A contribution for virtual prototyping of mechatronic systems based on real-time distributed high level architecture

Hassen Hadj-Amor, Thierry Soriano

A Contribution for Virtual Prototyping of Mechatronic Systems Based on Real-Time Distributed High Level Architecture

H. J. Hadj-Amor
e-mail: hassen.hadj-amor@supmeca.fr

T. Soriano
e-mail: thierry.soriano@supmeca.fr

LISMMA—Supmeca Toulon,
Maison des Technologies, Place Georges,
Pompidou, 83000 Toulon, France

Mechatronics is the integration of different sciences and techniques of mechanical engineering, automatic control, electronics, and informatics. The rapid evolution of the market competitors requires the reduction of development time of a product while increasing the quality and performance. It is, therefore, necessary to increase the efficiency of the design process. To meet this need, simulation and, especially, virtual prototyping have become a key technology. It is difficult to find simulation tools are able to analyze multidependent systems of different areas. However, an environment that allows a simulation integrating multidisciplinary mechatronic systems is necessary. This paper describes a method of design and simulation of mechatronic systems. First, we identify the behavior model and its associated 3D geometric model. The behavior model is seen as a dynamic hybrid system of two coupled hybrid automata (operative part and control part). Then, we present OpenMASK and OpenModelica simulators, the IEEE1516 standard HLA and work related to this distributed architecture for simulation. In a top-down approach, we present our method and experiments to integrate HLA functionalities in these simulators and to distribute the modeling elements of mechatronic systems. Also, we propose extensions to integrate real-time for interactive simulations. Finally, we apply our approach on a representative example of a mechatronic system.
[DOI: 10.1115/1.3647868]

1 Introduction

A mechatronic system is the combination of several components from different domains (mechanics, automatic control, electronics, and embedded control software). This combination makes possible the generation of small and powerful systems which integrate functions and ability for decisions. However, their design can be difficult and the development period of such complex systems has to be as short as possible. Especially, analyzing the behavior is a difficult task. To face this problem, virtual prototyping of mechatronic systems can be a good solution. Indeed, virtual prototypes can help manufacturers to predict behavior so they can make better design, manufacturing, and business decisions. Virtual prototyping involves a 3D geometric simulation and MULTIPHYSICS simulation. A first solution consists in coupling subsystem models within the same environment. This is cosimulation as described in Ref. [1]. Cosimulation is one of the possible techniques to enable closer interaction between existing submodels into a more complete model. The most common environment that sup-

plies as a cosimulation framework is SIMULINK. Many environments that support MULTIPHYSICS simulations, such as DYMOLA [2] or MULTIBODY, and dynamic systems, such as MSC.ADAMS [3], provide specific modules for running such cosimulations. Many methods are used for coupling of dynamic simulations. One method earlier used is transmission lines modeling (TLM). The TLM is based on physically motivated time delays and TLM elements to separate the components in time and enable efficient cosimulation. This technique was implemented for coupling different subsystems [4–6]. However, these tools are not specialized in the 3D animation. To keep the accuracy and precision of 3D graphic simulation, we are going to use two specialized environments of simulation. Our contribution is situated in the definition and the implementation of normalized exchanges and synchronization between two highly efficient open source simulators, one of them specialized in virtual reality. The goal here is to simulate a complete environment with many different actors and tools. The main concerns are not simulation methods but instead standards and protocols that allow tools to communicate.

In this paper, we present a method based on the high level architecture (HLA) to make communication possible between the 3D OpenMASK (Modular Animation and Simulation Kit) virtual prototyping simulator [7] and the MULTIPHYSICS OpenModelica [8] simulator. HLA is situated in continuation of the works on cosimulation with respect to disclosing proprietary information about the subsystem models by introducing a standardized layer to exchange data without requiring an integrator environment [1]. Using this method, we can facilitate simulating and analyzing mechatronic systems in a multidisciplinary workgroup. An approach for modeling and simulation of mechatronic systems is described in Sec. 2. Section 3 presents the HLA and some related works. Sections 4 and 5 present briefly OpenMASK and OpenModelica simulators. Section 6 presents solutions to integrate HLA services in both simulators. Section 7 deals with time management. Then, we present an application to test out our approach in Sec. 8. Lastly, we present a brief discussion in Sec. 9 to improve, in the future, the accuracy and efficiency of the framework environment obtained.

2 Modeling and Simulation for Mechatronics in a Distributed Architecture

From the functional point of view, a mechatronic system can be seen as a collaboration of two major parts:

— A control part: It consists of electronic, automatic, and computer technologies. It can be a closed-loop or open-loop control.
— An operative part: It consists of mechanical and electromechanical parts.

