7th EUROMECH Solid Mechanics Conference
J. Ambrósio et.al. (eds.)
Lisbon, Portugal, September 7-11, 2009

COMBINED USE OF MATLAB/SIMULINK AND MULTIBODY SIMULATION SOFTWARE BASED ON C++

Francisco J. González[1], Manuel González[1] and Aki Mikkola[2]

[1] Escuela Politécnica Superior, Universidad de A Coruña
Mendizábal s/n, 15403 Ferrol, Spain
fgonzalez@udc.es
lolo@cdf.udc.es

[2] Department of Mechanical Engineering, Lappeenranta University of Technology
P.O. Box 20, FIN-53851, Lappeenranta, Finland
aki.mikkola@lut.fi

Keywords: Multibody, multi-physics, co-simulation, MATLAB.

Abstract. *Simulation of complex mechatronic systems like an automobile, involving multibody components as well as active electronic control devices, can be accomplished by combining tools that deal with the simulation of the different subsystems. In this sense, it is desirable to couple a multibody simulation software (for the mechanical simulation) with a block diagram simulation software (for the simulation of electronical components), and to employ a co-simulation scheme to synchronize both tools.*

During the last years, the Laboratory of Mechanical Engineering of the University of A Coruña has been developing an open and modular software architecture for the simulation of multibody systems. Support for the analysis of non-mechanical components is not originally provided in this package and it has been added through the coupling with an external simulation tool.

In this research, a communication interface between the multibody simulation software and a platform for numerical computation has been developed and implemented. Several communication techniques have been tested, and optimal application fields have been delimited for each of them. The numerical performance of each technique has been studied, in order to assess its suitability for high demanding applications, such as real-time simulations. Comparisons between monolithic and co-simulation implementations of the models have been carried out, and results have been analyzed to quantify their numerical efficiency.

1 INTRODUCTION

Machines, in general, consist of several different subsystems such as mechanical components and actuators as well as control systems. These subsystems represent engineering disciplines that are coupled together in an assembled machine. Accordingly, the overall performance of a machine is defined by the operation of each individual subsystem as well as interactions of subsystems. For this reason, the traditional design procedure where mechanical components, actuators and control methods are considered separately is not able to produce optimum solutions. The multibody system (MBS) simulation approach meets the challenge and can be used in the design process of a machine that consists of different subsystems. It is noteworthy, however, that complex non-mechanical components such as control loops and actuators often fall beyond the scope of traditional multibody codes.

As the industry requirements increase, the demanded degree of realism in the simulation of multidisciplinary systems is continuously growing, so the engineer needs to take into account different phenomena simultaneously when simulating a system. When evaluating the behaviour of an automobile, for example, not only an accurate representation of its mechanical elements is needed, but also of the electronic control systems (like ABS or traction control), the hydraulic components or the thermodynamics of its engine. The realistic simulation of such multidisciplinary system, as required, for instance, by Human/Hardware in the Loop (HiL) devices, must account each different subsystem in an efficient way.

Several ways of dealing with multidisciplinary systems can be found in the literature, as mentioned by Valasek [1]. Two main approaches can be distinguished: communication between different simulation tools, and uniform modelling. Uniform or monolithic modelling is based on representing all the subsystems of a multi-domain problem in the same language [2]. Specialized software and languages exist for this purpose, such as ACSL [3], VHDL-AMS [4], and Modelica [5], that manage simultaneously the equations of the entire system. Another way of performing uniform modelling is based on the use of general mathematical software for defining and solving the equations of the system. Recently, this task has been simplified by the development of specific-domain modules in block-diagram software, such as SimMechanics and SimHydraulics for MATLAB/Simulink [6]. Coupling of tools, on the other hand, is based on the combination of specialized tools for modelling each sub-domain. These tools are interfaced during execution time in order to emulate the real interaction between physical subsystems. As stated by Kübler and Schiehlen [7], this is the optimal approach for the simulation of multidisciplinary systems. It allows the selection of optimized conditions for the simulation of each subsystem, such as the integration time-step, the numeric solver and other particularized settings. In many cases, these specialized tools have been developed during years by researchers, leading to robust and efficient software and wide collections of tested examples and toolboxes.

Coupling strategies can be further categorized to two main approaches, depending on the way integration is performed. The name *co-simulation* is usually reserved for those cases in which each simulation tool incorporates its own integrator. In this work, when the integration is performed only in one tool that requests values from the others, the name *function call* will be used.

Commercial multibody packages have been incorporating multi-physics capabilities during the last years and many of them, for example SIMPACK [8], offer a wide range of coupling possibilities to external software tools, as well as add-on modules with non-multibody functionality. When the multibody software has been developed by a non-commercial research group, as in the case of academia, and coupling capabilities need to be added to it, the programmer must often choose between several available implementation techniques. Currently,

it is nontrivial to make this decision, as the research about the suitability of the different coupling techniques for particular applications has been short shrifted. In particular, there is a lack of information about the amount of effort the implementation of a communication strategy takes and, more importantly, the efficiency of a specific technique when compared to other strategies applicable to the same problem. A study of the impact in performance of different co-simulation time-steps and processor configurations, in a simulation involving SIMPACK and MATLAB has been carried out in [9] for the model of a truck. However, the evaluation of the computational efficiency of different communication techniques, and a comparison with the performance of equivalent monolithic models, when possible, has not been performed yet. To this end, test models must be selected and built up, and simulations performed in order to measure the overhead the communication techniques give rise to.

A closely related open field of research in the simulation of multidisciplinary systems is the use of multirate integration schemes, which improves the numerical efficiency during the simulation of interacting subsystems with very different time scales. Multirate algorithms have been developed ([10,11]), while, however, the implementation of these techniques in the communication between software packages, specially when block-diagram software is involved, is still in progress. It is noteworthy that the numerical performance of multirate algorithms dependents greatly of the co-simulation strategy selected for solving the problem. The understanding of the limitations imposed by block-diagram software packages, and the definition of a convenient interface between them and other simulation tools is the first step in the definition of a general scheme for multirate co-simulation.

In this work, communication techniques to external simulation tools have been used for widening the capabilities of existing MBS software, through the addition of functionality with numerical computation platforms (such as MATLAB/Simulink [6], Scilab [12] or Mathematica [13]). To this end, communication possibilities between the software and MATLAB/Simulink are examined in detail. MATLAB has been selected for this work because of its wide acceptance in the research community, derived from its versatility and easiness of programming. A practical way of performing the communication in real cases has been implemented for each technique. It is important to note that the coupling techniques introduced in this study are not limited to a specific mathematical package, but they can also be applied to other similar tools, as similar communication capabilities are available in them. Finally, a generic co-simulation interface, which manages the communication between MBS software and Simulink block-diagram package, has been created and implemented. This interface is intended to allow multirate co-simulation, with different synchronization methods, between simulation tools.

This paper is organized as follows: Section 2 describes the MBS software that has been used in the research. Section 3 gives a general review of the existing techniques for communicating a multibody package to external simulation tools. In Sections 4 and 5, these techniques are implemented in the MBS software and a general software tool for numeric computations. Introduced computational strategies are utilized in two example problems. Finally, the conclusions of the work are summarized.

