17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, Delft, Oct 30-Nov 1, 2013

Distributed Simulation of Heterogeneous and Real-time Systems

Gilles Lasnier, Janette Cardoso, Pierre Siron
ISAE - Toulouse University, France
Email: {firstname.name}@isae.fr

Claire Pagetti
ONERA - Toulouse, France
Email: claire.pagetti@onera.fr

Patricia Derler
UCB - United States
Email: pd@eecs.berkeley.edu

***Abstract*—This work describes a framework for distributed simulation of cyber-physical systems (CPS). Modern CPS comprise large numbers of heterogeneous components, typically designed in very different tools and languages that are not or not easily composeable. Evaluating such large systems requires tools that integrate all components in a systematic, well-defined manner. This work leverages existing frameworks to facilitate the integration offers validation by simulation. A framework for distributed simulation is the IEEE High-Level Architecture (HLA) compliant tool CERTI, which provides the infrastructure for co-simulation of models in various simulation environments as well as hardware components. We use CERTI in combination with Ptolemy II, an environment for modeling and simulating heterogeneous systems. In particular, we focus on models of a CPS, including the physical dynamics of a plant, the software that controls the plant, and the network that enables the communication between controllers. We describe the Ptolemy extensions for the interaction with HLA and demonstrate the approach on a flight control system simulation.**

I. Introduction

A. Context

Designing real-time cyber physical systems (CPS) is a complex task which typically entails the use of many different methodologies and tools in different areas and at different stages of development process. Typically, different parts of the CPS are developed by different engineering teams or even external contractors. Integrating these heterogeneous parts and evaluating the composite behavior is challenging and, with today's tools, a systematic composition is nearly impossible. Over the last years, model-based tools have gained momentum in the design of CPS as they allow for modeling at higher levels of abstraction and they facilitate the communication between different teams of engineers. Additionally, many tools come with add-ons such as formal verification and code-generation. However, different parts of a CPS require different abstractions and tool support. The lack of interoperability between the tools poses a major challenge.

For control-command systems, Matlab/Simulink [1] is often used to specify the plant dynamics and the control. Thus from the specification it is possible to simulate the behaviour and verify the robustness and stability of the functions by observing the functional traces. This step occurs early in the development process. The control functions are then translated into low level code (automatically or manually) that is then executed on the target architecture (e.g. a distributed platform composed of several computing nodes connected with a real-time operating system). Analysis of the execution on the target occurs very late in the development process. It is then of great importance to provide methods and tools to give ways to analyze, at least partially and as early as possible, the behaviors of the future implementation taking into account, for instance, the mapping on the target or the interaction with some physical devices (e.g. sensors and actuators).

The process of integrating components and evaluating their interaction with the target architecture requires knowledge of several domains (e.g. hardware, software) that are typically described in heterogeneous models (e.g. continuous models, discrete models, timed properties). Analyzing worst case execution times is insufficient in order to validate the system. Formal analysis cannot usually handle systems of that complexity. As a result, simulations are performed to analyze the functional behaviour of high level specifications mixed with more low level elements. Imagine, for instance, a flight control composed of two sub-functions distributed on two calculators communicating via an Ethernet network. To observe the real behavior on the platform, it is necessary to describe the functional behavior of each component as well as the timing of the network that communicates the data between the two sub-functions.

B. Contribution

The purpose of our work is to provide a framework for simulating heterogeneous models designed at different levels in the development process. To do so, we propose a co-simulation framework that leverages two open source tools: Ptolemy II and HLA/CERTI.

Ptolemy II [2] is an open source modeling and simulation tool for heterogeneous systems, developed at the University of California Berkeley. This tool is well suited for modeling CPS [3] by providing different models of computation (MoC) such as continuous time for describing physical properties or discrete events for describing software and control.

The IEEE High-Level Architecture (HLA) standard [4], [5] targets distributed simulation. A CPS is seen as a federation grouping several federates which communicate via publish/subscribe patterns. This decomposition into federates allows to combine different types of components such as simulation models, concrete functional codes (in C++, Java, etc.) and hardware equipments. The key benefits of HLA are interoperability and reuse. Our experiments and framework have been developed upon the HLA-compliant Open-Source RTI named CERTI [6], [7].

In the co-simulation environment described here we use various Ptolemy simulations that are executed as federates in an HLA federation. The Ptolemy federates exchange data with federates that can be other Ptolemy simulations or even C++ or Java federates. In order to facilitate the interaction between HLA and Ptolemy, we extend Ptolemy with dedicated components that enable the connection to HLA and the data exchange. The *HlaManager* centrally manages the advancement of time in HLA and Ptolemy. Two Ptolemy actors, an *HlaPublisher* and an *HlaSubscriber* are in charge of the data communication. Combining these two frameworks allows experimenting with: heterogeneity provided by Ptolemy (i.e. the possibility to mix continuous, discrete or other MoCs) and interoperability provided by HLA (i.e. the possibility to mix simulation models, pieces of code and physical equipments).

C. Related work

Interoperability is a very important issue in distributed discrete-event simulation as can be observed by several works targeting co-simulation. Closely related to our work are two particular cooperation tools that provide similar capabilities by providing HLA plugins.

