Multibody Syst Dyn (2010) 24: 237–253
DOI 10.1007/s11044-010-9199-6

Efficient coupling of multibody software with numerical computing environments and block diagram simulators

Francisco González · Manuel González · Aki Mikkola

Received: 18 December 2009 / Accepted: 12 March 2010 / Published online: 27 March 2010

Abstract Simulation of complex mechatronic systems like an automobile, involving mechanical components as well as actuators and active electronic control devices, can be accomplished by combining tools that deal with the simulation of the different subsystems. In this sense, it is often desirable to couple a multibody simulation software (for the mechanical simulation) with external numerical computing environments and block diagram simulators (for the modeling and simulation of nonmechanical components).

In this work, an in-house developed C++ MBS simulation software has been coupled with the commercial tools MATLAB and Simulink, and different coupling techniques have been identified, implemented, and tested in order to assess their computational performance. Two categories of coupling techniques have been investigated: those in which only one tool performs the integration (function evaluation) and those in which each tool uses its own integrator (cosimulation). Furthermore, the efficiency of the described coupling methods has been compared to that of equivalent monolithic models, and indications are provided to implement them in other simulation environments.

Results show that state-of-the-art coupling techniques can reduce simulation times in one or two orders of magnitude with respect to standard techniques. Finally, advices are provided to select the coupling method best suited to a particular application, as a function of its efficiency and implementation effort.

Keywords Multibody · Multiphysics · Cosimulation · Coupling · Efficiency · Block diagram simulators

F. González (✉) · M. González
Escuela Politécnica Superior, Universidad de A Coruña, Mendizábal s/n, 15403 Ferrol, Spain
e-mail: fgonzalez@udc.es

M. González
e-mail: lolo@cdf.udc.es

A. Mikkola
Department of Mechanical Engineering, Lappeenranta University of Technology, P.O. Box 20, 53851, Lappeenranta, Finland
e-mail: aki.mikkola@lut.fi

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 and the overall performance of the 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 account of different phenomena simultaneously when simulating a system. When evaluating the behavior 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 handle 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 modeling. Uniform or monolithic modeling is based on representing all the subsystems of a multidomain 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 modeling 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 modeling each subdomain. 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 settings for the simulation of each subsystem, such as the integration time-step, the numeric solver, and other particularized details. Furthermore, 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 how the integration is performed. The name *cosimulation* 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 evaluation* will be used.

Commercial multibody packages have been incorporating multiphysics 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 noncommercial 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 overlooked. In particular, there is a lack of information about the amount of effort the implementation of a coupling 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 coupling 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 coupling 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 cosimulation.

In this work, coupling techniques with external simulation tools have been used for widening the capabilities of existing MBS software, through the addition of functionality with numerical computation environments (such as MATLAB, Scilab [12], Octave [13], Mathematica [14], or MATRIXx [15]) and block diagram simulators (Simulink, Scicos [12], or SystemBuild [15]). To this end, coupling possibilities between the above-mentioned 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 coupling 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 cosimulation 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 cosimulation, with different synchronization methods, between simulation tools.

This paper is organized as follows: Sect. 2 gives a general review of the existing techniques for communicating a multibody package to external simulation tools. In Sects. 3 and 4, these techniques are implemented in an MBS software tool and a commercial package for numerical computations and block diagram simulation. Introduced computational strategies are utilized in the dynamic simulation of two example problems. Finally, the conclusions of the work are summarized in Sect. 5.

2 Coupling techniques

The expansion of a multibody software tool via communication with external simulation packages can be performed in several ways, which can be categorized as data files exchange, function evaluation and cosimulation approaches.

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. In most conventional MBS applications, the computational cost of read/write operations is high, compared to the computational effort the numerical integration requires. There are exceptions to this rule, when any of the coupled software tools requires a very high computational effort to obtain its results (as it is the case of CFD packages); in these cases, the communication overhead due to the reading and writing of files can become less representative. However, in many MBS applications, for example, in Human/Hardware-in-the-Loop (HiL) settings, the read and write operations on data files would slow down the execution of the code and, therefore, this technique should not be applied in repetitive calls 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. 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 are *function evaluations* from one simulation tool to another. In this work, the name function evaluation is reserved for those communications in which only one of the software tools is actually performing a numerical integration, while the other one returns values on request, from the states passed by the integrator tool. It must be noted that *function evaluations* cannot be classified as cosimulation, as the integration of the equations of the motion is performed by a single integrator; however, this technique can be used to expand the range of phenomena the original simulation code is able to deal with. This configuration can be achieved through code exporting (via joint compiling, together with the integrator tool, or precompiled as a library) or by direct communication between processes. Application fields of the function evaluation strategy would be complex force evaluations during runtime, table look-up, and other processes in which numerical integration is not present.

