files are located in the respective folders where the models are saved. The MAT files can now be loaded into the respective Modelica tools an analyzed.

The blue plots are from the total model and the red are from DoublePendulum2. As can be seen, we have managed to carry out a parallel simulation with the TLM interface, and the results are almost identical. If the user wants to obtain better results, the 'Connection Form' parameters can be modified in the XML file to match the connection between the two TLM Modelica models even more accurately.

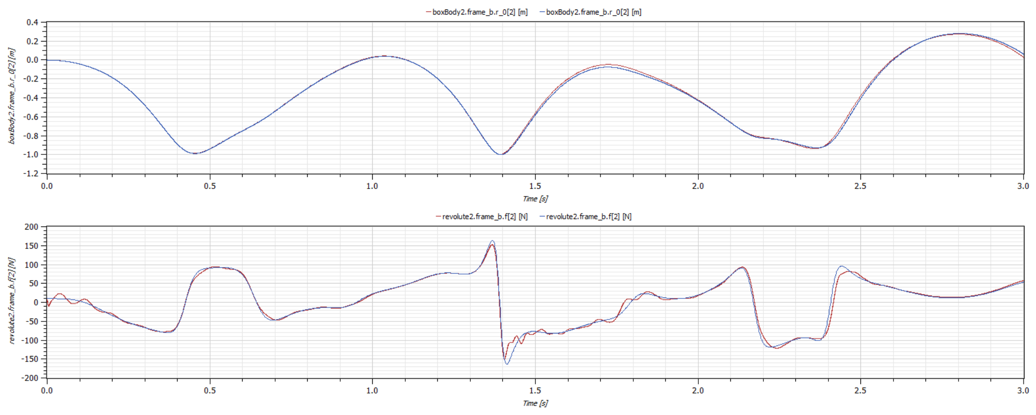


Figure 11.6: Co-simulation of double pendulum results

Bibliography

[1] Johns, P. B. and O'Brien, M., *Use of the transmission-line modelling (t.l.m.) method to solve non-linear lumped networks*, The Radio and Electronic Engineer, 1980.

[2] Krus, P. and Jansson, A, *Distributed Simulation of Hydromechanical Systems*, Third Bath International Fluid Power Workshop, Bath, UK 1990

[3] Yongjoo, K. and Kyuseok, K. and Youngsoo S., and Taekyoon, A. and Kiyoung, C., *An integrated cosimulation environment for heterogeneous systems prototyping*, Design Automation for Embedded Systems, 1998

[4] Stacke, L-E. and Fritzson, D. and Nordling, P., *BEAST—a rolling bearing simulation tool*, Proc. Instn Mech. Engrs, part K, Journal of Multi-body Dynamics, 1999

[5] Krus, P., *Modelling of Mechanical Systems Using Rigid Bodies and Transmission Line Joints*, Transactions of ASME, Journal of Dynamic Systems Measurement and Control. Dec 1999

[6] Agrawal A., Bakshi A., Davis J., Eames B., Ledeczi A., Mohanty S., Mathur V., Neema S., Nordstrom G., Prasanna V., Raghavendra, C., Singh M *MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems*, Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), Snowbird, Utah, June 2001.

[7] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, *The Generic Modeling Environment*, Proceedings of WISP'2001, May 2001, Budapest, Hungary.

[8] Fritzson, P. *Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-Interscience, 2004

[9] Siemers, A. and Nakhimovski, I. and Fritzson, D., *Meta-modelling of Mechanical Systems with Transmission Line Joints in Modelica*, Proceedings of the 4th International Modelica Conference, Hamburg, Germany, 2005

[10] Siemers, A. and Fritzson, P. and Fritzson, D., *Meta-Modeling for Multi-Physics Co-Simulations applied for OpenModelica*, ANIPLA2006 International Congress on 'Methodologies for Emerging Technologies in Automation', University of Rome "La Sapienza" ,November 13-14-15, 2006

[11] de Cogan, D. and O'Connor, W.J. and Pulko, S., *Transmission Line Matrix in Computational Mechanics.* Taylor & Francis, ISBN: 0-415-32717-2, 2006.

[12] Siemers, A. and Fritzson, D., *A Meta-Modeling Environment for Mechanical System Co-Simulations*, The 48th Scandinavian Conference on Simulation and Modeling (SIMS 2007), Gothenburg (Särö), Sweden, October 2007

[13] Fritzson, D. and Ståhl, J. and Nakhimovski, I. *Transmission line co-simulation of rolling bearing applications*, The 48th Scandinavian Conference on Simulation and Modeling (SIMS 2007), Gothenburg (Särö), Sweden, October 2007
I

[14] *http://ptolemy.eecs.berkeley.edu/ptolemyII*, Center for Hybrid and Embedded Software Systems (CHESS) in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley.

[15] TNI-Software, *Cosimate co-simulation software*, http://www.tni-software.com, Last visited on 14 February 2008

[16] MSC-Software, *MSC.ADAMS simulation software*, http://www.mscsoftware.com, Last visited on 14 February 2008