The distinction of the control and operative parts does not only mean to separate them and understand them independently but also to understand the links between them. The complexity of mechatronic systems is due to the integration of several disciplines. Accordingly, the behavior of these systems with various technologies is difficult to model. The control of a mechatronic system often uses discrete states, and the behavior of the operative part consists of discrete jumps between continuous states. This type of behavior is typically that of hybrid dynamic systems. We choose the hybrid automata formalism for a unique and homogeneous modeling of the operative part and the control part. Behaviors of the control part and the operative part are modeled by different hybrid automata, which can be coupled jointly following the method provided in Ref. [9]. The simulation of a mechatronic system, thus, has to support, on the one hand, the dynamic evolution represented by hybrid automata carrying in particular electromechanical equations and, on the other hand, its graphic representation. The global behavior is then modeled by hybrid automata which are implemented in OpenModelica using the
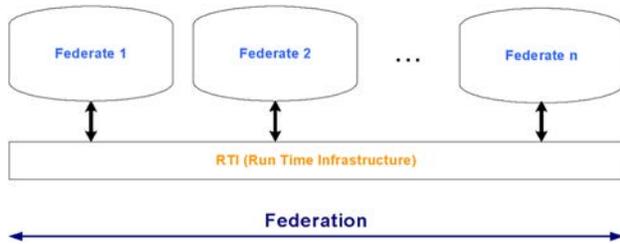
Fig. 1 A HLA federation

Fig. 2 Global approach

HybridAutomataLib [10]. On the other hand, the role of Open-MASK is to animate 3D physical objects representing the system. The communication between both simulators is maintained by the HLA RTI (real-time infrastructure).

In the HLA terminology, we mean by the term "federate," each elementary simulator, and by the term "federation" a group of intereffective federates. Notions of federate and RTI will be developed in Sec. 3.1.

We identified two federates: one federate for the animation of 3D objects and one behavioral federate, which represents the control and operative parts of the mechatronic system (Fig. 1). The first federate is implemented within OpenModelica; the second one is implemented within the OpenMASK virtual environment (Fig. 2).

3 HLA

3.1 Definitions and Terminology. HLA is defined under the IEEE standard 1516 for the architecture of interoperable distributed simulations. The HLA was proposed by the Defense Modeling and Simulation Office, instigated by Department of Defence [11–13].

In France, the ministry of defense and, in particular, the navy ministry is also interested in this standard.

This architecture has got three objectives:

— making the reuse of elementary simulator easier
— making interoperability between the distributed simulators easier
— reducing simulation and modeling costs.

The HLA is described by some specifications composed of:

— a series of rules that specifies the responsibilities of the federates and the federation
— an object model template
— specifications concerning the application programming interface (API).

The software elements of an HLA federation are composed of an implementation of the RTI and some federates. A federation is a set of federates having a common object model. It is the representation of a set of interoperating simulators. The first function of the federation object model is to specify, in a common and standard format, the nature of the data exchanged among federates in the federation. These data include an enumeration of all objects and interaction classes as well as a specification of the attributes and parameters characterizing theses classes. A simulation object model is a specification of the types of information that an individual federate could provide to HLA federation as well as the information that an individual federate can receive from other federates in HLA federation.

HLA is a specification and not an implementation. There are several softwares relating to HLA; RTI is the tool for the design of federation. We were interested in the open source RTIs. We chose the CERTI tool developed at ONERA [14,15].
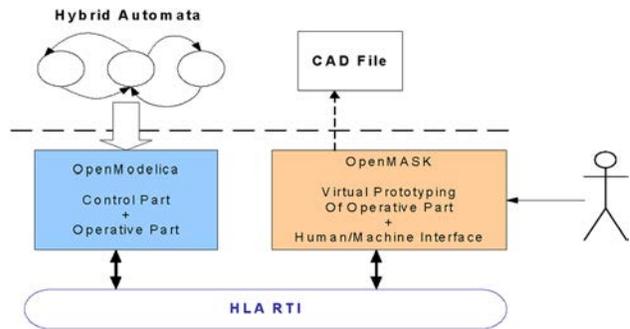
3.2 Related Works. HLA was developed mainly for military distributed simulations. There are few applications to integrate HLA in commercial and academic simulators. Difficulties arise when trying to make these simulators HLA compliant. Military simulators have been developed as specific tools where HLA functions may be implemented at the beginning of the development. However, HLA functions are not planned to integrate non-military simulators. Four solutions have been identified to integrate HLA in simulators in Ref. [16].

In the field of manufacturing systems, we find in Ref. [17] an environment based on HLA to integrate some simulators. In the field of control systems, the hybrid simulator AnyLogic integrated HLA services [17]. Many approaches were proposed to integrate HLA with MATLAB. Most of these approaches use wrapper programs and external libraries to communicate MATLAB with the RTI. The implementation of this interface was presented in Ref. [18]. The project is called the HLA TOOLBOX. In the field of 3D CAD software, we found an approach proposed in Ref. [18] to make a 3D simulator HLA compatible. In the field of virtual prototyping, we found an approach presented in Ref. [19] to link the oRis platform with the RTI. We did not find any application involving a 3D simulator and a MULTIPHYSICS simulator and integrating real-time aspects.

4 OpenMASK

OpenMASK is an open source platform for modular applications development and execution in animation, simulation, and virtual reality fields. OpenMASK has modular architecture [20]. A simulation with OpenMASK is composed of modules connected to a data bus. Each module can represent an elementary 3D object or can be a pure behavioral object. The way to program the behavior of an object is completely left to the user. An Open-MASK module can also contain inputs/outputs for the exchange of data. OpenMASK modules can exchange events directly or by means of the OpenMASK's bus.

5 OpenModelica

Modelica is an object-oriented language. Its goal is to model complex physical systems including electric, mechanical, hydraulic, and thermal components [21]. OpenModelica is a free environment of modeling, compilation, and simulation, based on a BSD (Berkeley Software Distribution) license. The current version of the OpenModelica environment allows the interactive execution of most of the expressions, the algorithms and the parts of Modelica functions as well as the generation of an effective C code from the models of equations and the Modelica functions. The C code generated is combined with a library of useful functions, a run-time library and a digital solver differential algebraic equations. The default integration method for OpenModelica is the DASSL code as defined by Brenan et al. [22]. In order to integrate event handling in the compiler and run-time system, the front-end must produce crossing functions and handlers for the

events; the actual search for zero crossings is left to the solver [23].