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 Fig. 1. 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 *cosimulation* 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 [16]. 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

$$\begin{cases} \dot{\mathbf{x}}_m(t) = \mathbf{f}_m\big(\mathbf{x}_m(t), \mathbf{u}_m(t)\big), \\ \mathbf{y}_m(t) = \mathbf{g}_m\big(\mathbf{x}_m(t)\big), \end{cases} \tag{1}$$

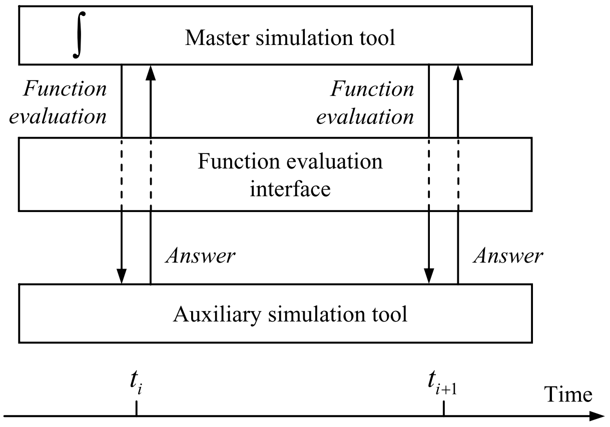


Fig. 1 Generic function evaluation configuration

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 analogue expression can be used for the external simulation tool

$$\begin{cases} \dot{\mathbf{x}}_e(t) = \mathbf{f}_e\big(\mathbf{x}_e(t), \mathbf{u}_e(t)\big), \\ \mathbf{y}_e(t) = \mathbf{g}_e\big(\mathbf{x}_e(t)\big), \end{cases} \tag{2}$$

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

$$\begin{cases} \mathbf{u}_e(t) = \mathbf{y}_m(t), \\ \mathbf{u}_m(t) = \mathbf{y}_e(t). \end{cases} \tag{3}$$

Nowadays, state-of-the-art commercial software performs cosimulation at constant time-steps, with the same external integration time-steps in every subsystem, although research is being carried out to introduce multirate methods in cosimulation environments [17]. Even with constant and equal time-steps in each subsystem, the evaluation of the inputs for each subsystem, given by (3), at synchronization point t_i can be performed in several ways. A frequent strategy is assuming that the inputs of each subsystem can be considered constant during each time-step $[t_i, t_{i+1}]$, which leads to

$$\begin{cases} \mathbf{u}_e(t) = \mathbf{u}_e(t_i) = \mathbf{y}_m(t_i), \\ \mathbf{u}_m(t) = \mathbf{u}_m(t_i) = \mathbf{y}_e(t_i). \end{cases} \tag{4}$$

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 cosimulation, in which cosimulated states are exchanged once in each integration step, and then each subsystem proceeds its own integration independently, has been used.

The use of cosimulation techniques can give rise to stability issues, which are out of the scope of this paper, as the way in which these issues must be dealt with depends on several factors (such as the inner algorithms of the numerical integrators used in each subsystem and the extrapolation techniques of the interface) that are not directly related to the nature of the coupling technique used to communicate the simulation tools.

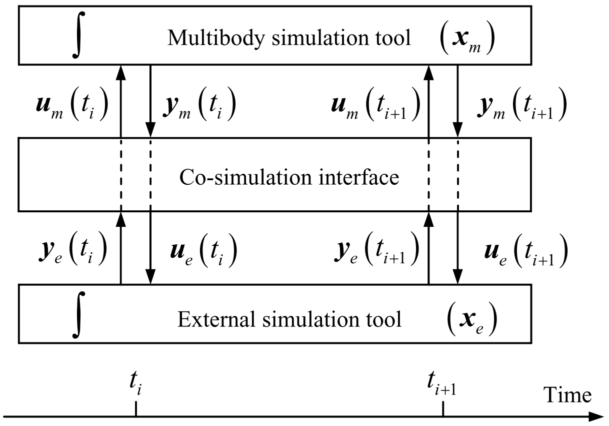

Multibody simulation tool $(\boldsymbol{x}_m)$
$\boldsymbol{u}_m(t_i)$
$\boldsymbol{y}_m(t_i)$
$\boldsymbol{u}_m(t_{i+1})$
$\boldsymbol{y}_m(t_{i+1})$
Co-simulation interface
$\boldsymbol{y}_e(t_i)$
$\boldsymbol{u}_e(t_i)$
$\boldsymbol{y}_e(t_{i+1})$
$\boldsymbol{u}_e(t_{i+1})$
External simulation tool $(\boldsymbol{x}_e)$
t_i
t_{i+1}
Time

Fig. 2 Generic cosimulation configuration

As it was the case in the function evaluation strategy, cosimulation can be implemented on the basis of intercommunication between processes, or through code export. Again, translation routines between data storage formats will likely be necessary. The synchronization of integrators and the exchange of data can be managed by a cosimulation interface, which can be implemented in one of the communicating software tools. A scheme of this composition is shown in Fig. 2.