2 MULTIBODY SIMULATION SOFTWARE

An MBS software tool implemented in C++ is used in this research. This software has been developed by the Laboratory of Mechanical Engineering of the University of A Coruña during the last four years. The software operates under the principle of being an open and modular platform in such a way that it can be developed and used by a wide community of researchers. Its modularity allows the easy implementation and addition of new components

and functionalities as well as interfacing with other open and commercial packages and libraries.

This multibody software is a general purpose program aimed at the simulation of generic multibody mechanisms accounting for large rotations and highly non-linear equations. For a multibody system, nonlinear equations of motion can be written as a system of Differential Algebraic Equations (DAE) as follows:

$$\begin{aligned} \mathbf{M}\ddot{\mathbf{q}}+\boldsymbol{\Phi}_{\mathbf{q}}^{\mathrm{T}}\boldsymbol{\lambda}&=\mathbf{Q} \\ \boldsymbol{\Phi}&=\mathbf{0} \end{aligned} \tag{1}$$

where $\mathbf{M}$ is the mass matrix of the system, $\ddot{\mathbf{q}}$ is the vector of the second derivatives of the generalized coordinates (accelerations), $\boldsymbol{\Phi}$ is the vector of constraint equations of the system and $\boldsymbol{\Phi}_{\mathbf{q}}$ is the Jacobian matrix of constraint equations with respect to generalized coordinates, $\boldsymbol{\lambda}$ is the vector of Lagrange multipliers, and $\mathbf{Q}$ is the vector of generalized forces that applies to the generalized coordinates.

The simulation software is designed to be able to manage different types of coordinates. In this study, however, natural coordinates (global and dependent) [14] are used for modelling the systems. Natural coordinates describe the position of the elements of the system by means of basic points and unit vectors associated with the bodies of the system. For this reason, there is no need for the use of rotation parameters, such as Euler angles, when describing the rotation of the bodies. At the moment, only rigid bodies can be modelled while, however, an additional module for the representation of flexible mechanisms will be implemented in the future.

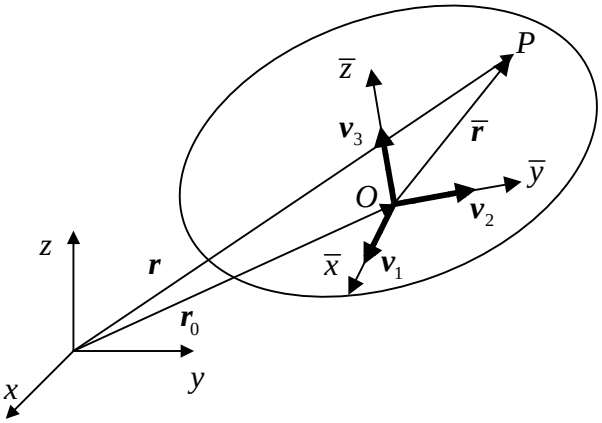

Figure 1: Generic rigid body in natural coordinates.

A description of a generic rigid body can be seen in Figure 1. The global position of an arbitrary particle of the rigid body *P* can be expressed by its position vector $\mathbf{r}$ defined with respect to unit vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ that form a base for the local frame of reference of the body. The unit vectors need not be co-lineal with the local axes of the body, $\bar{x}$, $\bar{y}$ and $\bar{z}$. The position of any particle *P* in the body can be expressed as a linear combination of the elements of its base in the following way:

$$\mathbf{r}=\mathbf{r}_0+\bar{\mathbf{r}}=\mathbf{r}_0+\alpha\mathbf{v}_1+\beta\mathbf{v}_2+\gamma\mathbf{v}_3 \tag{2}$$

where $\mathbf{r}_0$ is the position vector of the origin of the local frame of reference of the body, $\bar{\mathbf{r}}$ is the position vector of particle *P* in the local frame of reference, and α, β and γ are linear combination constant coefficients.

Constraint equations representing kinematic joints between bodies must be added in order to complete the modelling of the system. These equations are defined in the constraint vector **Φ**. Using a proper selection of the basic points and unit vectors of the body, mass matrix **M** remains constant during large rotations. This unique feature of the natural coordinates simplifies the equations of motion since inertial forces that depend quadratically on velocities do not appear in them.

The software developed by the University of A Coruña can manage both dense and sparse matrices. These matrices are stored in memory and managed through the use of templated C++ classes defined by the uBLAS library [15]. The structure of the multibody software allows the selection of different integrators and dynamic formulations for different problems. The numeric core module for the solution of the equations of motion employs streamlined numeric algorithms based on BLAS [16] routines and permits the selection of several high-performance dense and sparse linear solvers for symmetric and non-symmetric cases such as LAPACK [17], CHOLMOD [18], KLU [19] and WSMP [20]. Some of these implementations have been proved to be highly efficient when solving the dynamics of multibody systems [21]. Moreover, the algorithm that carries out the integration of the equations of motion has been parallelized via OpenMP directives and the inclusion of parallel linear equation solvers, resulting on higher performance when medium- and high-size problems are to be solved [22].

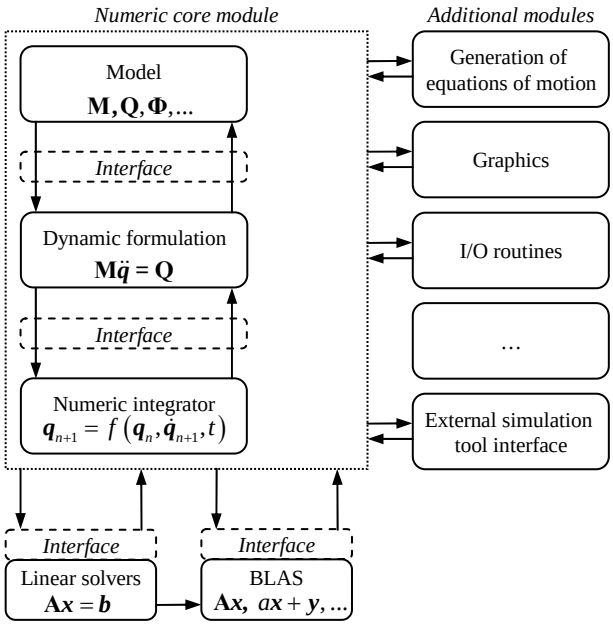

Figure 2: Layout of multibody simulation software.

The main structure of the MBS simulation software is depicted in Figure 2. The numeric core module of the multibody software consists of a set of three basic components: a numeric Ordinary Differential Equations (ODE) integrator, a dynamic formulation used for converting the equations of motion expressed in Equation (1) into ODE, and a model representing the simulated system. Abstract base classes are defined for each of the above-mentioned components. Specific components are then derived from these base classes through C++ inheritance, thus ensuring that every derived component shares the main features of the base class. Accordingly, in the software, a standard interface between model, integrator and formulation is used for the exchange of data. This allows the easy replacement of components without loss of functionality while avoiding the rise of incompatibilities. It is also noteworthy that other

modules have been developed to add graphic representation, file data I/O, and automatic generation of the equations of motion.

