Improving the HLA-CERTI framework

Towards a better cosimulation framework for cyber-physical systems

David Côme

Advisers

ISAE-SUPAERO
CARDOSO JANETTE AND PIERRE SIRON

UC BERKELEY
EDWARD LEE

September 18, 2015

This work is submitted to :

- **Institut supérieur de l'aéronautique et de l'espace** for the degree of *diplôme ingénieur Supaero*
- **Université Paul Sabatier** for the degree of *Master recherche Informatique et Télécommunications parcours systèmes répartis et logiciel crititque*

Contents

Acronyms

ADS Automatic Distribution System. 15

DE Discrete Events. 15

FOM Federation Object Model. 21, 25

HLA High Level Architecture. 4

NER Next Event Request. 13, 16, 19

RTI Run Time Infrastructure. 13, 14

SDF Synchronous Dataflow. 15

TAR Time advance Request. 13, 19

TSO Time stamp ordered. 13

Introduction

Cyber-physical systems have gained momentum over the last decades and pose many challenges. On large systems, simulation is often the only way to perform tests and get some properties on the overall system without having the real system. To improve simulation's performances, the simulation is usually distributed.

But building a simulator is not an easy task, one has to be proficient in computer science and in the domain that is being simulated. Building a distributed simulation is even harder, because one has also to take into account all the problems that may arise from the distributed aspect. Doing a distributed simulation for cyber-physical systems where simulation methods and tools may change from one component to another is nearly impossible.

High Level Architecture (HLA) (see section 2.2) is a middleware commonly used for easing the creation of distributed simulations. It is a first good step and tackles several difficult issues, but one still needs to have a good understanding of it to have a *working* distributed simulation.

The aim of the HLA-CERTI cosimulation framework is to simplify the creation of distributed simulations by releasing the user of deep understanding of HLA's key concepts and providing him with an user-friendly GUI for designing federate through the Ptolemy software (see section 2.1).

Gilles Lasnier wrote in 2012 the original framework in a joint work between UC Berkeley and ISAE Supaero. Even if it was functional, it lacked of several features. Firstly, it lacked of a formal semantics for the actors : section 4 tackles that subject. Then, a model could not distinguish several objects from each other. A project in 2014 at ISAE-Supaero made that possible but it requires that each model has to be tuned manually from one simulation to the other : section 5 shows a solution for that issue. Finally several other improvements and modeling issues are described in sections 6 and 7.

1. Context

1.1. UC Berkeley

The *University of California, Berkeley* is the main campus of the University of California system. Located in Berkeley, the university was founded in 1868 as the merge between the College of California and the public Agricultural, Mining, and Mechanical Arts College in Oakland. Nowadays, the university hosts more than 35 000 students (10 000 postgraduates) and is ranked among the top 5 universities in the world. It ranked fourth in the 2015 ARWU ranking and is the first public school. Among UC Berkeley's alumni, there are more than 65 Nobel prizes, 19 Oscars et 11 Pulitzer prizes.

Berkeley is also famous for the student activism in the 1960s : the Free Speech Movement started there in 1964 and was the initial spark for the hippie movement.

1.1.1. Electrical engineering and computer science department (EECS)

With more than 120 faculty members, 2500 undergraduate students and over 600 graduate students, the EECS department is the largest one at UC Berkeley. The department has had a tremendous impact on computing science and electrical engineering over the last decades. It was the home of many world wide famous projects and still host a wide variety of research projects.

Among the most famous projects, there are :

- SPICE, the original simulator for analogous circuits
- The original BSD operating system, ancestor of many current operating systems
- The RISC architecture for CPU
- The RAID technology for hard drives.

Without Berkeley, today's world would not be as it is.

1.1.2. Center for Hybrid and Embedded Software Systems (CHESS)

CHESS's mission is to provide an environment for cutting edge research in cyber-physical systems. The center is building foundational theories and practical tools for systems that combine computation, networking abilities, and some physical dynamics. This include model-based and tool-supported design methodologies for systems often need to be fault tolerant on heterogeneous distributed platforms. CHESS provides industry with innovative software methods, design methodology and tools while helping industry solve real-world problems.