In [8], the authors encode a connection between HLA and Modelica [9] for the virtual prototyping of mechatronic systems. In [10], the authors develop an HLA Blockset and an HLA Toolbox which provides a connection between HLA and MATLAB/Simulink. However, these solutions are not open source and the synchronization time between the federates is not described in literature. Our framework is open source, using existing open source frameworks and is intended for research, teaching, and industrial usage and several solutions for mixing the timing are explained in the current paper.

Related to this work in a broader context ins the work on Functional Mock-up Interfaces (FMI) [11], which is lead by a consortium of industrials and academics. The standard defines an interface standard for coupling different simulation tools in a co-simulation environment (e.g. by giving standardized access to simulation model equations). In this case, co-simulation is a simulation technique for coupled time-continuous and time-discrete systems. This work is rather recent and still undergoing various changes: a new version is currently being standardized [11]. Even if part of our research is similar, FMI does not answer all our objectives presented above. The current standard presents limitations to manage discrete-event simulation properly [12] and no information is provided to address distributed simulation challenges.

This paper is organized as follows. An overview of HLA and Ptolemy II are presented in Section II and Section III. Section IV describes the co-simulation framework and shows how interoperability is obtained. Section V illustrates the results of our approach applied to a concrete case-study: a flight control system of an aircraft. Finally, Section VI presents concluding remarks and our future work.

II. Distributed Simulation with HLA

In this section, we first give a brief description of how an HLA simulation is implemented. We then describe the subset of HLA services necessary to allow a Ptolemy model to participate in an HLA federation, focusing on the time management in an HLA simulation.

A. Overview of HLA

The High-Level Architecture (HLA) [5], [4] is a standard for distributed discrete-event simulations, generally used to support analysis, engineering and training. The approach promotes reusability and interoperability. In HLA terminology, the entire system to be simulated is represented by a *federation* which is a collection of *federates*, i.e. simulation entities performing a sequence of computations. Federates are connected via the *Run-Time Infrastructure* (RTI), the underlying middleware functioning as the simulation kernel. Figure 1 describes the global architecture of a HLA simulation.

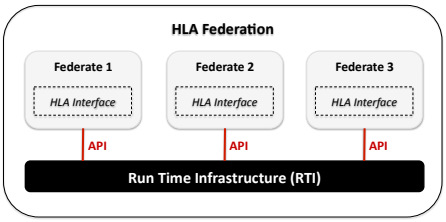


Fig. 1. HLA Federation

The HLA specification defines:

1) An interface specification for a set of services required to manage the federates and their interactions. For instance, it describes how a federate can join or create a federation.
2) An object model template (based on the OMT standard [**?**]) which provides a common framework for the communication between HLA simulations. For each federation, a Federation Object Model (FOM) describes the shared objects, interaction classes and their attribute.
3) A set of rules describing the responsibilities of federations and the federates. An example is the rule that *all data exchange among federates shall occur via the RTI.*

HLA services are grouped into six management areas related to the federate life cycle. Figure 2 describes the subset of HLA services used for our approach. An informal description of each service role and a user documentation is available. The reader is referred to [4], [5] for a complete description of all HLA services. The services in Figure 2 deal with the following areas: (1) Federation management: also includes a Federation Execution Data (FED) file used by the RTI to manage the whole federation; (2) Declaration management, i.e. which objects or *object attributes* each federate will publish or subscribe to; (3) Object management, i.e. the way federates produce attribute updates or receive updated attributes from the federation; (4) Time management, i.e. the mechanisms required to implement time management policies and to negotiate time advances.

B. Time Management

HLA time management services enable deterministic and reproducible distributed simulations. Each federate manages its own logical time and communicates this time to the RTI. The RTI ensures correct coordination of federates by

Areas	Services	Description (non-formal)
Federation	createFederationExecution()	create a federation
	joinFederationExecution()	join a federation
	resignFederationExecution()	quit a federation
	destroyFederationExecution()	destroy a federation
	registerFed...Sync...Point()	register a synchronization point
	sync...PointReg...Succeeded()⋆	register synchro point succeeded
	announceSynchro...Point() ⋆	wait a synchronization point
	synchronizationPointAchieved()	release from a synchro. point
	federationSynchronized() ⋆	announce synchronization
	tick()	allow to get callbacks from RTI
Declaration	publishObjectClass()	declare publication of a class
	subscribeObj..ClassAttributes()	subscribe to a class
	unsubscribeObjectClass()	unsubcribe to a class
	unpublishObjectClass()	unpublish a class
Object	registerObjectInstance()	register an object instance
	discoverObjectInstance() ⋆	for object instances discovering
	updateAttributeValues(), UAV	send & update value
	reflectAttributeValues() RAV ⋆	receive updated value
Time	enableTimeRegulation()	declare federate is regulator
	timeRegulationEnabled() ⋆	federate as regulator succeeded
	enableTimeConstrained()	declare federate constrained
	timeConstrainedEnabled() ⋆	federate as contrained succeeded
	timeAdvanceRequest(), TAR	ask to advance federate's time
	timeAdvanceGrant() TAG ⋆	notify time advancement granted
	nextEventRequest(), NER	ask to advance federate's time

Fig. 2. HLA services with a ⋆ are sent from RTI to Federates (callbacks); all other services are from Federates to RTI.

advancing time coherently. Logical time is roughly equivalent to *"simulation time"* in the classical discrete event simulation literature, and is used to ensure that federates observe events in the same order [13]. Logical time is not necessarily mapped to real time.