The modular structure of the software architecture simplifies the addition of new capabilities through the definition of new modules. The communication with external software is executed through a new module for tool coupling, avoiding thus the need of re-writing the core module, with only the implementation of the details of the corresponding model to be performed.

Numerical examples in this work are described using planar natural coordinates and the equations of motion of the multibody system are expressed using the index-3 augmented Lagrangian formulation as explained in [23]. The non-linear equations of motion are integrated using the trapezoidal rule. This combination has demonstrated favourable performance and robustness features in previous works ([24], [25]). The equations of motion of the system introduced in Equation (1) can be rewritten in the following way:

$$\begin{aligned} \mathbf{M}\ddot{\mathbf{q}} + \boldsymbol{\Phi}_{\mathbf{q}}^T \alpha \boldsymbol{\Phi} + \boldsymbol{\Phi}_{\mathbf{q}}^T \boldsymbol{\lambda} = \mathbf{Q} \\ \boldsymbol{\lambda}_{i+1} = \boldsymbol{\lambda}_i + \alpha \boldsymbol{\Phi}_{i+1}, \quad i = 0,1,2... \end{aligned} \tag{3}$$

In Equation (3), Lagrange multipliers are obtained from an iterative process where a penalty factor (α) and Lagrange multipliers from the previous time step $\boldsymbol{\lambda}_0$ are used.

Difference equations for the trapezoidal rule to describe velocities and accelerations in time step n+1 can be written as follows:

$$\begin{aligned} \dot{\mathbf{q}}_{n+1} &= \frac{2}{\Delta t}\mathbf{q}_{n+1} + \hat{\dot{\mathbf{q}}}_{n+1}; \qquad \hat{\dot{\mathbf{q}}}_{n+1} = -\left(\frac{2}{\Delta t}\mathbf{q}_n + \dot{\mathbf{q}}_n\right) \\ \ddot{\mathbf{q}}_{n+1} &= \frac{4}{\Delta t^2}\dot{\mathbf{q}}_{n+1} + \hat{\ddot{\mathbf{q}}}_{n+1}; \quad \hat{\ddot{\mathbf{q}}}_{n+1} = -\left(\frac{4}{\Delta t^2}\mathbf{q}_n + \frac{4}{\Delta t}\dot{\mathbf{q}}_n + \ddot{\mathbf{q}}_n\right) \end{aligned} \tag{4}$$

Introducing difference equations (4) into the equations of motion (3) at time step n+1, the dynamic equilibrium can be expressed as a nonlinear algebraic system of equations, whose unknowns are the dependent positions $\mathbf{q}_{n+1}$, as follows:

$$\mathbf{f}(\mathbf{q}) = \mathbf{M}\mathbf{q}_{n+1} + \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}_{n+1}}^T\left(\alpha\boldsymbol{\Phi}_{n+1} + \boldsymbol{\lambda}_{n+1}\right) - \frac{\Delta t^2}{4}\mathbf{Q}_{n+1} + \frac{\Delta t^2}{4}\mathbf{M}\hat{\ddot{\mathbf{q}}} = \mathbf{0} \tag{5}$$

This non-linear system can be solved by means of the Newton-Raphson iteration by defining the approximate tangent matrix as follows:

$$\begin{aligned} \left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]_i \Delta\mathbf{q}_{i+1} &= -\left[\mathbf{f}(\mathbf{q})\right]_i \\ \left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] &\cong \mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\left(\boldsymbol{\Phi}_{\mathbf{q}}^T \alpha \boldsymbol{\Phi}_{\mathbf{q}} + \mathbf{K}\right) \end{aligned} \tag{6}$$

where $\mathbf{C}$ and $\mathbf{K}$ are the damping and elastic forces of the system, respectively. After convergence is obtained, the obtained positions $\mathbf{q}$ satisfy the equations of motion (3) and the constraint equations $\boldsymbol{\Phi} = \mathbf{0}$ within iteration limits. However, the obtained sets of velocities $\dot{\mathbf{q}}^*$ and accelerations $\ddot{\mathbf{q}}^*$ may not necessarily satisfy the conditions $\dot{\boldsymbol{\Phi}} = \mathbf{0}$ and $\ddot{\boldsymbol{\Phi}} = \mathbf{0}$. For this reason, corrected velocities $\dot{\mathbf{q}}$ and accelerations $\ddot{\mathbf{q}}$ have to be obtained from initial ones ($\dot{\mathbf{q}}^*$ and $\ddot{\mathbf{q}}^*$) by mass-damping-stiffness orthogonal projections in order to fulfill $\dot{\boldsymbol{\Phi}} = \mathbf{0}$ and $\ddot{\boldsymbol{\Phi}} = \mathbf{0}$ conditions as follows:

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\dot{\mathbf{q}} = \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\right]\dot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}}^T \alpha \boldsymbol{\Phi}_{\mathbf{q}}$$
$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\ddot{\mathbf{q}} = \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\right]\ddot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}}^T \alpha \left(\dot{\boldsymbol{\Phi}}_{\mathbf{q}}\dot{\mathbf{q}} + \dot{\boldsymbol{\Phi}}_t\right) \tag{7}$$

3 COMMUNICATION CASES

The expansion of the multibody software via communication with external simulation tools can be performed in several ways, which can be categorized as data files exchange, function call and co-simulation approaches.

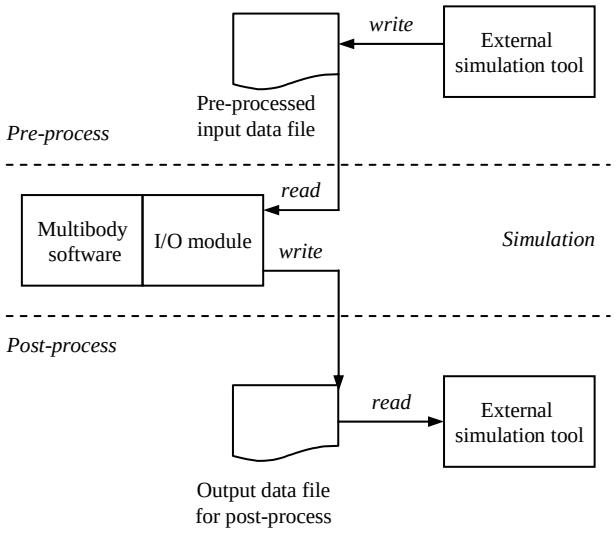

Figure 3: Data file input-output configuration.

The most straightforward and easy to implement way of sharing data between two different simulation environments is the use of importing and exporting of data files. As the computational cost of read/write operations is high, this technique should not be applied during runtime. For this reason, files exchange approach should be reserved for pre- and post-processing operations, where computational efficiency is not a key factor. A scheme of this method is described in Figure 3. In the MBS simulation field, a large variety of tasks can be managed with files exchange approach adding to the multibody software the functionality of an external processing tool. The off-line realistic graphic representation of results and the pre-processing of complex dynamical terms when these are remaining constant during simulation are examples of this approach. The software requirements for the use of this strategy are the existence of a common data format, understandable by the involved packages, and the availability of input-output routines for handling the data files in each program.