1.2. Overall motivation : cyber-physical systems

Cyber-physical systems (CPS) are integrations of computation with some physical processes [28]. Usually, the embedded controllers will control and monitor the physical processes (through the actuators) while a feedback loop (through the sensors) will allow the physical processes to affect the computations. Today, one can found CPS in many domains such as spatial, avionics, automotive, transportation or more recently health care or energy. Most of the projects are usually worth millions and put lives at stake. Most of the CPS are distributed systems with real time constraints and requirements for fault tolerances mechanisms.

Designing a such system 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.The figure 1 summarizes in an elegant way the CPS design complexities through a concept map. Due to the inherent complexity of CPS and to the expertise needed for a single component, the different parts are developed as least by different engineering teams or even by external contractors if needed.

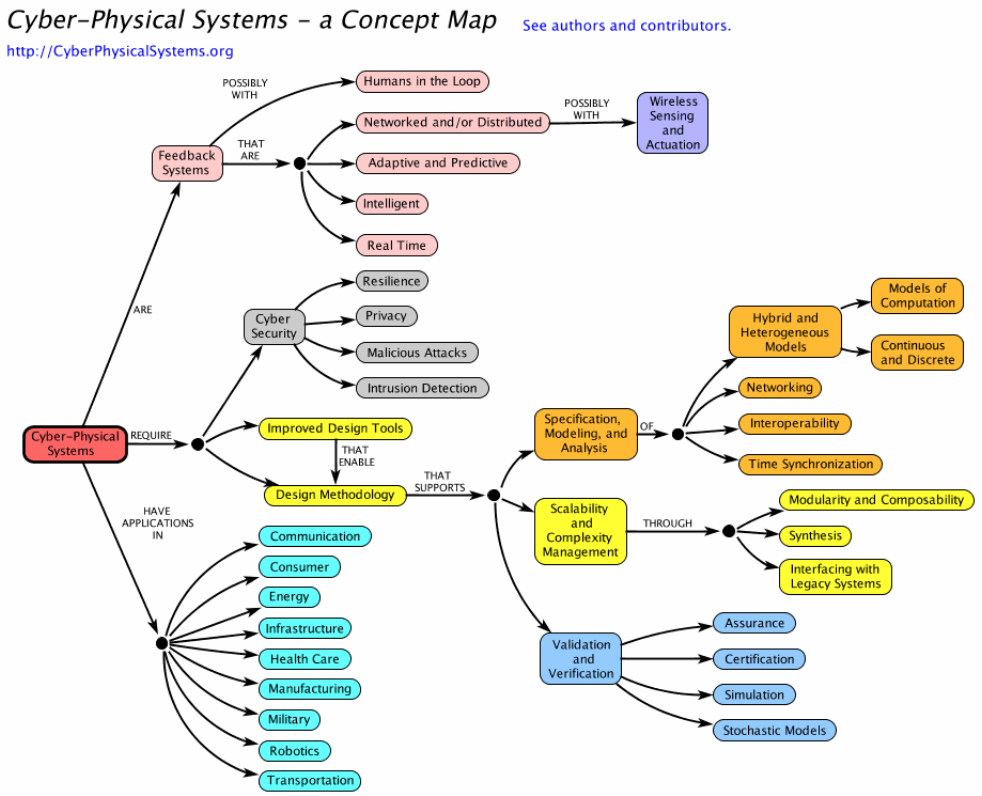


Figure 1: Concept map for CPS (from http://cyberphysicalsystems.org/)

Integrating all the parts and evaluating the resulting behavior is challenging. Over the last decade, model-based tools and model-based engineering have gained momentum in the design of CPS [34]. With them, the modeling process is done at a higher level of abstraction and they facilitate the communication between different teams of engineers. They are also widely used for doing code generation and prove several properties of the model. However, different parts of the CPS usually requires different abstractions and different tools (e.g. continuous or discrete models, timed or untimed properties). The lack of interoperability between the tools is a major challenge.

Analyzing the worst case execution time and the schedulability is needed but not enough 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 behavior of high level specifications mixed with more low level elements.