1) Federate's time policies: HLA time policies describe the involvement of each federate in the progress of time. It may be necessary to map the progress of one federate to the progress of another. A *regulating* federate participates actively in the decisions for the progress of time. A *constrained* federate follows the time progress imposed by other federates. A combination of both policies is possible. As our approach deals with the synchronization of logical time from different simulation tools, only *regulating and constrained* federates are allowed.

To properly handle time progress and ensure causality, HLA defines Time-Stamp Ordered (TSO) events which are supposed to occur at specific points in time. Regulating federates generate TSO events (possibly out of time-stamp order) that must occur no earlier than the current local time plus the *lookahead*. The *lookahead* acts as a contract value which guarantees that the federate will not produce a TSO event earlier than its current local time plus *lookahead*.

The RTI handles the ordering of the TSO events generated by the federation. Each federate implements a priority time-stamp queue for TSO events. Events stored in the ordered queue are only delivered to the federate in a time advancement phase if their timestamps are between the current and the next federate local time (the granted time, see later).

2) Time progress: Time advancement requests by federates are made through two particular services (see Figure 2): the *timeAdvanceRequest* service (*TAR*) is used to implement time-stepped federates; the *nextEventRequest* service (*NER*), is used to implement event-based federates. The granted time is provided by *timeAdvanceGrant* (TAG).

The time advancement phase of a Federate F in HLA is a three-step process: 1) F sends a request using NER or TAR services; 2) F can receive *reflectAttributeValue* (RAV) callbacks (i.e. updated HLA attributes); 3) F waits for the granted time t_G (TAG). At the TAG(t_G) reception, the federate's local time will be advanced to t_G according to the made request (NER or TAR).

A part of the time advancement phase semantics is shown in Figure 3. FederateTAR (see figure 3.a) produces an event at time t_1, with *updateAttributeValue* (*UAV*). The current (logical) time is t_1; let us consider that the next event is e_2 with timestamp t_2, $t_2 = t_1 + \Delta$, $\Delta > 0$. The federate asks the RTI for a time progress with the invocation of *TAR(t_2)*. Until the reception of the *TAG(t_2)* callback, the logical time of the federate is stalled at t_1, and it can receive a *RAV(v,t_1')* callback. Timestamp t_1' is in the interval $]t_1, t_2]$. FederateTAR can realise a computation with the values received with *RAV*. The federate then receives *TAG(t_2)* and can increase its local time to t_2.

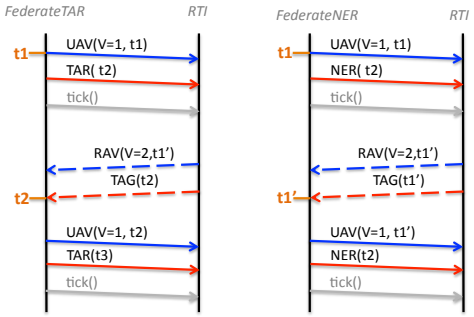


Fig. 3. Time advancement services a. TAR b. NER

Let us now consider Figure 3.b. FederateNER also produces the event *UAV* at time t_1 and asks for a time advance to t_2, in that case with a *NER(t_2)*. The reception of *RAV(v,t_1')* is followed by a *TAG(t_1')*. FederateNER moves its logical time to t_1' and then can realize a computation with the received value. Let us point out that a federate using the NER service is a more reactive federate since it can produce new events from time t_1'.

In a nutshell, if a TAR(t_2) has been sent, the time granted by the TAG service is $t_G = t_2$. If a NER(t_2) has been sent, the granted time is $t_G = t_1'$, with $t_1 < t_1' \leq t_2$. In the case of NER, if $t_1' < t_2$, the current federate have received one or more events from the federation with timestamp t_1'.

III. The Ptolemy II framework

A. Overview of Ptolemy II

Ptolemy II [2] is an open source modeling and simulation framework for *heterogeneous* systems. Ptolemy models are actor-oriented. *Actors* are executable and concurrent components that communicate via *ports*. Actors can be atomic or composite, where a composite actor contains an entire model inside but behaves like an atomic actor to the outside. A special model component, the director, describes the semantics of a model. Ptolemy supports a wide variety of models of computations, including discrete event (DE), continuous time (CT), synchronous reactive (SR) or synchronous data flow (SDF).

The operational rules for executing a model are given by the MoC, defining the meaning of execution, concurrency and communication. These rules determine when actors perform internal computation, update their internal state, and perform external communication. Models in different MoCs can be composed hierarchically with a clearly defined semantics.

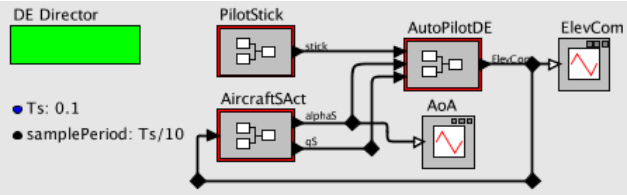


Fig. 4. A hierarchical Ptolemy model of a CPS.