An alternative to data files exchange, more adequate for runtime, are function calls from one simulation tool to another. In this work, the name function call is reserved for those communications in which only one of the software tools is actually performing a numeric integration, while the other one returns values on request, from the states passed by the integrator tool. This configuration can be achieved through code exporting (via joint compiling together with the integrator tool or pre-compiled as a library) or by direct communication between processes. Application fields of the function call strategy would be complex function

evaluations during runtime, table look-up and other processes in which numerical integration is not present.

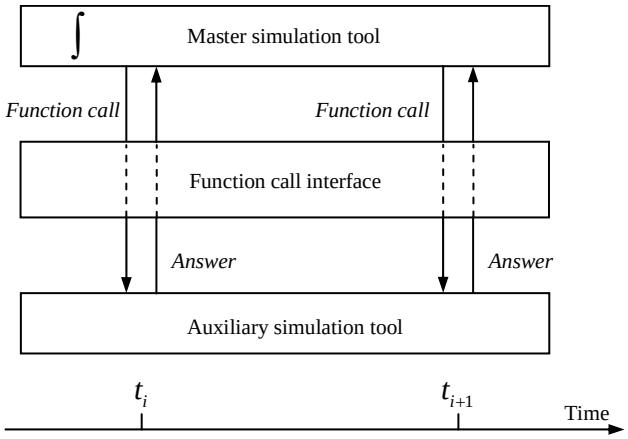

Figure 4: Generic function call configuration.

The implementation of this technique requires the development of an interface between the software tools to allow the main process to use the functionality of the auxiliary software and to receive the return data conveniently. Data formats in different tools are often incompatible, so translation routines may be necessary for the correct transmission of information. A simplified depiction of this technique can be seen in Figure 4. The block representing the auxiliary software tool at the bottom of the figure can be a standalone process, if direct communication between processes is used, a library or even exported source code, that has been previously compiled together with the source code of the driver program. The availability of these methods is determined by the nature of the external tool, as it may or not allow communication to external processes (for example, via TCP/IP) or the access to inner functions in case of it is compiled as a library.

Finally, a co-simulation approach in the strict sense can be developed, in which two simulation tools, each of them with its own states and integrator, share data at defined synchronization points [26]. Again, code export or direct communication between processes can be used to implement this configuration. In the case of a multibody simulation tool, state-space equations can be represented by

$$\begin{cases} \dot{\mathbf{x}}_m(t) = \mathbf{f}\left(\mathbf{x}_m(t), \mathbf{u}_m(t)\right) \\ \mathbf{y}_m(t) = \mathbf{g}\left(\mathbf{x}_m(t)\right) \end{cases} \tag{8}$$

where $\mathbf{x}_m$ are the states of the multibody system, $\mathbf{u}_m$ the inputs to the system and $\mathbf{y}_m$ the system outputs. An analog expression can be used for the external simulation tool

$$\begin{cases} \dot{\mathbf{x}}_e(t) = \mathbf{f}\left(\mathbf{x}_e(t), \mathbf{u}_e(t)\right) \\ \mathbf{y}_e(t) = \mathbf{g}\left(\mathbf{x}_e(t)\right) \end{cases} \tag{9}$$

being the inputs of a system the outputs of the other one

$$\begin{cases} \mathbf{u}_e(t) = \mathbf{y}_m(t) \\ \mathbf{u}_m(t) = \mathbf{y}_e(t) \end{cases} \tag{10}$$

Nowadays, state-of-the-art commercial software performs co-simulation at constant time steps, with the same integration time-steps in every subsystem, although research is being carried to introduce multirate methods in co-simulation environments [27]. Even with constant and equal time-steps in each subsystem, the evaluation of the inputs for each subsystem, given by Equation (10), at synchronization point t_i can be performed in several ways. A frequent strategy is assuming that each subsystem inputs can be considered constant during each time-step $\left[t_i, t_{i+1}\right]$, which leads to

$$\begin{cases} \mathbf{u}_e\left(t\right) = \mathbf{u}_e\left(t_i\right) = \mathbf{y}_m\left(t_i\right) \\ \mathbf{u}_m\left(t\right) = \mathbf{u}_m\left(t_i\right) = \mathbf{y}_e\left(t_i\right) \end{cases} \tag{11}$$

This approach, known as constant extrapolation, has been followed in this work, as the detailed testing of different interpolation degrees and multirate techniques falls beyond the scope of this paper. Direct co-simulation, in which co-simulated states are exchanged once in each integration step, and then each subsystem proceeds its own integration independently, has been used.

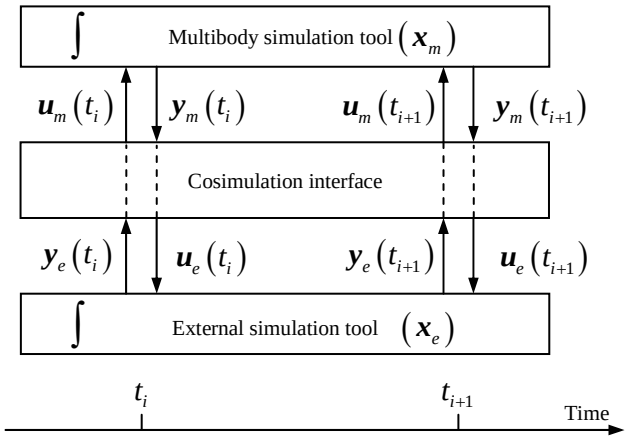

Figure 5: Generic co-simulation configuration.

As it was the case in the function call strategy, co-simulation can be implemented on the basis of intercommunication between processes, or through code export. Again, translation routines between data storage formats are likely to be necessary. The synchronization of integrators and the exchange of data can be managed by a co-simulation interface, which can be implemented in one of the communicating software tools. A scheme of this composition is shown in Figure 5.

In order to test the described communication techniques, the MBS software described in Section 2 has been linked to MATLAB/Simulink. MATLAB is a technical computing tool that provides state-of-the-art algorithms for a wide range of applications (optimization, control, data acquisition and analysis). MATLAB add-in Simulink can be considered as the de facto standard for model-based design of control systems. This software package includes a library of a wide variety of components and it allows the user to create new elements in a straightforward manner. It is important to note that MATLAB/Simulink code has to be interpreted during runtime, which leads to a considerable increase in simulation time and inefficient execution. This rules out the software for demanding applications, for example real-time simulation. Communication between the MBS software and MATLAB/Simulink programs, representing control loops, actuators and other external components, can provide an additional functionality that is missing in the original version of the multibody software.

The described techniques can be applied to other software tools, different from MATLAB. In general, communication between processes can often be achieved if the software supports the use of inter-process communication (IPC), like sockets. The exporting of code can be performed with calls to dynamically linked libraries, with the corresponding import libraries and header files, if necessary.

4 IMPLEMENTATION OF FUNCTION CALL

The runtime call to MATLAB functions from the multibody software would be desirable in order to evaluate complex force functions or to access look-up tables. Additionally, MATLAB can also be used as a test environment for the definition of new implementations for integrators, formulations or models. These could be written in MATLAB's easy-to-use M language, and called from the multibody software as library functions in order to test their correctness before performing their final implementation in an efficient language such as C or FORTRAN. This would make possible the definition and testing of new models even for users without exhaustive programming skills. In this study, MATLAB Engine, MATLAB Compiler and A MEX API of functions based approaches have been tested.