6 Integration of HLA Services in Simulators

6.1 Design of the Federation. A mechatronic system is made up of several mechatronic subsystems that are composed by elementary components of various domains. It is possible to describe every component by an HLA object. Given that we are only interested in the dynamic state variables, we describe every mechanical component by an HLA object. Attributes of these objects are dynamic state variables of the global system. The OpenModelica federate has to publish attributes of its HLA objects before the beginning of the simulation. The OpenMASK federate has to subscribe to all attributes of HLA objects of the OpenModelica federate. During the simulation, the OpenModelica federate updates its HLA object attributes by calling the publish service or by sending interaction. Thereby, the RTI reflects these attributes to the OpenMASK federate.

6.2 Integration of HLA Services in OpenModelica. We propose a wrapper, which has to communicate with the simulator through sockets. This wrapper is implemented in C++ and so we are able to integrate the HLA services to exchange data with the RTI [24]. It is possible to call functions implemented in C or FORTRAN language from a Modelica program [25]. We use this Modelica property to exchange data with a wrapper by using sockets. A socket is a unique identifier representing an address on the network. The socket address is specified by the host name and the port number. We use the TCP-IP protocol to assure data transport (packages) between the process server (the wrapper) and the client (the simulation under OpenModelica).

6.2.1 Interface From OpenModelica to the RTI Communication from OpenModelica to the wrapper. The wrapper is implemented as a TCP server. The simulation calls the external C functions to connect to the wrapper and exchange data with it. Once the connection is established, the simulation sends information to the wrapper server. This last one can also send back information to the simulation. We developed five basic functions for the communication process: Function createSocket() to exchange the simulation data, function sendMessage() to send a message, function receiveMessage() to receive a message, function recMsgBlock() to freeze OpenModelica, and function clean() to close the connection socket. We created a Modelica library that we called CommunicationLib. This library contains a socket class as well as the wrapped functions using the external functions defined in C. The socket class is customizable. We can modify the IP address and the port. In our case, we use the localhost address because OpenModelica and its wrapper have to run on the same machine. We choose a fixed time step to send and receive data. The wrapper representing the TCP server is implemented in C++ (Fig. 3). A
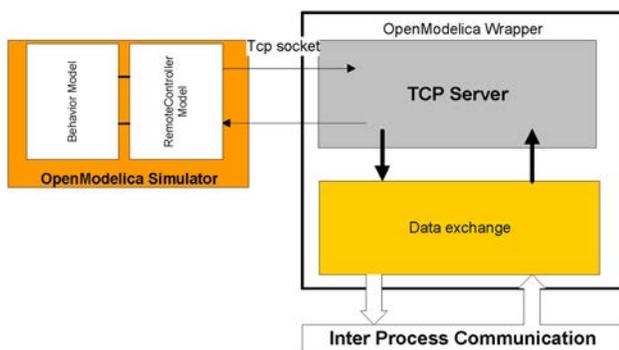
socket server is created and is listening for connection requests. When a connection request is received, a connection is established and the wrapper can receive the data from the simulation under OpenModelica. Each value received is decrypted. The name of the variable is extracted as well as its value. For that, we developed a general Modelica module that we called RemoteController (Fig. 3). This module represents a connection point between OpenModelica and any external program using a TCP server.

Every hybrid automaton represents a component of the global system. An hybrid automaton of the operative part contains a vector of state variables. Each hybrid automaton of the operative part is represented by an object in the wrapper. The attributes of this object are the state variables of the corresponding hybrid automaton. Once the wrapper receives values of the state variables of an hybrid automaton, it allocates them to the attributes of the corresponding objects.

Communication from the wrapper to the RTI. As the OpenModelica wrapper is implemented in C++, we chose that it calls directly the functions of the libRTI library, which includes the HLA services, to join the federation, to publish and/or subscribe to attributes, etc. More specifically, all requests sent by a federate to the RTI take the form of methods calls to the object RTIAmbassador within libRTI.

6.2.2 Interface From the RTI to OpenModelica. Another task that the wrapper has to carry out is the implementation of the federate ambassador class to provide callbacks functions. This last one is responsible in each federate for all the data received from the RTI. Any data received by the FederateAmbassador object can be sent to the simulation under OpenModelica through sockets. The RTI sends messages to the wrapper by calling functions. These functions are called callbacks and they are defined in the wrapper.

6.3 Integration of HLA Services in OpenMASK. An OpenMASK simulation is a set of C++ modules that interact. These modules are compiled before being executed. The idea is to integrate the HLA services in one or more of these modules before they are compiled. As the HLA RTI (CERTI) used is a C++ library, we have no problem of compatibility.

An OpenMASK simulation is organized in a tree simulation composed of modular elements called the simulation modules. These modules are connected to a bus which allows them to exchange messages and/or signals. An OpenMASK module has a predefined architecture. It is a C++ class, which can contain generic methods: Init(), compute(), processEvent(), etc. Our approach to integrate HLA services in an OpenMASK simulation is to create an overall module that we called HLAC (HLA communication). This latter invokes HLA services to communicate with the HLA RTI. The HLAC module can also contain inputs/outputs for the data exchange. At each step of simulation, the general module can read all the state variables of each visual module to allocate new values by exchanging events or signals through OpenMASK proper bus.