The ability to model heterogeneous systems in Ptolemy is interesting for CPS design as those typically comprise heterogeneous components expressed naturally in different MoCs. An example of a hierarchical Ptolemy model of a CPS – based on the F-14 Longitudinal Flight Control demo model from Matlab – is shown in Figure 4. `Aircraft` and `Stick` are composite actors modeled in the continuous MoC and the `AutoPilot` composite actor (controller) contains is a DE system. The top-level director is also a DE director. Heterogeneous composition of different MoCs sometimes requires special actors. For instance, signals in the continuous domain need to be sampled in order to be used in the DE domain. In Ptolemy, a Sampler actor needs to be placed. In order for continuous actors to read signals generated in the DE MoC, a ZeroOrderHold actor is used [14].

The next sub-section presents the semantics of the Ptolemy entities involved in our approach.

B. Ptolemy's entities semantics

1) Actor: An actor executes in three phases: a *setup* phase, a sequence of *iterations*, and a *wrapup* phase, as represented in Figure 5.

Fig. 5. Actor's lifecyle in Ptolemy II

The *setup* phase of an actor is divided in two sub-phases: *preinitialize* and *initialize*. The preinitialize sub-phase is performed once, at the beginning of a simulation. It specifies actions that may modify or influence the actor architecture and consequently impact static analysis as well as resource deployment (composite actors may require the instantiation of internal actors), type resolution, schedulability, etc. The initialization sub-phase initializes parameters, resets local state, and sends initial messages on output ports.

An *iteration* is a sequence of operations that read input data, produce output data, and update the state. It is divided in three sub-phases (prefire, fire and postfire). The *prefire* (optionally) allows the verification of preconditions to ensure that the actor will complete the iteration. The *fire* sub-phase typically performs the computation of the actor. In this sub-phase, input data from ports is read, data is processed and output data is produced on output ports. An actor may have persistent state that evolves during execution. Finally, the *wrapup* phase performs the correct termination of the actor.

An actor comes with a set of primitive communication operations which allow to retrieve information from the communication channels (*get* method) or send information to the channels (*put* method). As the meaning of the communication is determined by the director, the connection between the external port of a composite actor and some other port on the outside will obey the semantics of the director on the outside. For example, in figure 4, the three composite actors must send/receive data obeying DE semantics imposed by the DE top level director.

2) Attribute: Attributes are used to store parameters and access them anywhere in the model.

The attribute's lifecycle is similar to the actor's one but contains only the setup and wrapup phases. Thus, an attribute can be used to statically or dynamically parameterize a model.

3) Decorator: Some actors and some attributes are implemented as so-called decorators. These entities can decorate other Ptolemy entities (e.g. model, actors, etc) with specific information in an 'aspect-oriented' way. Any number of attributes can be attached to a Ptolemy entity.

The remainder of this paper explains how a specific attribute may be designed to enable the time synchronization between both Ptolemy and HLA/CERTI at the director level and by preserving the MoC semantics.

4) Model and director: The director is responsible for the progress of the simulation by firing the actors in a specific order. In timed actors the model time (i.e. logical time) is advanced as part of this process. In a hierarchical model, only the top-level director advances time. The rules of time advancement depend of the (timed) MoC implemented by the director. Ptolemy II uses the same model of time for all MoC, known as *superdense time*. Superdense time is represented by a pair (t, n), called a timestamp, where t is a model time and n is a micro-step. The model time represents the time at which some event occurs, and the microstep represents the sequence of events that occur at the same model time. The superdense time semantics is defined in [15] and only an overview is given here.

In the case of a CT director, at time t_k, all actors are fired following a precise order and the next timestamp $t_k + 1$ is computed by the solver.

In the case of a DE director, an actor is only fired at the timestamp t if either it has an event in one of its input ports or it has explicitly asked to be fired at t (using the `fireAt` method). To ensure determinism, the order in which actors are fired is important. The DE director maintains a calendar queue of DE events and a global ordering between these events. A DE event is a tuple $(value, timestamp)$, where the timestamp is a superdense time (t, n). Let us consider $e1 = (t_1, n_1)$ and $e2 = (t_2, n_2)$. The calendar queue is sorted:

1) by timestamp, if $t_1 < t_2$ or if $t_1 = t_2$ and $n_1 < n_2$ then $e1$ is executed before $e2$;
2) by actor's ranking, if $t_1 = t_2$ and $n_1 = n_2$, the director selects the actor that has the lowest rank.

The actor ranking is defined by a topological sort of the actors in the model in data-precedence order [15].

The time advancement phase in a DE model is observed when the DE director selects the earliest event in the calendar queue, making its timestamp the current model time (and firing the destination actor of the event). This algorithm is performed at the director level and during this phase no actor is fired.

IV. Cooperation for CPS Development

The goal of this work is to integrate a (hierarchical) Ptolemy model as one or more federates in a HLA federation. This is done by introducing a new attribute to the Ptolemy model, called `HlaManager`, and two new actors `HlaPublisher` and `HlaSubscriber`. These components perform the interface between Ptolemy and HLA/CERTI environments.