The Engine library is a set of routines that allows calling MATLAB functionality directly from external C/C++ and FORTRAN programs. The Engine is a way of intercommunicating running processes such that a MATLAB command window must be open, waiting for receiving the commands sent by the external program and executing them. As the Engine uses its own data structure, *mxArray*, to exchange information with the caller program, several translation functions had to be defined in order to manage the data type and to make it compatible with the data types used in the multibody program. Once this problem has been solved, MATLAB functions can be called from the C++ code of the multibody tool. It is noteworthy that the Engine receives its commands as a string of characters which must be parsed resulting to deceleration of the execution of the code.

Function call has also been achieved through code exporting, with the use of MATLAB Compiler, transforming *.m* code files into dynamically linked libraries (*.dll*, *.so*). The libraries are then loaded by the multibody software during runtime allowing the invocation of functions. As the Engine does, the Compiler uses its own storage data type, *mwArray*, and translation routines between the MBS code and the Engine must be written. The generated library has still to be interpreted during runtime, as C/C++ output files are only wrappers for original routines.

A third way of communicating both tools is the definition of an application programming interface (API), which allows calling from MATLAB the functions that are defined and implemented in the multibody package. This way, MATLAB acts as driver tool, starting the integration performed by the MBS software. The API consists of a series of MEX functions that manage the data types defined by MATLAB and make the convenient translation to those types that the C++ program uses and vice versa.

Equivalent methods of function call communication can be found in similar numerical software, such as the *intersci* program, which allows calling C and FORTRAN routines from Scilab, and the calling routines defined in *CallScilab.h*, which make Scilab work as a calculus engine.

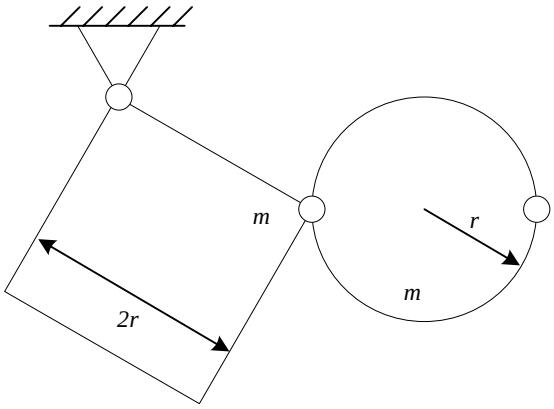

Figure 6: Double pendulum.

The function call strategy has been used to generate the equations of motion of a double pendulum in the three different described ways, in order to test the above mentioned communication methods. This simple example has been chosen, as there is no practical increase of complexity derived from applying the function call technique to more involved problems.

The double pendulum is shown in Figure 6. In this study, the mass (*m*) and radius (*r*) parameters have been set to 1 kg and 1 m. The code for the updating of the dynamic terms of the system, including the mass matrix **M**, the constraints vector **Φ**, the Jacobian matrix of the constraints vector $\mathbf{\Phi_q}$ and the generalized forces vector **Q**, is written in ***.m*** files and it is accessed from the MBS simulation software through function call methods. In this way, the integrators and formulations written in C++ can be applied to easy-to-code *.m* file models. A similar approach could be taken in order to test formulations or integrators written in MATLAB with already tested problems, avoiding the need for translating them to C++, and to invoke specific MATLAB functionality such as involved matrix operations or complex function evaluations.

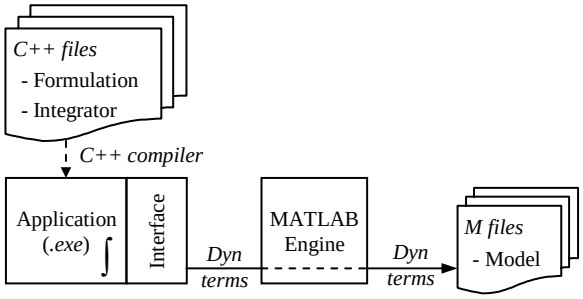

Figure 7: Function call configuration with MATLAB Engine.

The function call configuration through the Engine is represented in Figure 7. The MBS software acts as a main tool, integrating the positions of the double pendulum, while the evaluation of dynamic terms is performed, through the Engine, by calls to the *.m* files that code the model.

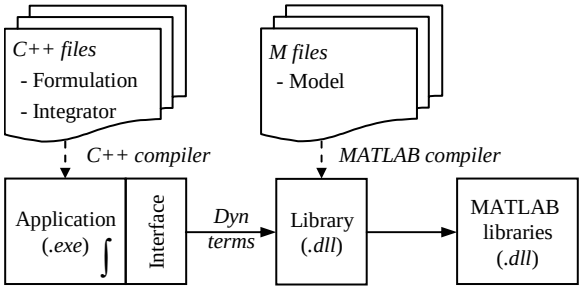

Figure 8: Function call configuration with MATLAB Compiler.

The use of the Compiler for the *.m* files removes the need for the use of the Engine, as shown in Figure 8, replacing the process communication with the export of the pre-compiled code. The evaluation of dynamic terms is directly called from the main application while the library that wraps the routines coded in *.m* files still needs to invoke additional MATLAB functions.

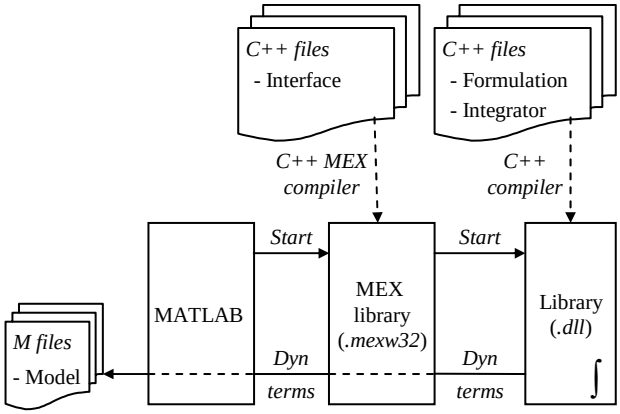

Figure 9: MEX configuration for function call.

Figure 9 shows the layout of the communication through the use of a MEX function. Under this configuration, the interface routines are separated from the MBS software and compiled into a library that manages the communication between MATLAB and the MBS software, compiled as a dynamic library. The MBS code calls the model *.m* files for the evaluation of the dynamic terms of the model through this MEX function and this one, in time, through MATLAB.

Method of update of dynamic terms	**Time elapsed ($\Delta t = 10^{-3}$ s)**	**Ratio**	**Time elapsed ($\Delta t = 10^{-2}$ s)**	**Ratio**
MBS code alone	0.05 s	1	0.008 s	1
MATLAB Engine	18.48 s	368.16	3.57 s	422.46
MATLAB Compiler	5.56 s	110.70	1.07 s	126.28
MEX API of functions	0.79 s	15.75	0.14 s	16.68
Number of solver iterations	10,000		1,840	

Table 1: Elapsed times in a 10 s dynamic simulation of a double-pendulum.