2.1 Software environment

In order to test the coupling techniques described in this section, a coupled simulation environment composed of an in-house developed MBS simulation tool and the commercial numerical package MATLAB, with its block diagram add-on Simulink [6] has been used. Despite its wide acceptance among the multibody community, 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 fact rules out the software for demanding applications, for example real-time simulation. On the other hand, communication between MBS software and MATLAB/Simulink programs, representing control loops, actuators, and other external components, can provide an additional functionality that is missing in many multibody software packages.

The MBS simulation software used in this research has been developed by the Laboratory of Mechanical Engineering of the University of A Coruña during the last 4 years, and features optimized implementations of the formalisms used for the dynamic simulation of mechanical systems [18, 19]. The test problems in this work are described using planar natural coordinates; the equations of motion of the multibody system are expressed using the index-3 augmented Lagrangian formulation, as explained in [20], and the nonlinear equations of motion are integrated using the trapezoidal rule. This combination has shown favorable performance and robustness features in previous works [21, 22].

The techniques described in the following sections can be applied to other software tools different from MATLAB/Simulink, for example, Scilab/Scicos [12] or MATRIXx/SystemBuild [15]. In general, communication between processes can often be achieved if the software supports the use of inter-process communication (IPC), like sockets. The use of external code can be performed through calls to dynamically linked libraries, with their corresponding import libraries and header files, if necessary.

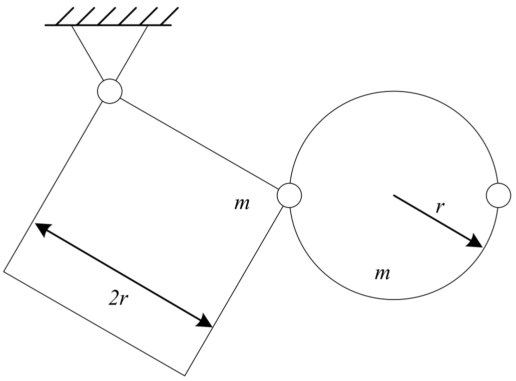


Fig. 3 Double pendulum

3 Function evaluation

A 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 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 research, three implementation approaches for the function evaluation method have been tested: MATLAB Engine, MATLAB Compiler, and a MEX API of functions. These approaches can be modeled in different math tools as they represent three strategies that could be referred to as interprocess communication through an engine, precompilation of interpreted code, and direct call to C routines from the math simulation tool, respectively. Thus, the methods for implementing the function evaluation described in this section can be applied to similar numerical software, different from the one that has been used in this work, making use of alternative communication facilities. For example, Scilab provides the *intersci* program, which allows calling C and Fortran routines from Scilab, and the calling routines defined in *call_scilab.h*, which make Scilab work as a calculus engine. Code precompilation is a common feature in some interpreted languages such as Java and Python.

A dynamic simulation of a double-pendulum has been selected as test example for the above-mentioned implementation approaches: the multibody software carries out the numerical integration and the math software tool is used to evaluate the equations of motion at each time-step. This simple example has been chosen, as there is no practical increase of complexity derived from applying the function evaluation technique to more involved problems.

The double pendulum is shown in Fig. 3. 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 $\mathbf{M}$, the constraints vector $\mathbf{\Phi}$, the Jacobian matrix of the constraints vector $\mathbf{\Phi_q}$ and the generalized forces vector $\mathbf{Q}$, is written in *.m* files and it is accessed from the MBS simulation software through function evaluation 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 written in MATLAB with already tested problems, avoiding the need for translating them to C++, and

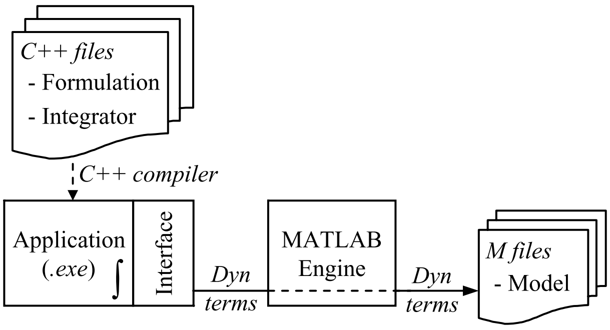


Fig. 4 Function evaluation configuration with MATLAB Engine

to invoke specific MATLAB functionality such as involved matrix operations or complex function evaluations.