This section is organized as follows. Subsection IV-A discusses the extension of the Ptolemy's framework to handle the time management between Ptolemy models and a HLA/CERTI federation. Subsection IV-B describes the `HlaManager` attributes which implements the time synchronization and the communication interface between both environments. Subsection IV-C presents the `HlaPublisher` and `HlaSubscriber` actors which respectively allows to publish and to subscribe to a HLA attribute from a Ptolemy model. Finally, Subsection IV-D describes the relation and the interaction between these three components to enable the interface Ptolemy - HLA/CERTI.

A. Time management

The time management between Ptolemy and HLA/CERTI is a fundamental issue in our co-simulation approach. As we pointed out in our motivation, dealing with co-simulation for distributed real-time and embedded systems requires synchronization between the different clocks involved in a simulation model, a simulation implementation or the physical world. The next sub-section defines the semantics of the time synchronization between both environments.

1) Time synchronization semantics: Our strategy requires the use of adequate time management mechanisms to handle clock time synchronization between both simulation frameworks. Section II-B has detailed the two semantics of time advancement provided by the HLA standard. Section III-B has described how the time is managed by Ptolemy in particular in the discrete-event (DE) domain.

In order to use a DE model as a federate, the time advancement algorithm of a DE director needs to be adapted. Let us consider that the current time of both environments is t_1 and the next DE event in the calendar queue is e_2 with timestamp t_2 and $t_1 < t_2$. After selecting e_2, the director must check, using a TAR or NER request, if it can advance its current time to t_2 considering the federation time. A granted time t_G with $t_1 \leq t_G \leq t_2$, allowed by the federation, has to be returned to the director to advance time.

No DE actor is fired during the DE time advancement phase and no DE event (from Ptolemy) is added to the calendar queue. The HLA time management semantics ensures that all HLA events with a timestamp smaller than t_G have been delivered and no HLA event with a timestamp smaller than t_G will be produced by the federation after the reception of TAG(t_G). This guarantees that the director can advance its time to t_G safely.

A more complex case is when NER(t_2) is used and TAG(t_G) is received with $t_G < t_2$. As HLA events (from the federation) are introduced in the Ptolemy model as DE events in the calendar queue and as t_G is the time that the director advances to then the DE semantics is preserved. These events will be handled on time.

The proposed time synchronization semantics guarantees that no event will be missed and a correct time synchronization between both environments is performed.

2) `TimeRegulator` *interface:* A specific interface called `TimeRegulator` has been designed to implement the time synchronization between Ptolemy and HLA/CERTI [1]. This interface is implemented by a Ptolemy attribute and allows to parameterize the time advancement algorithm of a director. The `TimeRegulator` provides a unique method called `proposeTime`() which implements the required time advancement adaptation by preserving the director semantics. Whenever a director has to advance its current time, if one or more `TimeRegulator` attributes which implement different time synchronization semantics (e.g. one for real-time synchronization with the execution platform or one to synchronize with the HLA federation time) are deployed in the model, then the `proposeTime`() method of those attributes will be called before the time is advanced. The smallest time returned by those attributes is returned to the director. If there is no such attribute, the initial time advancement algorithm is performed.

The `proposeTime`() method has a parameter `proposedTime` which indicates the time the director wants to advance to. In the case of the DE director, the `proposedTime` is the timestamp of the next event to be consumed from its calendar queue (i.e. timestamp t_2 of event e_2 in our example presented above). In our approach, the `proposeTime`() method has to assume the call to the required HLA time management services (NAR or TER) and to treat the HLA events received through the RAV callbacks during this phase. The method also ensures that the returned time is at least equal to the current time of the director or at more equal to the `proposedTime`. This method is sill executed at the director level.

The next section presents the new Ptolemy attribute defined in our approach: the `HlaManager`. This attribute implements the `TimeRegulator` interface and the time synchronization semantics presented above (implemented in its `proposeTime`() method). The `HlaManager` also implements the communication interface between Ptolemy and HLA/CERTI, using two new actors `HlaPublisher` and `HlaSubscriber` (presented in Subsection IV-C).

[1]This contribution is a joint work between ISAE/DMIA, UCB/EECS and ONERA/DTIM) The *SyncronizeToRealTime* attribute is another attribute using the `TimeRegulator` interface.

B. `HlaManager` *attribute*

The `HlaManager` attribute enables the interoperability between a Ptolemy model and a HLA/CERTI Federation. In addition to manage the time synchronization between the *Ptolemy model time* and the *HLA logical time* (by implementing the `TimeRegulator` interface), it handles all the logic, data structures and operations required to manage a Ptolemy Federate. To do so, the `HlaManager` attribute is built on the top of the *JCERTI* API, a Java binding of CERTI based on HLA 1.3 version. The API provides service relative to Federation, Declaration, Object and Time management areas in HLA, as summarized on Table 2.

1) Mapping between HLA services and Ptolemy attribute's method: As presented in Section III an attribute deployed in the model can be executed by both director and actor. In our approach, the `HlaManager` attribute provides methods that are executed by the DE director or the HLA actors. Figure 6 presents the mapping between HLA services execution (indicating the related management areas presented in Table 2) and the attribute's methods. This correspondence HLA/CERTI - Ptolemy acts as backbone in the architecture of the `HlaManager` implementation.