6.3.1 The HLAC Module. The HLAC module is composed of three parts: A first part to detect events, to catch objects attributes from the simulation model, and to transform them into calls of RTIAmbassador functions (Fig. 4). This part is called LRC (local RTI component). The second part catches messages and updates from RTI using callbacks functions declared in the class FederateAmbassador. This part is called FedCode (federate code). The third part is the internal simulation model which sends updates to other OpenMASK modules and receives events from them.

LRC. The Init() method is called one time like the constructor to initialize the attributes of the HLAC object. We use this method to invoke some HLA services to create and/or join a federation, to get handles of HLA items (HLA objects, HLA interactions, object attributes, interaction parameters, etc.) and to publish and
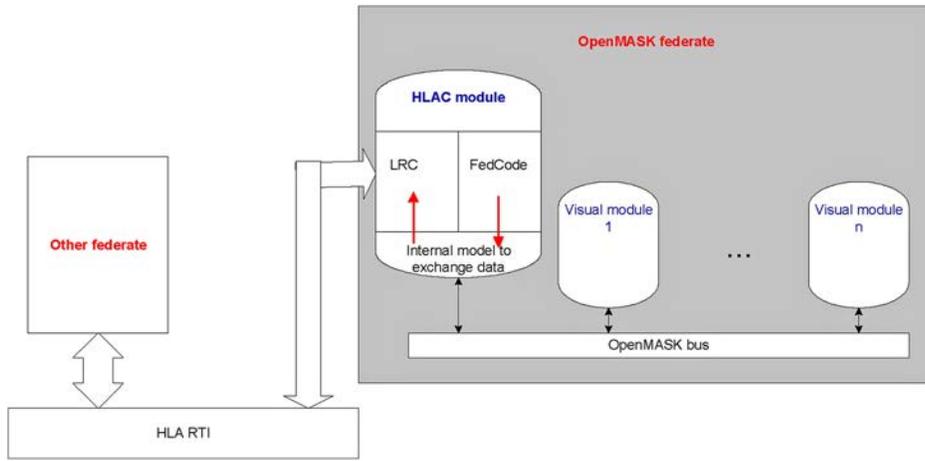

Fig. 3   OpenModelica wrapper

Fig. 4 The HLAC module

subscribe to some HLA items. In our approach, to create a federation, the LRC calls the createFederationExecution() function. To join the federation, the LRC calls the joinFederationExecution() function with three parameters (federate name, federation name, and a pointer to the class implementing the federate callbacks). Before a federate can produce any data, it must declare its intent to publish. Interaction between a user and the simulation or collision detection is represented by an HLA interaction class. To receive values for instance attributes, the federate must first subscribe to the desired class attributes. To receive interactions of a certain class, the federate must subscribe to that class of interactions. The actions carried out by a federate only once are integrated in the Init() method. Dynamic actions "performed in a cyclical manner" are implemented in the compute() method. Indeed, the compute() method is executed at a frequency attributed to each OpenMASK module at the beginning of the simulation. During the simulation, the HLAC module sends data to the RTI on updating its attributes or sending interactions. Collision detection during the simulation or an interaction with the user triggers an event. Due to this, the HLAC module can send interactions to other federates or update some of its attributes to inform other federates about the interaction or the collision. Once the end of simulation reached, it calls the resignFederationExecution() service to leave the federation.

FedCode. The FedCode part catches events and updates from the RTI using callback functions declared in the FederateAmbassador class. To use them, HLAC class must inherit from the FederateAmbassador. These callback functions are implemented in the HLAC module and outside the generic methods. The attribute values received through the callbacks are used to animate the 3D objects in OpenMASK. Each 3D object is represented by an OpenMASK module (visual module). It receives updates from the HLAC module via messages through the OpenMASK bus (see Fig. 4).

6.3.2 Communication Sequence Between the OpenMask federate (HLAC) and the RTI. We identified six steps of communication between the HLAC module and the RTI: Create and/or join federation, initialization, publication and subscription, synchronization, update attributes, and advancing in time and the last step is the simulation end (Fig. 5). The OpenMASK federate begins by joining the federation if it was already created. Otherwise, it can create the federation and join it. Then, the federate gets the handles of HLA items by sending requests to the RTI. The returned handles are unique between the RTI and the federate. The federate can at this stage declare its intent to publish or subscribe. After that, the federate requests to synchronize its execution with other federates. Once all federates are synchronized, the
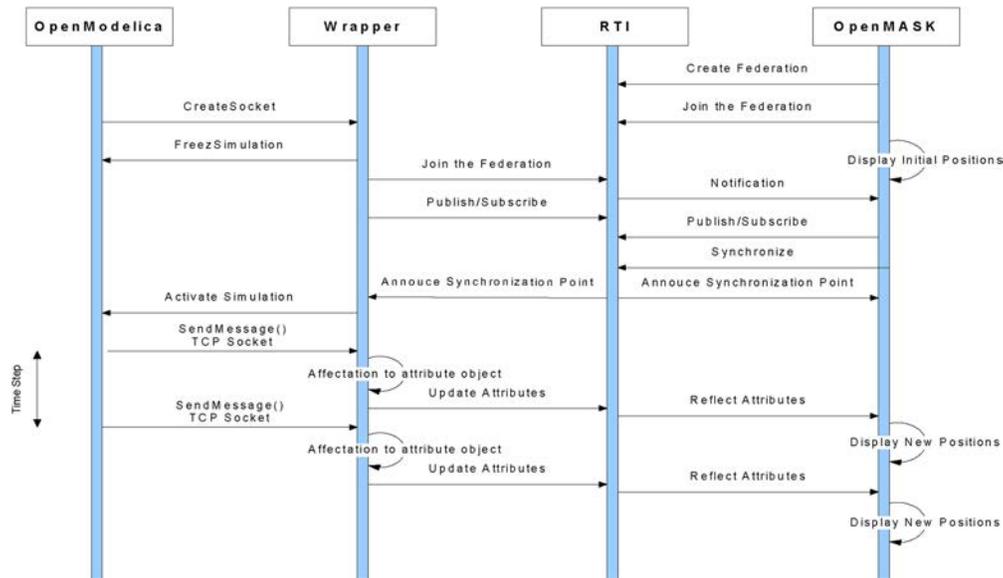