3.1 MATLAB Engine

The MATLAB 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 have 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 should be noted that the engine receives its commands as a string of characters which must be parsed, resulting in the deceleration of the execution of the code.

The function evaluation configuration through the engine is represented in Fig. 4. The MBS software acts as a master 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.

3.2 MATLAB Compiler

Function evaluation has also been achieved through code export, 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, thus 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 compiled MATLAB code must be written. The C/C++ library generated by the compiler only contains wrappers for the MATLAB routines, and hence it still depends on MATLAB to carry out the computations on runtime.

The use of the compiler on the *.m* files removes the need for the use of the engine, as shown in Fig. 5, replacing the process communication with the export of the precompiled 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.

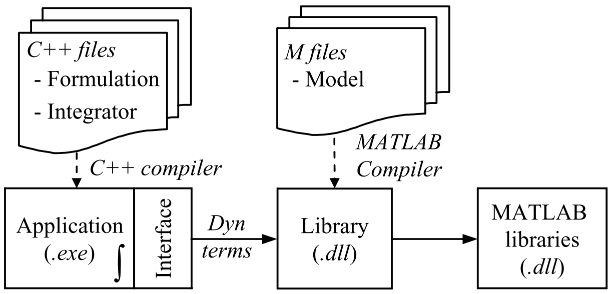


Fig. 5 Function evaluation configuration with MATLAB Compiler

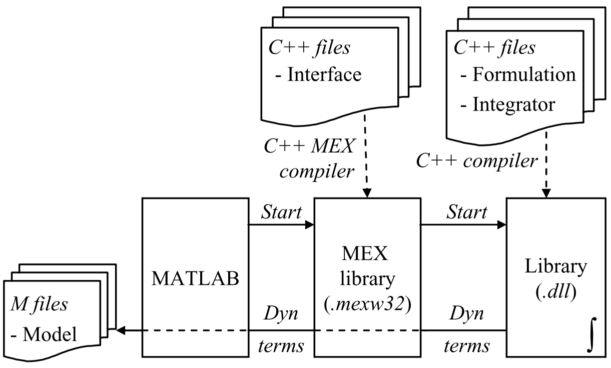


Fig. 6 Function evaluation configuration with MEX API of functions

3.3 MEX functions

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, the mathematical package 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 the C++ program uses and vice versa.

Figure 6 shows the layout of the function evaluation 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.

3.4 Results

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 configured to use LAPACK routines *gtrf* and *gtrs* as linear solver, which have been proved to be efficient for small-size problems ([18]) and allow an easy conversion of data from MATLAB storage format.

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

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

Function evaluation method	$\Delta t = 10^{-3}$ s		$\Delta t = 10^{-2}$ s	
	Elapsed time (s)	Ratio	Elapsed time (s)	Ratio
Standalone MBS code	$5.02 \cdot 10^{-2}$	1	$8.40 \cdot 10^{-3}$	1
MATLAB Engine	18.12	361.0	3.32	395.2
MATLAB Compiler	5.56	110.8	1.07	127.4
MEX API of functions	0.64	12.7	0.12	14.3
Number of solver iterations	10,000		1,840	

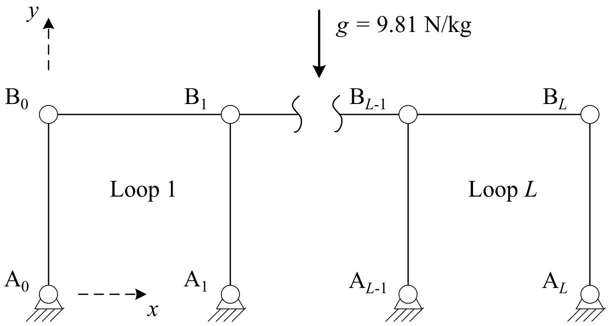
identical for each time-step, independently of the method used for providing the dynamic terms. The ratios defined in the table refer to the elapsed time of the correspondent function evaluation implementation when compared to standalone C++ MBS code (without the use of MATLAB). The number of iterations is the number of times the iterative solution of the system, required by the implicit integrator used, 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 evaluation methods slows down the execution of the code, makes negligible the amount of computational time elapsed out of the iterative loop. In the standalone 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 function evaluations in external simulation tools slows down the execution of the program. The engine approach is very easy to implement, but it also delivers very poor efficiency: it has been estimated that the parsing of a single empty function evaluation takes 0.25 ms. Therefore, the use of MATLAB Engine should be discouraged when function evaluations in the auxiliary tool are repetitive (for example, several times in each integration step).