Two simulations of 10 seconds have been performed using a penalty factor of $\alpha = 10^8$ and constant integration time-steps of 10^{-3} s and 10^{-2} s, respectively. The MBS software is config-

ured to use LAPACK routines *gtrf* and *gtrs* as linear solver, which have been proven to be efficient for small-sized problems.

The elapsed times in calculations, on an AMD Athlon 64 3000+, at 1.81 GHz with 1.00 GB of RAM, are summarized in Table 1. As the input terms are the same in every implementation, output results (positions, velocities and accelerations during the motion) are identical for each time-step, independently of the method used for providing the dynamic terms. The ratios defined in the table refer to the elapsed time of the correspondent function call implementation when compared to pure C++ MBS code. The number of iterations is the number of times the iterative solution of the system in Equation (6) has been performed. The evaluation of dynamic terms takes place within the Newton-Raphson iteration loop. This, together with the fact that the use of function call methods slows down the execution of the code, makes negligible the amount of computational time elapsed out of the iterative loop. With the pure C++ implementation, however, the code out of the loop takes around 20% of the time, and this explains the variations that appear in the ratios when using different time-steps.

As it was expected, the use of communication techniques with external simulation tools slows down the execution of the program. The use of MATLAB Engine should be discouraged when calls to the auxiliary tool are repetitive (for example, several times in each time-step); it has been estimated that the parsing of a single empty call takes 0.25 m*s* of computation. The use of MATLAB Compiler removes the need of parsing string instructions as function calls are performed directly on routines stored in dynamically linked libraries. Even so, MATLAB C-translated code is not as efficient as specifically designed MBS programs. The source code is still interpreted during runtime and, moreover, the need of converting data structures between languages still exists.

The implementation of the multibody code as MEX API of functions, shown in Figure 9, has yielded the best performance. This approach, nevertheless, requires a high development effort due to the need for building a MATLAB compliant C interface for each function in the multibody package. It is surprising that the implementation of the MBS code as a MEX API leads to almost 8 times faster execution time when compared to MATLAB Compiler. This may be related to the way in which MATLAB functionality is invoked from the compiled library in the latter case.

5 IMPLEMENTATION OF CO-SIMULATION

Under the co-simulation approach, the MBS simulation tool has been connected to the MATLAB add-on Simulink, a block-diagram simulation tool. With this configuration, two integrators are coupled in the simulation process: the MBS integrator contained in the multibody software and the general purpose integrator in Simulink.

Data interchange between two processes running simultaneously can be carried out using a TCP/IP connection through standard sockets. To this end, the MBS software is modified in order to make it behave as a server socket. Accordingly, a user-defined block (*S-function*) is added to the Simulink model to act as a client socket. In other similar block simulation packages, the role of the *S-function* block can be performed by an equivalent component, such as the *UserCode* block in SystemBuild [28] and the *C* or *Fortran* block in Scicos [29]. Thus, the interface is split into two parts, one in the Simulink environment and one in the MBS software. Both must be correctly coordinated to allow a suitable interchange of information.

With the code exporting approach, the MBS code can be compiled as a dynamically linked library (*.dll* or *.so*) and directly called from an *S-function* block. In this case, the *S-function* operates as an interface between the MBS original code (compiled as a MEX function) and the Simulink model, and must manage the required conversions between storage formats in both environments. In spite of code exporting, two integrators are acting simultaneously and

for this reason a careful coordination between them is required. Simulink behaviour is, in many aspects, beyond the control of the user, so the co-simulation interface has to be specifically defined to fit Simulink.

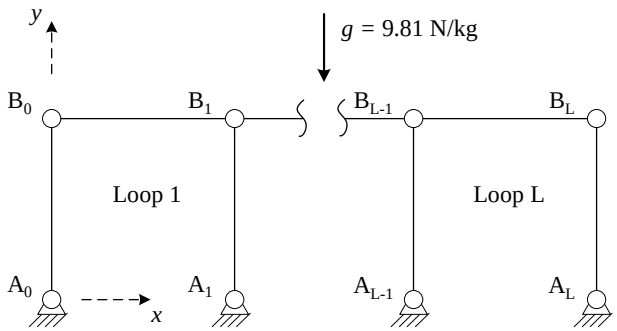

Figure 10: *L*-loop fourbar linkage.

In order to test co-simulation, a multi-physics model, composed by an engine and a mechanical system is simulated. Each subsystem is modelled and integrated in a different environment. The engine model has been obtained from Simulink library of example models and is based on results published by [30]. It describes the thermodynamic simulation of a four-cylinder spark ignition internal combustion engine. The multibody system moved by the engine is a planar assembly of four-bar linkages with *L* loops, composed by thin rods of 1 m length with a uniformly distributed mass of 1 kg, moving under gravity effects. Initially, the system is in the position shown in Figure 10 and the velocity of the *x*-coordinate of point B_0 is +30 m/s. This mechanism has been previously used as a benchmark problem for multibody system dynamics [31,32]. It has been selected for this work because it allows testing the effect of variations in the problem size without modifying the structure of the model, just by adding more loops to the mechanism.

In this example, the simulation time is 30 s. A penalty factor of $\alpha = 10^8$ and a constant integration time-step of 10^{-3} s have been used. The MBS software is configured to use LAPACK routines *gtrf* and *gtrs*, as in the previous example.

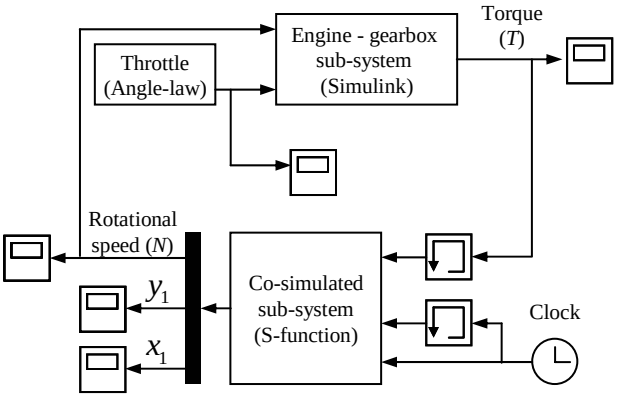

Figure 11: Simplified Simulink model for co-simulation, implemented with an *S-function*.

The engine provides a motor torque to the linkage through a gearbox, which is also modelled in Simulink. A constant rotational damping is considered to act on the mechanism, of value 3.18 Ns/rad. Both damping and motor torque are assumed to be applied on the rotational coordinate of point A_0. The angular speed of the linkage is returned to the engine model

as input value, together with the *x* and *y* positions of the first point of the linkage, for graphical output. The throttle angle of the engine is guided through a pre-defined angle-law. The resulting Simulink model can be seen in Figure 11. The use of memory blocks is motivated by the need of avoiding algebraic loops.

The communication between simultaneously running processes has been performed with the two different ways above described. A first approach has involved the designing and building of an *S-function* block in Simulink that performs the MBS integration and manages the exchange of data between processes. The MBS code is compiled as a library and then called from the interface block each time Simulink asks for its return values.