Fig. 5 A generic sequence diagram for every simulation

4

simulation starts. The simulation loop is implemented in the compute() method. The HLAC module receives values from the RTI and sends them via events to other OpenMASK modules to animate their 3D objects.

7 Synchronization and Real-Time Management

7.1 Simulators Synchronization at the Beginning of the Simulation. We propose a method to synchronize the beginning of the execution of the simulators. This solution is based on the HLA Time Management service. It consists in starting the execution of the two federates without defined order. The creator federate requests to the RTI to deliver a synchronization point to the federates. Then, the RTI deliver to both federates a synchronization point. The two federates invoke the tick() service and wait until the reception of the federationSynchronized() callback. Once this callback received, both federates start execution simultaneously. This method is simpler to implement within OpenMASK than within OpenModelica federate. Indeed, the OpenModelica wrapper must convert all the RTI callbacks to TCP requests to the OpenModelica internal simulation model. For that purpose, we have defined the procedure which is following.

In the OpenModelica wrapper, we declare the OpenModelica as a creator federate. The wrapper sends a TCP request to the OpenModelica simulation to freeze it. For this, it sends the word "stop" via the TCP request. During the simulation, the function receiveMsg() developed in the simulation internal model is listening to TCP requests from the wrapper. When the word stop is received, an internal Modelica event is generated and the function recMsgBlock() is activated. This function freezes the simulation until the reception of the word "wake." It is based on blocking socket. Then, the wrapper requests a synchronization point from the RTI and waits until reception of the callback announceSynchronizationPoint() by invoking the tick() service. Once both federates synchronized, the global simulation can be launched by pressing a key on the keyboard or by creating a timer in the creator federate. Once the user presses a key or the timer is elapsed, the wrapper must order to the OpenModelica simulation to resume its execution. This is done by sending a TCP request containing the word wake.

7.2 Degree of Involvement of Federates and Real-Time Module

7.2.1 Degree of Involvement of Federates. Time in the federate can be represented by points along a time axis of the federation. Each federate can then advance along this axis during the execution. The time management service provides mechanisms making it possible to control the advance of the federates in order to guarantee a causal order between the various emitted events. The perception of the current time can be different according to the federate, but the advance of time is coordinated by the federation. There are three kinds of federates: regulating federate, constrained federate, or both. A federate that creates an event or sends a message with a time stamp to other federates is called a regulating federate. A constrained federate is a federate that receives events or messages with time stamps from other federates. The time advance of a constrained federate depends on that of regulating federates.

The simplest approach to manage logical time in a simulation is to advance time in equal steps [26]. Each of the OpenMASK federate and the OpenModelica federate can be regulator or constrained or both. Also, at each step, the federate can send interactions and receive data from other federates. Both federates must be involved in time management and each one may choose its degree of involvement. The OpenModelica federate is a behavior federate. Its object attributes are used by the graphical OpenMASK federate to animate the 3D model. In another way, the OpenModelica federate must keep the lid on the OpenMASK federate to provide a smooth simulation. For that reason, we choose

OpenModelica as a time-regulating federate and OpenMASK as a time-constrained federate, whose advance of logical time is constrained by the regulator federate. So, the OpenModelica federate which is regulator and not constrained, paces the rest of the federation but is not constrained by it. As for the OpenMASK federate, it is constrained but not regulator. It allows the rest of the federation to regulate its logical time, but it cannot affect the other federates. This is recommended for display or passive federates.

The HLA time management coordinates the advance of logical time among all federates in a federation. The RTI prevents time-constrained federates from running off without respect to time-regulating federates. Since the OpenModelica federate is regulator, we choose to make its real-time execution. In this case, the advance of OpenMASK time is constrained by the time advance of OpenModelica. So, the federation will advance its time together in synchronicity with wallclock time. In the next part, we present a module to make OpenModelica execution real time.

7.2.2 Real-Time Module for OpenModelica. The progression of simulation time during the execution of a simulation may have or not a relationship with the progression of wallclock time. With analytic simulations where human and external materials do not interact during the execution, the simulation time progression is often not synchronized with wallclock time. These simulations are sometimes referred to as as-fast-as-possible simulations because they are executed as quickly as possible. This is the case of OpenModelica. With the real-time simulations, a mapping function to translate wallclock time to simulation time can be used [26].

$$T_s = f(T_w) = T_{\text{start}} + \text{Scale}^*(T_w - T_{w\text{Start}})$$

where T_w is a value of wallclock time, T_{Start} is the simulation time at which the simulation begins, $T_{w\text{Start}}$ is the wallclock time at the beginning of the simulation, and Scale is the scale factor.