MATLAB Compiler is usually claimed to be the fastest coupling technique, since it removes the need of parsing string instructions as function calls are performed directly on routines stored in dynamically linked libraries. Even so, the generated C code is still two orders of magnitude slower than standalone C++ MBS code. The overhead of the MATLAB Compiler approach comes from the need of converting data structures between the MBS software and MATLAB routines. This approach has an additional drawback: if the M code is modified, it must be compiled and linked again, and this process slows down the code development.

The implementation of the function evaluations as MEX API of functions, shown in Fig. 6, 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 an 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. Another advantage of the MEX API of functions is that the MATLAB code stays in *.m* files and, therefore, it allows fast development iterations because it can be modified and tested again without going through a compilation and linking process (as in the case of the previous approach based on MATLAB Compiler).

Fig. 7 L-loop fourbar linkage



4 Cosimulation

Under the cosimulation approach, the MBS simulation tool has been connected to MATLAB's 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.

In order to test the cosimulation, a multiphysics model composed by an engine and a mechanical system is simulated. Each subsystem is modeled 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 [23]. 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 Fig. 7 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 [24, 25]. 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.

The engine provides a motor torque to the linkage through a gearbox, which is also modeled 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 predefined angle-law. The resulting Simulink model can be seen in Fig. 8. The use of memory blocks is motivated by the need of avoiding algebraic loops.

In this research, three implementation approaches for the co-simulation have been tested: network connection, *Simulink as master* and *MBS software as master*. These approaches can be used in other block diagram simulators, as the *Simulink as master* configuration is equivalent as using the MBS code as a library, and the *MBS as master* configuration represents the opposite approach, where the block diagram model becomes the compiled component.

4.1 Network connection

Data interchange between two processes running simultaneously can be carried out using a TCP/IP network connection through standard sockets. To this end, the MBS software is

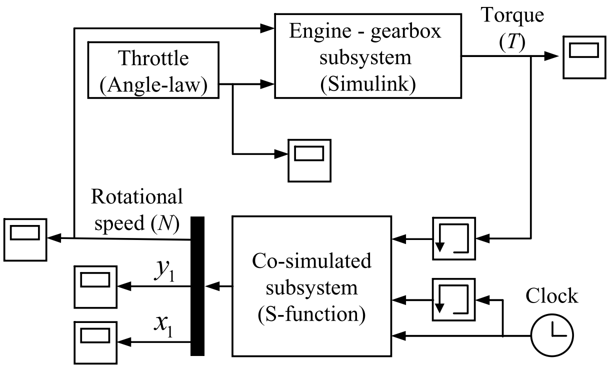

Engine - gearbox subsystem (Simulink)
Torque (T)
Throttle (Angle-law)
Rotational speed (N)
y_1
x_1
Co-simulated subsystem (S-function)
Clock

Fig. 8 Simplified Simulink model for cosimulation, implemented with an *S-function*

modified in order to make it work 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 [15] and the *C* or *Fortran* block in Scicos [26]. Thus, the interface is split into two parts, one in the block-diagram environment and one in the MBS software. Both parts of the interface are responsible for the translation of the storage formats and for the adequate interchange of information at each time-step, so they must be correctly coordinated. Moreover, the communication sequence between both subsystems has to be separately coded in each environment, adding an extra burden to the task of keeping the synchronization of the integrators.

4.2 Simulink as master

An alternative to network communication is the code export approach, in which the MBS code can be compiled as a dynamically linked library (*.dll* or *.so*) and directly called from an *S-function* block inside Simulink. In this case, the *S-function* includes all the code of the interface between the MBS code and the Simulink model, and must manage the required exchange of data and format conversions between both environments. In this configuration, Simulink becomes the driver software, starting and ending the simulation and calling the MBS software through the interface block each time a return value is needed by the Simulink integrator.

4.3 MBS software as master

Another possibility, following the opposite approach to that used in the previous subsection, is using the MATLAB product Real-Time Workshop (RTW) [6] to translate the Simulink model into fast standalone C code, which can be called from the MBS code. In this case, the Simulink model is converted into a dynamically linked library (*.dll*) through the use of RTW functionality; this can be done with little modifications to the Simulink model used in the previous subsection. With this configuration, the MBS code starts and manages the co-simulation process, invoking the functions compiled in the *.dll* in each integration time-step in order to obtain the value of the torque the engine supplies to the linkage.