Direct co-simulation with the same integration time-step in both subsystems has been used. The values of the exchanged variables have been taken as constant from one time-step to the next one (constant interpolation).

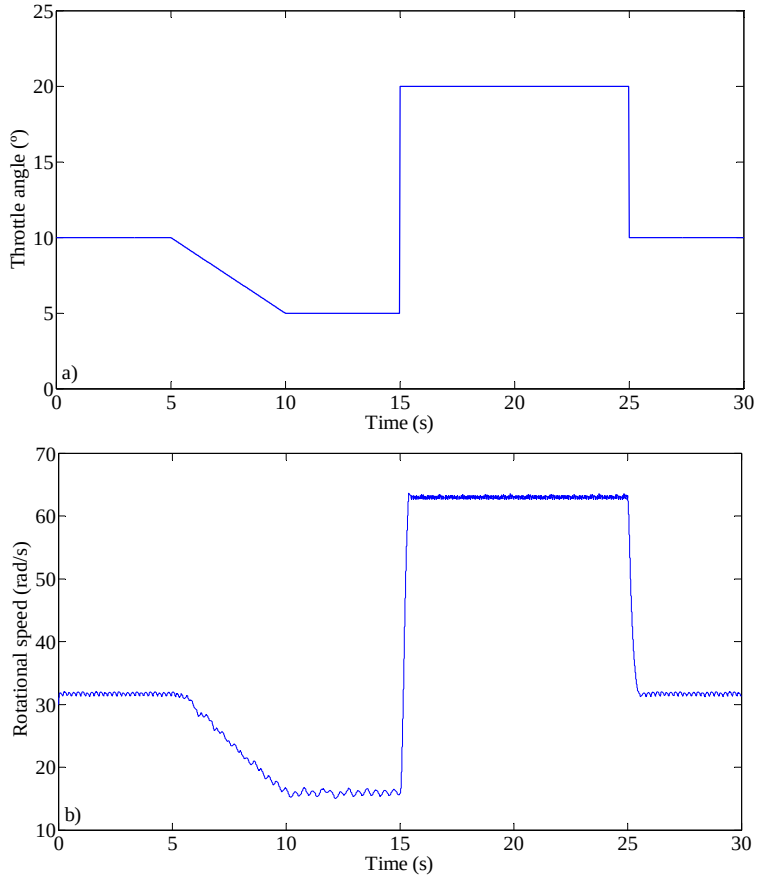

Figure 12: Throttle angle (a) and rotational speed of the mechanism (b) for a 30 s simulation of a 1-loop linkage.

Results of the simulation can be seen in Figure 12, for a 1-loop linkage. The angle-law of the engine throttle is pictured at the top of the figure. The rotational speed of the mechanism, depicted below, shows that the linkage follows the input given by the pedal angle, with the limitations imposed by its rotational inertia and damping.

The second tested approach is based on of TCP/IP connection between both simulation tools. The MBS simulation tool was modified to make it behave as a server socket. The Simulink model replaces the code of the S-function block in order to make it act as a client socket. Thus, modifications in the code of both tools had to be added; moreover, the communication sequence between both subsystems had to be separately coded in each environment, adding an extra burden to the task of keeping the synchronization of the integrators. Every simulation

setting, such as penalty and integration time-step, was taken equal to that of the previous implementation. Again, direct co-simulation with equal time-step in both subsystems has been used, with similar results to those shown in Figure 12 b).

The performance of the described techniques has been tested against a model of the whole system, entirely built in Simulink with SimMechanics elements. The computational efficiency of this model has been further improved via Real-Time Workshop (RTW) package, translating the model to C-code. Simulink *ode1* integrator has been used in simulations, as the use of higher-order integrators would give higher precision at the cost of increasing the simulation time. The integration time-step has been kept at 10^{-3} s for every technique. A comparison of the elapsed times for a 30 s simulation can be seen in Figure 13. The co-simulation methods are designed as *Library call*, for the implementation where the S-function calls MBS code compiled as a library, and *TCP/IP*, when the communication is performed via sockets between simultaneously running processes.

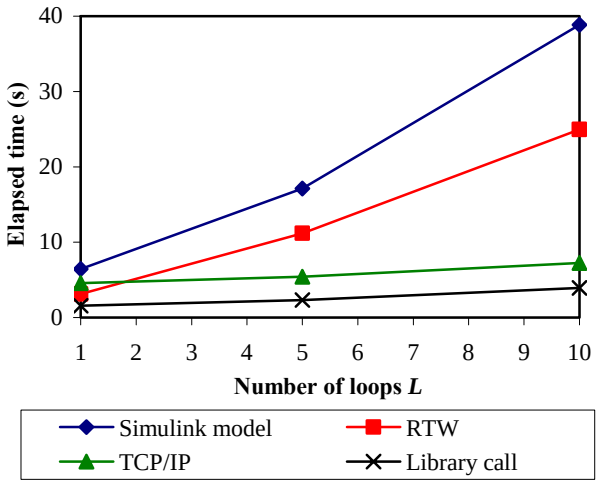

Figure 13: Elapsed times for a 30 s simulation of the *L*-loop linkage, with different simulation techniques.

Results show that the elapsed time for the Simulink model, as expected, grows fast when the number of variables of the problem increases. This is valid even in the case of a simple integrator *ode1*. The use of RTW mitigates this problem and reduces the calculation time between a 30% and a 50%. However, the use of co-simulation techniques leads to lower computation times, as they permit taking advantage of the highly optimized routines of the MBS code, reducing thus the time needed for calculating the mechanic subsystem of the problem. It can be seen that the library call implementation is somewhat faster than the TCP/IP method, as the overhead from socket communications is not present.

Trends indicate that co-simulation will achieve greater differences with respect to models fully implemented in Simulink as the number of variables of the problem increases. In fact, the use of sparse solvers such as KLU has optimized even more the computations for linkages with more than 10 loops, achieving real-time simulation (less than 30 s of computations) for models up to 280 variables. This upper limit would allow the efficient simulation of many non-academic multidisciplinary systems.

6 CONCLUSIONS

In this study, two approaches of sharing data between two different simulation tools, which can be used during runtime, have been described. The studied approaches are function call and co-simulation. Both approaches can accomplish the code exporting as well as direct

communication between processes. The third approach for sharing data between two different simulation tools, data file exchange, is computationally expensive and for this reason it should be reserved for pre- or post-processing of large amounts of data.

Three function call methods between an MBS simulation code and MATLAB have been described is this study. A comparison between MATLAB Engine, MATLAB Compiler and the development of a MEX API of functions has been performed. The methods have been used for generate the dynamic terms in the equations of the motion of a double pendulum. Results show that function calls introduce a considerable overhead with respect to the original MBS software, with elapsed time ratios ranging from 15, in the case of the most efficient implementation (MEX API) to 400, in the case of using MATLAB Engine. Results show that these methods are not suitable for highly-demanding simulations (for example, real-time settings) while, however, they can be used in off-line simulation for complex function evaluation and development and testing of new models, integrators and formulations. As the relatively poor performance of the tested methods is more related to MATLAB structure rather than to limitations in the communication techniques, function call should not be ruled out for more demanding applications when the external simulation tool to be coupled to the MBS code is an efficient and optimized code. The use of auxiliary software as a calculation engine is the easiest to implement. However, its use should be discouraged when it implies the parsing of instructions as it was the case in the tested platform. If the multibody software is written in a computationally efficient language, it is preferable to import it in the auxiliary tool as an export library. A common storage format in both environments will help the efficiency of the overall assembly.