To gain real-time execution, we augmented OpenModelica with a mechanism to pace its execution with wallclock time. The pacing mechanism simply introduces a waiting mechanism to prevent the simulation from advancing simulation time ahead of wallclock time. We applied this mechanism in the RealTime module for OpenModelica. RealTime is a Modelica model, which can be instantiated in other Modelica models that are then synchronized with real time. This feature is useful in hardware-in-the-loop simulations and for the use of interface human machine within OpenModelica simulator [10]. The basic idea of the RealTime module is to test at equal small steps if the wallclock time is greater than the simulation time. In this case, the OpenModelica simulation is frozen until the wallclock time is equal to simulation time. Sometimes, the simulation gets stuck for a while due to, for example, too many events. The lost time simulation is regained then as fast as possible by simulating as fast as possible. This can lead to incoherence execution with the external program or user interacting with the simulation because signals and values will be exchanged extremely fast in the regain phase. By limiting the simulation speed, this behavior can be avoided. The algorithm used for this module is presented in Fig. 6.

7.3 Time Evolution of the Global Simulation. The RemoteController module in OpenModelica sends values of simulation at each fixed time step to the wrapper (see Fig. 3). The time step is a parameter in the RemoteController module that can be modified. The wrapper plays the role of the OpenModelica federate. It does the usual cycle of TIME ADVANCE REQUEST (labeled «TAR») and TIME ADVANCE GRANT (labeled «GRANT»). On the other hand, the OpenMASK federate does the same cycle to advance its logical time. It is constrained by the advance of the OpenModelica time. This means that it is frozen when OpenModelica does not update its attributes or it is waiting to synchronize with wallclock time. More explicitly, when OpenModelica is

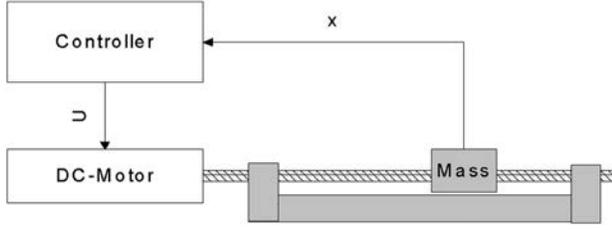**Fig. 6** Algorithm for RealTime Modelica module

Fig. 7 Drive guide system

waiting to synchronize with wallclock time, it does not send simulation values to the wrapper. The latter requests to advance its logical time only if it receives a new value from OpenModelica via sockets. As the wrapper is time-regulating and the OpenMASK federate is time-constrained, this latter cannot advance its logical time and it is frozen when OpenModelica is synchronizing with wallclock time.

8 Experimentation

After the integration of the HLA services in both simulators, we applied our global approach to a mechatronic system. We chose a simple but significant example. At the end, we obtained a virtual prototype of the system. We chose a linear drive system to test out our approach. The linear drive is controlled by a controller module which delivers the dc motor supply voltage according to the position x of the linear drive. The specification of the control-

ler is such as follows: start and drive to left then run ten times between left and right end positions. Finally, pause 3 s, afterward continues.

The physical system model of the linear drive, depicted in Fig. 7, is composed of a dc motor, a threaded rod, and a threaded mass.

The dynamic equations of the global system are [27]

$$(J + mp^2)\ddot{\theta} + b\dot{\theta} = ki \ \text{ and } \ x = p\theta \tag{1}$$

$$L\frac{di}{dt} + Ri = V - k\dot{\theta} \tag{2}$$

The behavior of the global system is represented by an hybrid automaton (Fig. 8). This one is implemented under OpenModelica using our HybridAutomataLib [10]. α and β represent transitions coupling. If a transition is fired in the control part, its corresponding transition is fired at the same time in the operative.

We instantiated the RealTime module and inserted it in the global model to synchronize the time execution with the wallclock time. We performed tests and measured the time difference along the simulation between wallclock time and simulation time is shown in Fig. 9. The result is that the time difference is on the order of 10^{-2} s.

A 3D model was created and inserted in OpenMASK. This 3D model was inserted in the generic OpenMask simulation tree needed for execution. The RTI is running in a first Linux terminal. OpenMASK is running in a second terminal. OpenModelica wrapper is running in a third terminal (Fig. 10). Once the OpenMASK
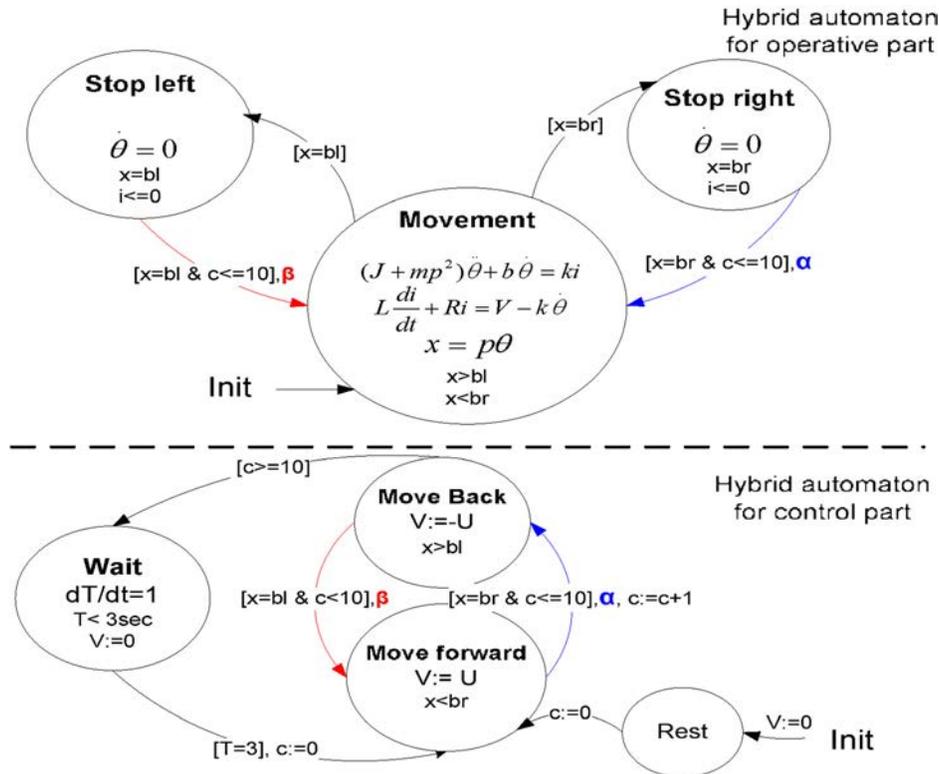