This approach has a drawback when compared with the previous techniques: if the Simulink model is modified, it must be translated into C code and compiled again; this is a complex and delicate procedure, which slows down the iterations in code development.

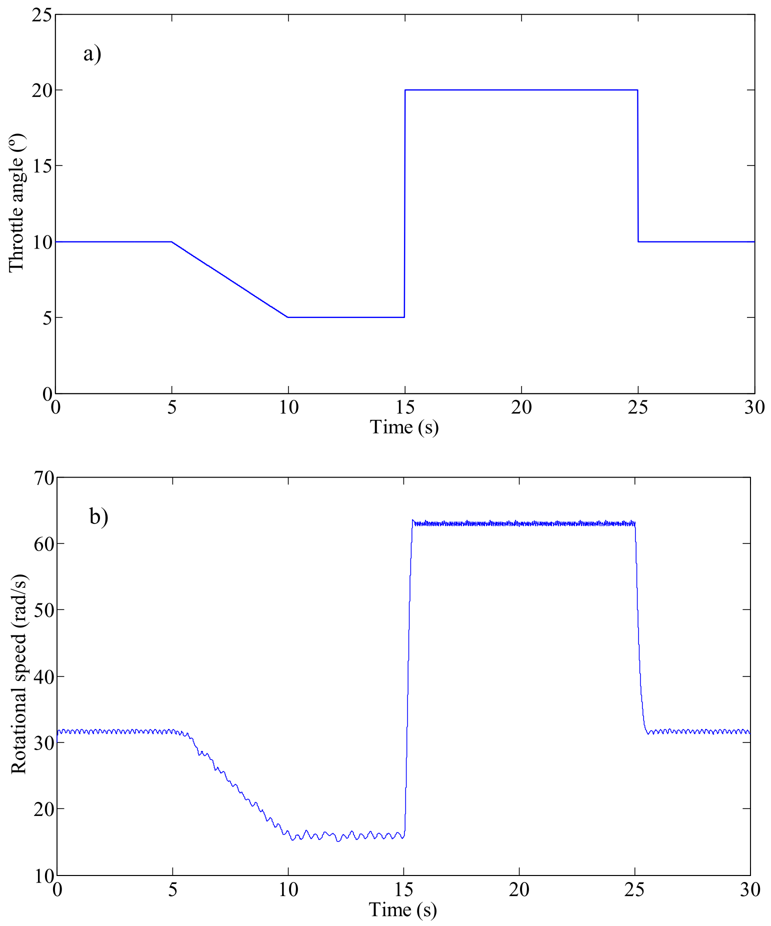


Fig. 9 Throttle angle (**a**) and rotational speed of the mechanism (**b**) for a 30 s simulation of a 1-loop linkage

In both cases (Simulink as master and MBS software as master) two integrators are acting simultaneously and, for this reason, a careful coordination between them is required. Simulink behavior is, in many aspects, beyond the control of the user, so the cosimulation interface has to be specifically defined to fit Simulink.

4.4 Results

The simulation time in the previously described test example is 30 s. A penalty factor of $\alpha = 10^{10}$ and a constant integration time-step of 10^{-3} s have been used. Direct cosimulation with the same integration time-step size 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). The MBS software is configured to use KLU [27] routines for solving the linear systems the simulation requires. The cosimulation coupling has been implemented with the three different approaches described above.

Results of the simulation can be seen in Fig. 9, 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,

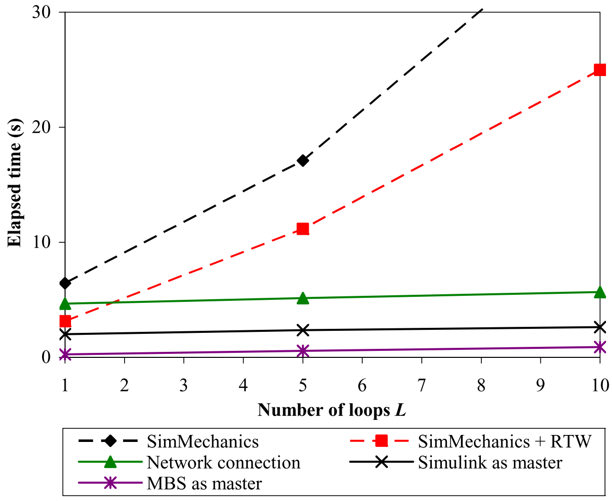


Fig. 10 Elapsed times for a 30 s simulation of the L-loop linkage powered by the engine, with different simulation techniques

depicted below, shows that the linkage follows the input given by the pedal angle, with the limitations imposed by its rotational inertia and damping. Results do not show significant variations between the three tested coupling techniques.