Ptolemy attribute	JCERTI bindings	HLA
• pre-initialize() ***invoked by Director***	• *instantiate rti'sproxy and federate's proxy*	
	• createFederationExecution • joinFederation	*Federation*
	• subscribeObjectClassAttributes • publishObjectClass	*Declaration*
	• registerObjectInstance	*Object*
• initialize() ***invoked by Director***	• enableTimeRegulation • timeRegulationEnabled (callback) • enableTimeConstrained • timeConstraintedEnabled (callback)	*Time*
	• registerFederateSynchronizationPoint • Synch...PointReg...Succeeded (callback) • announceSynchronizationPoint (callback) • synchronizationPointAchieved • federationSynchronized (callback)	*Federation*
• wrapup() ***invoked by Director***	• unpublishObjectClass • unSubscribeObjectClass	*Declaration*
	• resignFederationExecution • destroyFederationExecution	*Federation*
• proposeTime() ***invoked by Director***	• timeAdvanceRequest ou nextEventRequest • timeAdvanceGrant (callback)	*Time*
	• reflectAttributeValues (callback, if received)	*Object*
• updateHlaAttribute() ***Invoked by HlaPublisher***	• updateAttributeValues (invoked by a HlaPublisher actor)	*Object*

Fig. 6. Architecture of the *HlaManager* attribute

The pre-initialize() method of the `HlaManager` executes some services of the Federation management (e.g. create and join tasks for a federation) and some services of the Declaration management relative to publication and subscription of object's instances in a federation. The initialize() method executes services of the Federation management relative to HLA synchronization points and services of the Time management about the declaration of the federate time policy (e.g. constrained and regulating).

The wrapup() method is related to some services of Declaration management and Federation management corresponding to unpublish/unsubscribe actions, correct termination (e.g. resignation) of federates and the destruction of a federation.

These methods are executed by the director during the lifecyle of the Ptolemy model. Subsection IV-A have showed that the `proposeTime`() method is executed by the director during the time advancement phase. The role of the `updateHlaAttribute`() method is detailed in Subsection IV-C.

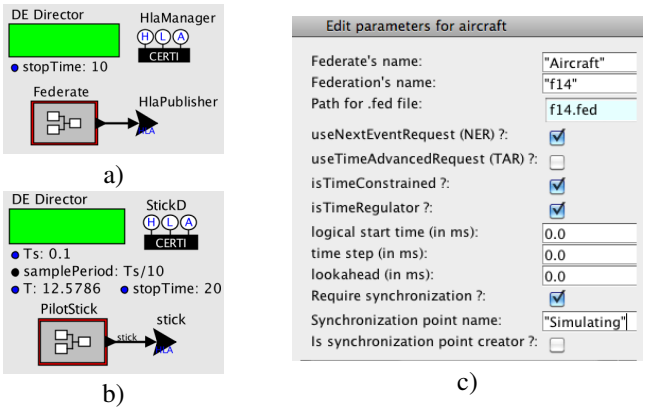


Fig. 7. a) and b) `Ptolemy Federates`, c) Parameters of `HlaManager` attribute

2) Configuration of a Ptolemy Federate: A Ptolemy federate must deploy only one `HlaManager` attribute and one or more `HlaPublisher` or `HlaSusbcriber` as represented in Figure 7. The `HlaManager`'s configuration interface, depicted in Figure 7.c, allows the user to configure and to enable or disable HLA services used by the Ptolemy federate.

According to the HLA standard, a Ptolemy federate needs to specify its name, the name of the federation in which it participates and its corresponding FOM (.fed file). The interface allows to configure the required time management services. The user must choose between NER or TAR services. Time policy (e.g constrained or/and regulator) and timing properties (e.g start time of the HLA logical time, lookahead value, etc.) are also specified through this interface.

In our approach, the HLA synchronization point services synchronize federates during their initialization and ensure the correct starting of Ptolemy federates (if required). This avoids that a Ptolemy federate that only receives values from HLA, finishes its simulation before the launch of the other federates.

C. `HlaPublisher` *and* `HlaSubscriber` *actors*

In Sections II and III we have identified the main communication artifacts for both simulation environments. In Ptolemy, actors communicate with each other by sending events through input and output ports, when HLA is based on updated values of attribute following the publish/subscribe pattern. A HLA attribute owns specific declaration according to the *FOM* and comes with information as its name, its type, the object's class which it belongs to and its time policy (e.g declared as *TIMESTAMP*).

In our approach, an HLA actor is mapped to an HLA object attribute. Two distinct actors `HlaPublisher` and `HlaSubscriber` have been created according to Ptolemy's

actor semantics to publish and to subscribe to an HLA attribute. Each actor specifies a unique port (an input port for the `HlaPublisher` and an output port for the `HlaSubscriber`). The configuration of these actors involves their name, the type of their unique port and the parameter `ObjectHandle` which indicates the object class that the attribute belongs to according to the FOM. The data type of the port has to be the same as in the HLA attribute. Figure 7 a) and b) describe a `HlaPublisher` actor before and after its configuration.

D. Interaction between `HlaManager` *and HLA actors*

1) `HlaPublisher`*:* Let us consider now the methods `updateHlaAttribute`() and `proposeTime`() of the `HlaManager` attribute presented in Figure 6. Figure 8 describes the interaction (through its execution flow), to publish an update of a HLA attribute, between the `director`, the `HlaPublisher` actor, the `HlaManager` attribute and the RTI. The methods proposed by the RTI are provided by the JCERTI API.