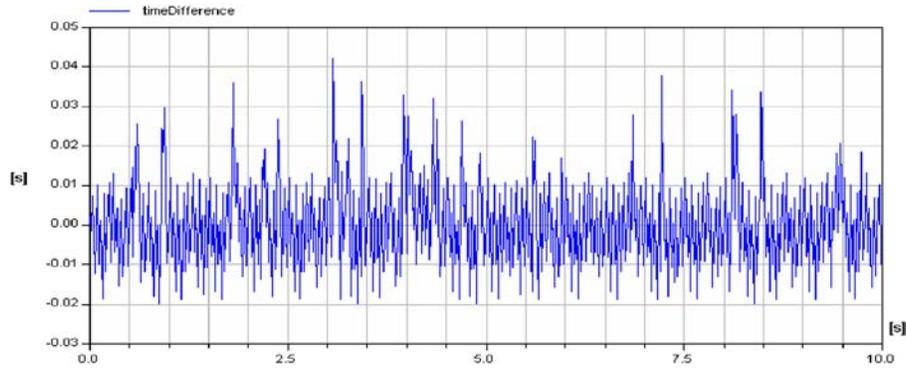**Fig. 8** Hybrid automaton of the drive guide system

Fig. 9   Time difference between wallclock time and simulation time inside OpenModelica

federate is launched, it displays the 3D objects with initial attributes. Finally, in a fourth terminal OpenModelica starts to simulate the hybrid automaton. Once the OpenModelica simulation starts, variables are communicated to OpenMASK via the RTI to animate the 3D objects.

9   Discussion

In the case of large models, efficiency problems can arise. We require making sure that all of the computations associated with a single integration step are completed within the allowed time slot [28]. Indeed, the computations that need to be performed in each integration step of the simulation can vary greatly. A summary of issues on the integration algorithms related to the special demands of real-time simulation can be found in Ref. [28]. Methods in available literature on simulation speed-up are mentioned in this book. A distributed simulation in multiple machines in our case could be a solution to avoid the problem of overrun. As the Open-Modelica simulator is HLA compatible, large models can be dissociated and executed by different instances of OpenModelica.

Another aspect is to make the simulation more reactive referring to events. We propose to manage external events as interactions with haptic interfaces and internal events as collision detection into the OpenMASK simulator. In this research, we have not yet addressed this aspect. A reform of the time advancing strategy of the global simulation is required. We propose two ways of improvement. The first one is to define each simulator as constrained and regulator at the same time. Thus, OpenMask will not be only a passive federate because it will interact with the user. The second way is linked to the accuracy in the case of bidirectional communication where events from OpenMASK serve as inputs for OpenModelica. In this case, stepped time synchronization must be avoided because it can result in significant errors linked to undetected events inside a time step. But in addition to the time-stepped pattern, another typical pattern of time manage-

ment is the event-driven pattern [26]. Instead of using TIME ADVANCE REQUEST, as does a time-stepped federate, the event-driven federate uses NEXT EVENT REQUEST to request an advance of its clock. The federate specifies with the request the logical time of the next event on its internal queue. The simulation proceeds by processing its next event, that is, the known event with the smallest future logical time [13].

10   Conclusion

In this paper, we presented an open source approach for modeling and simulating mechatronic systems based on the HLA. General methods for integrating HLA services in simulators are presented and applied on the OpenMASK and OpenModelica simulators. Synchronizing the different simulators is achieved by using the HLA time management services. To make the global simulation real-time, we proposed an approach based on real-time techniques and HLA time management services. All open source modules developed are available in our laboratory and will be officially registered on the Modelica site. In the future work, we will treat the extension presented especially in Sec. 9. We plan to manage events taken into account by OpenMASK and benefits from its interactive feature.

Fig. 10   A print screen of the global simulation

References

[1] Gu, B., and Asada, H., 2004, "Co-Simulation of Algebraically Coupled Dynamic Subsystems Without Disclosure of Proprietary Subsystem Models," ASME J. Dyn. Syst., Meas., Control, **126**(1), pp. 1–13.