The performance of the described techniques has been tested against a model of the whole system (engine and fourbar linkage) entirely built in Simulink. The fourbar linkage has been modeled with the SimMechanics library [6], a Simulink add-on for modeling and simulation of rigid multibody systems. In a second stage, the computational efficiency of this model has been further improved via the RTW package, translating the whole model into a standalone C executable. Simulink *ode1* integrator has been used in these simulations, since it is the fastest available integrator and it provides enough precision for the test problem. A comparison of the elapsed times for a 30 s simulation can be seen in Fig. 10. The monolithic approaches are represented with dashed lines, as they are not properly cosimulation, and labeled as *SimMechanics* for the pure Simulink model, and *SimMechanics* + *RTW* for the model translated into C code via RTW. The cosimulation methods are designed as *Simulink as master*, for the implementation where Simulink calls MBS code compiled as a library; *Network connection*, when the communication is performed via sockets between simultaneously running processes and *MBS as master*, when the MBS code calls Simulink routines from the *.dll* library compiled with RTW.

Results are summarized in Table 2, where the ratios of elapsed time with respect to the fastest method are also included. They 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 using a very simple integrator as *ode1*. The use of RTW mitigates this problem and reduces the calculation time between 30% and 50%. However, the use of cosimulation techniques leads to even 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 *Simulink as master* implementation is somewhat faster than the *Network connection* method, as the overhead derived from socket communications is not present. The *MBS as master* yields the best results, as it was expected, because the intercommunication takes place, in this case, between an executable and a library both of them coded in an efficient language (C/C++).

It should be noted, however, that the *MBS as master* implementation is significantly more complex than the *Network connection* or *Simulink as master* implementations, and it

Table 2 Elapsed times in a 30 s dynamic simulation of an L-loop fourbar linkage powered by the engine. N stands for the number of variables of the mechanical system

Cosimulation method	$L = 5$ ($N = 13$)		$L = 10$ ($N = 23$)	
	Elapsed time (s)	Ratio	Elapsed time (s)	Ratio
MBS software as master	0.58	1	0.90	1
Simulink as master	2.37	4.1	2.62	2.9
Network connection	5.15	8.9	5.67	6.3
SimMechanics + RTW	11.17	19.3	24.97	27.7
SimMechanics	17.11	29.5	38.88	43.2

forces to follow a complex translation process every time the Simulink model is modified, as explained in Sect. 4.3. For these reasons, the *Network connection* or *Simulink as master* cosimulation approaches are better suited for developing and fine-tuning Simulink models, while the *MBS as master* cosimulation approach is appropriate for production code and real-time applications.

Trends indicate that cosimulation will achieve greater differences with respect to models fully implemented in Simulink as the number of variables of the problem increases. In fact, real-time simulation (less than 30 s of computations) has been achieved with the described cosimulation techniques for multibody models up to 300 variables. This upper limit would allow the efficient real-time simulation of many industrial, nonacademic multidisciplinary systems.

5 Conclusions

In this study, several implementation methods for coupling MBS simulation software with block diagram simulators and numerical computing environments have been tested. The methods have been tested in a software environment where a C/C++ MBS code is coupled with MATLAB/Simulink, a quite common setup in the modeling and simulation of complex mechatronic systems. The investigated coupling techniques have been divided in two categories: *function evaluation* and *cosimulation*.

Regarding the implementation methods for function evaluation in MATLAB, the following conclusions can be established:

- The MATLAB Engine approach is the easiest to implement but also the slowest one. The use of MATLAB Compiler has reduced the simulation times to 30% of the time consumed by MATLAB Engine in the simulated example, but at the cost of slowing down the code development iterations. Both approaches have been two orders or magnitude slower than standalone MBS code.
- The MEX API of functions is the fastest approach, being only one order of magnitude slower than standalone MBS code. The implementation effort is higher that in other methods, but not overwhelming and, therefore, it is recommended as the best approach for function evaluation when simulation efficiency is needed.

Regarding the implementation methods for cosimulation with Simulink, the following conclusions can be established:

- Cosimulation methods are approximately one order of magnitude faster than simulations based on monolithic models developed in Simulink, even if tools like Real-Time Workshop are used.
- The method labeled *Simulink as master* provides the best trade off between simulation efficiency and ease of implementation and code development and, therefore, it is recommended for developing and fine-tuning models for cosimulation setups.
- The method labeled *MBS software as master* is the fastest approach (several times faster than *Simulink as master*, depending on the relative complexity between the block diagram and the multibody models), but its implementation is more complex and requires the translation of Simulink models into C code, a step that slows down the development iterations. Therefore, this method is recommended for production code and real-time applications.