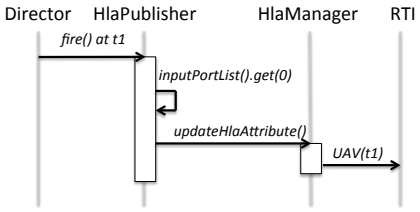


Fig. 8. Execution flow to handle an update of a HLA Attribute

The `updateHlaAttribute`() method is provided by the `HlaManager` to `HlaPublisher` actors. This method encapsulates the operations to build an updated value of a HLA attribute from a Ptolemy event, and calls the Object management services relative to the publication of the updated value (only the call to the `updateAttributeValues`() service is showed in the Figure 8).

The initialize() method (not showed in Figure 8) of the `HlaPublisher` allows to retrieve the reference to the `HlaManager` deployed in the model. Thus the actor can invoke the `updateHlaAttributeValue`() method provided by the manager during the execution of its `fire`() method (triggered by the director). A publication action is enabled by the connection of an output port of a Ptolemy source actor of the core library to the input port of the `HlaPublisher`. To summarize, each reception of an event $e = (v, t)$ on this input port (at time t) leads to the call of the updateAttributeValues() (UAV) service, by the `HlaManager`, with value v and time t. The JCERTI API is used to build a correct representation of the timestamp t.

2) `HlaSubscriber`*:* Figure 9 describes the execution flow to introduce a new update of a HLA attribute, received from the HLA/CERTI Federation. This interaction involves the `director`, the `HlaSubscriber` actor, the `HlaManager` attribute and the RTI.

The `HlaSubscriber` contains a queue and provides the putReflectedAttribute() method to the `HlaManager` to store every updated value in it. When a Federate uses the HLA Time management, the reception of these values (e.g. reception of

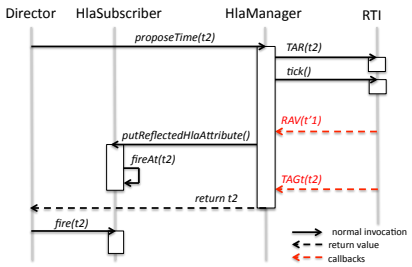


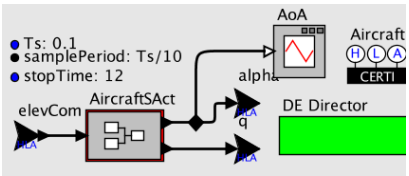
Fig. 9. Execution flow to handle reflected HLA Attribute

the reflectedAttributeValues() (RAV) callbacks from the RTI) is only possible during a time advancement phase, through the call to the HLA tick() (see Figure 9). At the end of this phase each RAV received by the `HlaManager` is stored as a (Ptolemy) event in a `HlaSubscriber`. The manager is able to retrieve the corresponding `HlaSubscriber` and to invoke its putReflectedHlaAttribute(). The value of the event is the updated value and its timestamp is the HLA logical time specified by the RAV. The treatment of the RAV is handled in the `proposeTime`() method.

During the execution of the putReflectedHlaAttribute(), a call to the fireAt() method is performed to program the next firing time of the `HlaSubscriber` (i.e. the time when the director have to call its fire() method). This ensures the delivery of the timed-event at the correct time in the Ptolemy simulation. A subscription operation is enabled by connecting the `HlaSubscriber`'s output port to an input port of a Ptolemy sink actor of the core library.

V. Case Study

Let us consider the hierarchical Ptolemy model of Figure 4, This model is split in three Ptolemy Federates (ptII-Fed for short) represented in Figure 10.a, 7.b and 10.c. Each ptII-Fed has a DE director and is directly obtained from a composite actor `C` by just adding: 1) an `HlaManager` attribute in the model, 2) an `HlaSubscriber` actor to each input port of `C`, and 3) an actor `HlaPublisher` to each output port of `C`. The FOM for this federation is represented in the `f14.fed` file depicted in Figure11.a. As explained in section IV-B, an `HlaManager` needs to be configured. The interface depicted in Figure 7.c is the configuration of the `Aircraft HlaManager`. Due to lack of space, other configurations are omitted here. Besides the Federate name, `AutoPilot` ptII-Fed has the same properties as the `Aircraft`: it uses NER, it is time constrained and time regulator, it has a lookahead of 0 and the synchronization point is the same. `Stick` ptII-Fed has two differences: it uses TAR and is the creator of the synchronization point.