[2] http://www.3ds.com/products/catia/portfolio/dymola/

[3] http://www.mscsoftware.com/Products/CAE-Tools/Adams.aspx

[4] Krus, P., 1999, "Modelling of Mechanical Systems Using Rigid Bodies and Transmission Line Joints," Trans. ASME J. Dyn. Syst. Meas., Control., **121**(4), pp. 606–612.

[5] Krus, P., and Jansson, A., 1990, "Distributed Simulation of Hydromechanical Systems," Third Bath International FluidPower Workshop, Bath, United Kingdom.

[6] Pulko, S. H., Mallik, A., Allen, R., and Johns, P. B., 1990, "Automatic Time-stepping in TLM Routines for the Modelling of Thermal Diffusion Processes," Int. J. Numer. Model., **3**, pp. 127–136.

[7] http://www.irisa.fr/bunraku/OpenMASK/

[8] http://www.ida.liu.se/~pelab/modelica/OpenModelica.html

[9] Sghaier, A., and Soriano, T., 2008, "Using High Level Models for Modelling Industrial Machines in a Virtual Environment," Int. J. Interact. Des. Manuf., **2**, pp. 99–106.

[10] Hadj-Amor, H. J., 2008, "Contribution au Prototypage Virtuel de Systèmes Mécatroniques Basé sur une Architecture Distribuée HLA. Expérimentation sous les Environnements OpenModelica—OpenMASK," Ph.D. thesis, Université du Sud Toulon Var, France.

[11] IEEE1516-2000, 2000, IEEE Standard for Modeling and Simulation High Level Architecture (HLA)—Framework and Rules.

[12] IEEE1516.1-2000, 2000, IEEE Standard for Modeling and Simulation High Level Architecture (HLA)—Federate Interface Specification.

[13] Kuhl, F., Dahman, J., and Weatherly, R., 1999, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*, Prentice Hall PTR, Piscataway, NJ.

[14] http://www.cert.fr/CERTI/

[15] Bréholée, B., and Siron, P., 2002, "CERTI: Evolutions of the ONERA RTI Prototype," 2002 Fall Simulation Interoperability Workshop, Orlando.
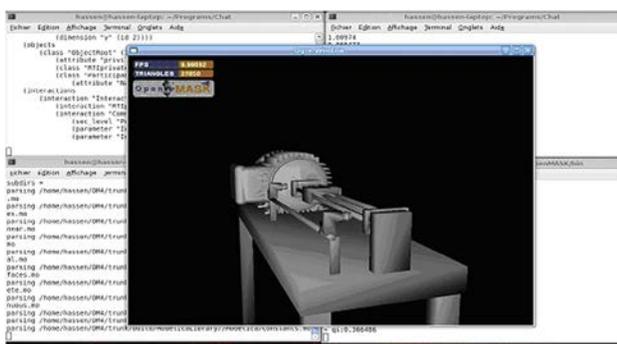
[16] Straßburger, S., Schulze, T., Klein, U., and Henriksen, J. O., 1998, "Internet-Based Simulation Using Off-The-Shelf Simulation Tools And HLA," *Proceedings of 30th Conference on Winter Simulation*, Washington, DC, pp. 1669–1676.

[17] Borchshev, A., Karpov, Y., and Kharitonov, V., 2002, "Distributed Simulation of Hybrid Systems With AnyLogic and HLA," FGCS, Future Gener. Comput. Syst., **18**(6), pp. 829–839.

[18] Kanai, S., and Shimizu, T., 2003, "HLA/RTI-Based ScalableDistributed Virtual Prototyping Environment for Embedded System Design," *Proceedings of Virtual Concept 2003*, Biarritz, France, pp. 100–107.

[19] Raulet, V., 2003, "Prototypage Interactif et Collaboratif. Vers une Architecture de Communication pour une Interactivité Coopérante Dynamique dans les Environnements Virtuels Distribués," Ph.D. thesis France, Université de Bretagne Occidentale, France.

[20] Margery, D., Arnaldi, B., Chauffaut, A., Donikian, S., and Duval, T., 2002, "OpenMASK: Multi-Threaded or Modular Animation and Simulation Kernel or Kit: A General Introduction," *VRIC 2002 Proceedings*, Laval, France, pp. 101–110.

[21] Fritzon, P., 2003, *Principles of Object-Oriented Modeling and Simulation With Modelica 2.1*, IEEE Press/Wiley-Interscience, New York.

[22] Brenan, K. E., Campbell, S. L., and Petzold, L. R., 1989, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier, New York.

[23] Lundvall, H., Fritzson, P., and Bernhard, B., 2008, "Event Handling in the OpenModelica Compiler and Runtime System," Technical Reports in Computer and Information Science.

[24] Hadj-Amor, H., and Soriano, T., 2008, "Integrating OpenModelica Simulator With HLA," 7th France-Japan (5th Europe-Asia) Congress on Mechatronics, IEEE Mecatronics2008.

[25] Fritzon, P., Lundvall, H., Fritzson, P., and Bachmann, B., 2007, "OpenModelica System Documentation," Preleminary Draft, 2007-06-20 for OpenModelica 1.4.3.

[26] Fujimoto, R. M., 2000, *Parallel and Distributed Simulation Systems* (*Wiley Series on Parallel and Distributed Computing*), Albert Y. Zomaya, ed.

[27] Aublin, M., Boncompain, R., Boulaton, M., and Caron, D., 1992, *Systèmes mécaniques, théorie et dimensionnement*, Dunod, Paris.

[28] Cellier, F. E., and Kofman, E., 2006, *Continuous System Simulation*, Springer-Verlag, New York.