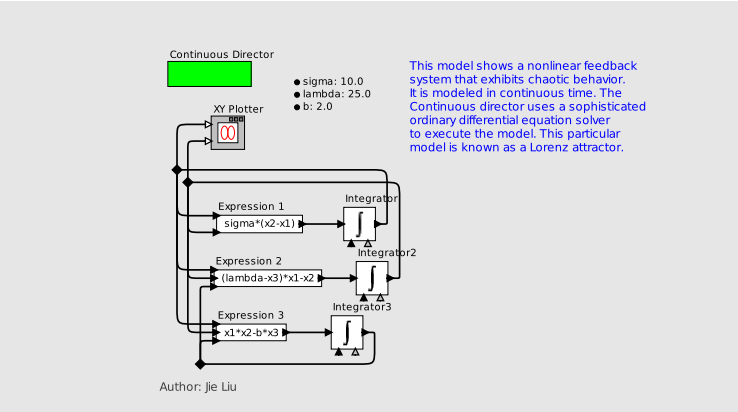
2. Technical context

This sections aims to give the reader some knowledge on the technologies used in the project.

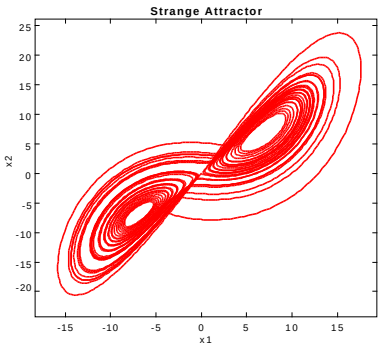
2.1. Ptolemy project

2.1.1. Definition

The Ptolemy project, led by Prof E. Lee, studies the modeling, simulation, and design of cyber physical systems. A open source software, called Ptolemy II [1] has been developed for the last 20 years and is focused on providing modeling and simulation tools for heterogeneous systems. The simulation is actor orientated.



(a) A Ptolemy model for Lorenz attractor



(b) Plot from the simulation

Figure 2: A Ptolemy model and its execution output

The emphasis in Ptolemy II is put on assembly of concurrent components, as they are in the real system. A key aspect of the software is the use of models of computation. A model of computation is a set of formal rules that describes what kind of actors can be used inside a model, how they communicate and in which order they are executed. That enables the composition of distinct models of computation with a well-defined behavior. Thus, this tool is well suited for modeling CPS by providing different models of computation that are widely used within such systems.

Ptolemy II's core is a set of Java classes and packages. This core provides support for executing the abstract syntax which describes the models. It also enables the composition and connection of entities in hierarchical way. A graphical editor is also provided for editing the abstract syntax. A derivative format from XML (called MoML) provides a persistent way to store the models. Another key part is the tagged signal model with super-dense time [30], [32]. As mentioned earlier, in Ptolemy each model of computation is described by a formal semantics. Overtime, these semantics have been described in many papers, see [31],[30] or [27].

2.1.2. Ptolemy's semantics

The following section aims at giving the reader a quick overview of the formalism used in Ptolemy. Those results are originally from [35].

Why semantics matters The purpose of a formal semantics is to formally describe the intended behavior in order to have a reference that we can compare to the simulation's run.

Without it, the reference is the code and it could be hard to determine if some behavior is correct or not. With it, we can also prove some properties on the system, such as causality or correctness.

Valuation A valuation over a set X is a function $x : S \to \mathscr{U}$ that assigns to each variable $v \in S$ some value $x(v) \in \mathscr{U}$. The set of all assignments over X is denoted by $\hat{X}$.

Within $\mathscr{U}$, there are two special values : `absent` and $\perp$. `absent` represents the absence of a signal at a particular point in time whereas $\perp$ represents unknown, faulty or undefinable values. Note that `absent` is a perfectly legal value for a signal but unknown signals should only be used a temporary value during the computation. Any signal that is still unknown at the end of a computation will result in an ill formed model.

We use the following notations for valuations :

Creating a valuation A new valuation is denoted by listing the assignments for all variables in S. For example, if $S = \{v_1, v_2\}$ and $u_1, u_2 \in U$, then $\{v_1 \mapsto u_1, v_2 \mapsto u_2\}$ denotes the valuation $x \in \hat{S}$ such as $x(v_1) = u_1$ and $x(v_2) = u_2$.

Changing a value If $x \in \hat{X}$, $v \in X$, $u \in U$ then $\{x|v \mapsto u\}$ denotes the new valuation obtained from x by setting v to u and leaving all other variables unchanged.