The co-simulation approach allows the efficient realistic simulation of multi-physics phenomena. The definition of a co-simulation interface allows communicating two different simulation tools with minimal impact in their inner -and often heavily optimized- code structure. Additionally, the co-simulation interface is able to deal with block-diagram software requirements without substantially modifying the integration algorithm of each subsystem. An implementation based on calls to a previously compiled library and inter-process communication via TCP/IP sockets were successfully implemented and tested. Simulink models for co-simulation have been built for the two approaches. Additionally, two equivalent monolithic models for the same problem have been built: a Simulink model of the whole system, where the mechanical components have been modelled with SimMechanics blocks, and a translated to C version of the model via RTW. In the tested examples, co-simulation models have shown higher efficiency than their counterpart monolithic models. Results suggest that co-simulation could be used even in demanding applications such as in real-time settings or intensive design study analyses. The defined co-simulation interface, used in this study, allows the implementation of several interpolation methods and the use of multirate integration techniques, which are being currently investigated and constitute a presently open line of research.

ACKNOWLEDGEMENTS

This research has been sponsored by the Spanish MEC, through the F.P.U. Ph.D. fellowship No. AP2005-4448.

REFERENCES

[1] M. Valasek, Modelling, Simulation and Control of Mechatronical Systems. M.Arnold and W.Schiehlen eds. *Simulation Techniques for Applied Dynamics*, 75-140. Springer Wien New York, 2008.

[2] J. C. Samin, O. Bruls, J. F. Collard, L. Sass, and P. Fisette, Multiphysics Modeling and Optimization of Mechatronic Multibody Systems. *Multibody System Dynamics*, **18**, 345-373, 2007.

[3] The AEgis Technologies Group, Inc., ACSLX. http://www.acslsim.com/, 2009.

[4] IEEE 1076.1 (VHDL-AMS) Working Group, VHDL-AMS. http://www.eda.org/vhdl-ams/, 2008.

[5] Modelica Association, Modelica. http://www.modelica.org/, 2009.

[6] The Mathworks, Inc., MATLAB. http://www.mathworks.com/, 2009.

[7] R. Kubler, and W. Schiehlen, Modular Simulation in Multibody System Dynamics. *Multibody System Dynamics*, **4**, 107-127, 2000.

[8] Intec GmbH, SIMPACK. http://www.simpack.de/, 2009.

[9] O. Vaculin, W. R. Kruger, and M. Valasek, Overview of Coupling of Multibody and Control Engineering Tools. *Vehicle System Dynamics*, **41**, 415-429, 2004.

[10] O. Oberschelp, and H. Vocking, Multirate Simulation of Mechatronic Systems. *LCM '04: Proceedings of the IEEE International Conference on Mechatronics 2004*, 404-409, 2004.

[11] S. S. Shome, E. J. Haug, and L. O. Jay, Dual-Rate Integration Using Partitioned Runge-Kutta Methods for Mechanical Systems With Interacting Subsystems. *Mechanics Based Design of Structures and Machines*, **32**, 253-282, 2004.

[12] INRIA, Scilab. http://www.scilab.org/, 2009.

[13] Wolfram Research, Mathematica. http://www.wolfram.com/, 2009.

[14] J. García de Jalón, and E. Bayo, *Kinematic and Dynamic Simulation of Multibody Systems - The Real-Time Challenge*. Springer-Verlag, New York 1994.

[15] J. Walter, and M. Kock, UBLAS. http://www.boost.org/libs/numeric/, 2006.

[16] NIST, Basic Linear Algebra Subprograms. http://www.netlib.org/blas/, 2006.

[17] NETLIB, LAPACK. http://www.netlib.org/lapack/, 2007.

[18] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Transactions on Mathematical Software*, **35**, 2008.

[19] T. A. Davis, and K. Stanley, KLU: a Clark Kent Sparse LU Factorization Algorithm for Circuit Matrices. http://www.cise.ufl.edu/~davis/techreports/KLU/pp04.pdf, 2004.

[20] A. Gupta, M. Joshi, and V. Kumar, WSSMP: A High-Performance Serial and Parallel Symmetric Sparse Linear Solver. *Applied Parallel Computing*, **1541**, 182-194, 1998.

[21] M. González, F. González, D. Dopico, and A. Luaces, On the Effect of Linear Algebra Implementations in Real-Time Multibody System Dynamics. *Computational Mechanics*, **41**, 607-615, 2008.

[22] F. González, A. Luaces, U. Lugrís, and M. González, Non-Intrusive Parallelization of Multibody System Dynamic Simulations. *Computational Mechanics*, **Online First**, DOI: 10.1007/s00466-009-0386-3. 2009.

[23] E. Bayo, and R. Ledesma, Augmented Lagrangian and Mass-Orthogonal Projection Methods for Constrained Multibody Dynamics. *Nonlinear Dynamics*, **9**, 113-130, 1996.

[24] J. Cuadrado, R. Gutierrez, M. A. Naya, and P. Morer, A Comparison in Terms of Accuracy and Efficiency Between a MBS Dynamic Formulation With Stress Analysis and a Non-Linear FEA Code. *International Journal for Numerical Methods in Engineering*, **51**, 1033-1052, 2001.

[25] J. Cuadrado, D. Dopico, M. González, and M. Naya, A Combined Penalty and Recursive Real-Time Formulation for Multibody Dynamics. *Journal of Mechanical Design*, **126**, 602-608, 2004.

[26] M. Arnold, Numerical Methods for Simulation in Applied Dynamics. M.Arnold and W.Schiehlen eds. *Simulation techniques for applied dynamics,* 191-246. Springer Wien New York, 2008.

[27] Busch, M., Arnold, M., Heckmann, A., and Dronka, S.: Interfacing SIMPACK to Modelica/Dymola for Multi-Domain Vehicle System Simulations, SIMPACK News 11, 1-3 (2007)

[28] National Instruments, MATRIXx/SystemBuild. http://www.ni.com/matrixx/what_is_matrixx.htm, 2009.

[29] INRIA, Scicos: Block Diagram Modeler/Simulator. http://www.scicos.org/, 2009.

[30] P. R. Crossley, and J. A. Cook, A Nonlinear Engine Model for Drivetrain System Development., 921-925, 1991

[31] K. S. Anderson, and J. H. Critchley, Improved 'Order-N' Performance Algorithm for the Simulation of Constrained Multi-Rigid-Body Dynamic Systems. *Multibody System Dynamics*, **9**, 185-212, 2003.

[32] M. González, D. Dopico, U. Lugrís, and J. Cuadrado, A Benchmarking System for MBS Simulation Software: Problem Standardization and Performance Measurement. *Multibody System Dynamics*, **16**, 179-190, 2006.