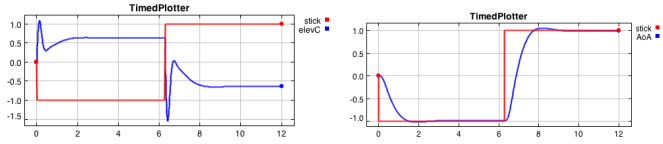
a) `Aircraft` model b) `AutoPilot` model

Fig. 10. Ptolemy Federates

Figure 11.b shows the plot of `elevCom`, the elevator command sent by the `AutoPilot` controller to the `Aircraft` and Figure 11.c shows the angle of attack of the aircraft. Additionally, in both graphs, the Stick output is plotted. The values obtained are the same for the centralized case of Figure 4 and the distributed simulation with the three ptII-Fed.

```
;; f14
(Fed
  (Federation f14)
  (FedVersion v1.3)
  (Federate "aircraft" "Public")
  (Federate "stick" "Public")
  (Federate "autopilot" "Public")
  (Spaces)
  (Objects
    (Class ObjectRoot
      (Attribute privilegeToDelete reliable timestamp)
      (Class RTIprivate)
    (Class myObjectClass
    (Attribute stick RELIABLE TIMESTAMP)
    (Attribute alpha RELIABLE TIMESTAMP)
            (Attribute q RELIABLE TIMESTAMP)
    (Attribute elevCom RELIABLE TIMESTAMP))))
    (Interactions
      (Class InteractionRoot BEST_EFFORT RECEIVE
        (Class RTIprivate BEST_EFFORT RECEIVE))))
```

a) `f14.fed` file



b) Stick and `elevCom` c) Stick and `AoA`

Fig. 11. a) FOM ; b) Elevator command; c) Angle of attack of the aircraft

VI. Conclusion and Future Work

In this paper we propose a co-simulation framework for complex, heterogeneous systems such as encountered in CPS. Topics addressed in this work are different notions of time across distributed components, time synchronization, communication latencies, co-simulation of heterogeneous components, and interoperability of different simulation environments. This approach discusses the distributed simulation capabilities of HLA/CERTI, and describes how the simulation tool Ptolemy was extended in order to interact with HLA. The extensions in Ptolemy, although specific to the tool, can be generalized for arbitrary simulation tools.

The framework presented here allows the design of complex simulations containing several different simulation models as well as simulation tools, functional source code executed on specific embedded platforms and hardware.

The case study, an aircraft longitudinal flight control system, shows the distribution of a Ptolemy model over three federates in an HLA federation. The simulation results obtained by the distributed simulation are equivalent to the ones recorded during the centralized simulation.

Future work directions are numerous. As an immediate next step, we will replace the pilot stick model in Figure 7.b (ptII-Fed `Stick`) with a real joystick controlling the elevator (modeled by ptII-Fed `Aircraft`). Another direction is the integration of a network model as a Ptolemy federate. In fact, Ptolemy II has introduced special entities called quantity managers [3] that model the network behavior allowing to take into account latencies, without changing the functional model. We are working on a bigger case study, a complex flight simulation platform containing many, heterogeneous components, called DSES (Distributed Simulation of Embedded Systems) [16], using Ptolemy federates.

In the prototype presented here we do not account for phenomena such as clock drift or non-deterministic communication. Future work will investigate these topics. With regards to deterministic computation and communication, a model of computation, PTIDES, and the potential integration with HLA, is investigated.

Acknowledgment

This work is a joint work between ISAE/DMIA, ONERA/DTIM and UC Berkeley/EECS. G. Lasnier has been funded by the RTRA-FCS STAE Foundation.

References

[1] MathWorks, "MATLAB/Simulink," 2013. [Online]. Available: http://www.mathworks.com

[2] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91 - 1, pp. 127–144, 2003.

[3] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, pp. 13 – 28, Jan 2012.

[4] IEEE, "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules," *IEEE Std 1516TM-2010*, pp. 1–38, 2010.

[5] ——, "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification," *IEEE Std 1516.1TM-2010*, pp. 1–378, 2010.

[6] E. Noulard, J.-Y. Rousselot, and P. Siron, "CERTI, an open source RTI, why and how ?" *Spring Simulation Interoperability Workshop*, 2009.

[7] ONERA, "The Open Source middleware CERTI," 2013. [Online]. Available: http://www.onera.fr/dtim-en/hla-distributed-simulation/index.php

[8] H. Hadj-Amor and T. Soriano, "A contribution for virtual prototyping of mechatronic systems based on real-time distributed high level architecture," *Journal of computing and information science in engineering*, vol. 12, no. 1, 2012.

[9] Modelica Association, "Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3," Modelica Association, 2012.

[10] ForwardSim Inc. Simulation and Technologies, "HLA Toolbox for MATLAB - HLA Blockset for Simulink," 2013. [Online]. Available: http://www.forwardsim.com

[11] Modelisar, "Functional mock-up interface for model exchange and co-simulation, Version 2.0 beta 4," Information Tech for European Advancement, Tech. Rep., Aug 2012. [Online]. Available: https://www.fmi-standard.org

[12] M. Awais, P. Palensky, A. Elsheikh, E. Widl, and S. Matthias, "The HLA RTI as a master to the functional mock-up interface components," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*, 2013, pp. 315–320.

[13] R. M. Fujimoto, "HLA time management: Design document," Georgia Tech College of Computing, Tech. Rep., Aug 1996.

[14] E. A. Lee, "Heterogeneous actor modeling," in *Proceedings of the 11th International Conference on Embedded Software (EMSOFT 2011)*, 2011, pp. 3–12.

[15] ——, "Modeling concurrent real-time processes using discrete events," *Ann. Software Eng.*, vol. 7, pp. 25–45, 1999.

[16] C. Gervais, J. Chaudron, P. Siron, R. Leconte, and D. Saussie, "Real-time distributed aircraft simulation through hla," in *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*, 2012, pp. 251–254.