The described coupling techniques can be also implemented which minor changes in other numerical computing environments and block diagram simulators different from MATLAB/Simulink, for example, SystemBuild or Scilab/Scicos. However, the efficiency of the different tested methods highly depends on the internal data structures and algorithms of software and, therefore, their relative efficiency would be different.

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

References

1. Valasek, M.: Modelling, simulation and control of mechatronical systems. In: Arnold, M., Schiehlen, W. (eds.) Simulation Techniques for Applied Dynamics. CISM Courses and Lectures, pp. 75–140. Springer, Wien (2008)
2. Samin, J.C., Bruls, O., Collard, J.F., Sass, L., Fisette, P.: Multiphysics modeling and optimization of mechatronic multibody systems. Multibody Syst. Dyn. **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. Kubler, R., Schiehlen, W.: Modular simulation in multibody system dynamics. Multibody Syst. Dyn. **4**, 107–127 (2000)
8. SIMPACK AG: SIMPACK. http://www.simpack.com (2009)
9. Vaculin, O., Kruger, W.R., Valasek, M.: Overview of coupling of multibody and control engineering tools. Veh. Syst. Dyn. **41**, 415–429 (2004)
10. Oberschelp, O., Vocking, H.: Multirate simulation of mechatronic systems. In: LCM '04: Proceedings of the IEEE International Conference on Mechatronics 2004, pp. 404–409 (2004)
11. Shome, S.S., Haug, E.J., Jay, L.O.: Dual-rate integration using partitioned Runge–Kutta methods for mechanical systems with interacting subsystems. Mech. Based Des. Struct. Mach. **32**, 253–282 (2004)
12. INRIA: Scilab. http://www.scilab.org/ (2009)
13. Eaton, J.W.: Octave. http://www.gnu.org/software/octave/ (2010)
14. Wolfram Research: Mathematica. http://www.wolfram.com/ (2009)
15. National Instruments: MATRIXx/SystemBuild. http://www.ni.com/matrixx/what_is_matrixx.htm (2009)
16. Arnold, M.: Numerical methods for simulation in applied dynamics. In: Arnold, M., Schiehlen, W. (eds.) Simulation Techniques for Applied Dynamics. CISM Courses and Lectures, pp. 191–246. Springer, Wien (2008)
17. Busch, M., Arnold, M., Heckmann, A., Dronka, S.: Interfacing SIMPACK to Modelica/Dymola for multi-domain vehicle system simulations. SIMPACK News **11**, 1–3 (2007)
18. González, M., González, F., Dopico, D., Luaces, A.: On the effect of Linear algebra implementations in real-time multibody system dynamics. Comput. Mech. **41**, 607–615 (2008)
19. González, F., Luaces, A., Lugrís, U., González, M.: Non-intrusive parallelization of multibody system dynamic simulations. Comput. Mech. **44**, 493–504 (2009)

20. Cuadrado, J., Cardenal, J., Morer, P., Bayo, E.: Intelligent simulation of multibody dynamics: Space-state and descriptor methods in sequential and parallel computing environments. Multibody Syst. Dyn. **4**, 55–73 (2000)
21. Cuadrado, J., Gutierrez, R., Naya, M.A., Morer, P.: A comparison in terms of accuracy and efficiency between a MBS dynamic formulation with stress analysis and a non-linear FEA code. Int. J. Numer. Methods Eng. **51**, 1033–1052 (2001)
22. Cuadrado, J., Dopico, D., González, M., Naya, M.: A combined penalty and recursive real-time formulation for multibody dynamics. J. Mech. Des. **126**, 602–608 (2004)
23. Crossley, P.R., Cook, J.A.: A nonlinear engine model for drivetrain system development. In: International Conference on Control (Control 91), vol. 2, pp. 921–925, Edinburg (1991)
24. Anderson, K.S., Critchley, J.H.: Improved 'order-N' performance algorithm for the simulation of constrained multi-rigid-body dynamic systems. Multibody Syst. Dyn. **9**, 185–212 (2003)
25. González, M., Dopico, D., Lugrís, U., Cuadrado, J.: A benchmarking system for MBS simulation software: Problem standardization and performance measurement. Multibody Syst. Dyn. **16**, 179–190 (2006)
26. INRIA: Scicos: Block Diagram Modeler/Simulator. http://www.scicos.org/ (2009)
27. Davis, T.A., Stanley, K.: KLU: A Clark Kent sparse LU factorization algorithm for circuit matrices. http://www.cise.ufl.edu/~davis/techreports/KLU/pp04.pdf (2004)