Concatenation Let be S_1 and S_2 two disjoint sets of variables, $x_1 \in \hat{S_1}, x_2 \in \hat{S_2}$ two valuations. Then we define the concatenation of s_1 and s_2 by $x \in \widehat{S_1 \cup S_2}$ such has

$$\begin{cases} \forall v \in S_1 & x(v) = x_1(v) \\ \forall v \in S_2 & x(v) = x_2(v) \end{cases} \tag{1}$$

x is also written (x_1, x_2).

Restriction For $S' \subseteq S$ and $x \in \hat{S}$, we define the restriction of x to $S', x \restriction_{S'},$ such as $\forall v \in S', x \restriction_{S'} (v) = x(v)$.

Atomic actor's semantics This section will describe the semantics for an atomic actor only.

An atomic actor is the most fundamental actor that can be used inside a model. More complex actors (called composite actors), can be built by composing them in a model with a director (see 2.1.2).

Description Formally, an atomic actor A is a tuple

$$A = (I, O, S, s_0, F, P, D, T) \tag{2}$$

where I is a set of input variables, O is a set of output variables, S is a set of state variables, $s_0 \in \hat{S}$ is a valuation over S representing the initial state. Note that any of the sets of variables I, O or S may be empty or infinite.

By convention, the set of valuations over an empty set of variables is a singleton, we will denote by $*$. The input and output spaces can also be infinite. The terms input, output, state mean valuations respectively over I, O, S.

F, P, D, T are total functions with the following types:

$$F : \hat{S} \times \hat{I} \to \hat{O} \tag{3}$$
$$P : \hat{S} \times \hat{I} \to \hat{S} \tag{4}$$
$$D : \hat{S} \times \hat{I} \to \mathbb{R}_+^\infty \tag{5}$$
$$T : \hat{S} \times \hat{I} \times \mathbb{R}_+^\infty \to \hat{S} \tag{6}$$

The F, P, D and T functions are called the fire, postfire, deadline and time-update functions of A, respectively:

- F and P are similar to the output and transition functions of a state machine
- F produces an output given a state and an input
- P produces a new state, given the same information as F
- D returns a deadline, indicating how much time the actor is willing to let elapse.
- T updates the state given information on the actual delay chosen by the environment

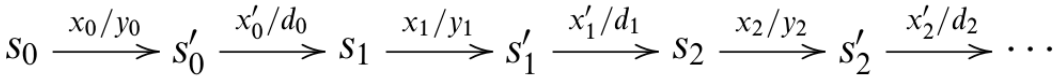


Figure 3: Visual explanation of timed behaviors

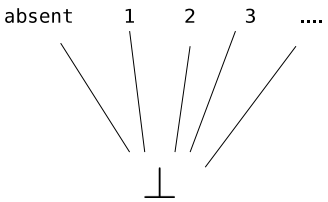


Figure 4: Flat CPO used

Timed behaviors An actor A defined as above generates a set of behaviors over time. A timed behaviour of A is a sequence (s_i, s'_i) as shown on figure 3 with

$$(s_i, s'_i) \in \hat{S} \tag{7}$$

$$y_i = F(s_i, x_i) \tag{8}$$

$$s'_i = P(s_i, x_i) \tag{9}$$

$$d_i \leq D(s'_i, x'_i) \tag{10}$$

$$s_{i+1} = T(s'_i, x'_i, d_i) \tag{11}$$

At a given time t, the actor A is in state s_i. The environment provides some input x_i. It produces the output y_i without changing its state. Then, the environment calls P and A moves to a temporary state s'_i. P receives the same input as F. The environment asks how much time it is willing to let elapse, given x'_i, an estimate for the input variables next values. Finally A computes its new state s_{i+1} knowing the amount of time d_i that has effectively elapsed. The new time is now $t + d_i$ and the execution process starts over.

Complete partial order

A complete partial order (CPO) is set with an *partial* order $\lhd$ such as each *totally ordered subset* has an upper bound with respect to $\lhd$. Ptolemy uses a special kind of CPO called flat CPO. Such CPO have properties

- There is an element $\perp$ that is smaller than any other element
- All other element are not comparable

Figure 4 shows a such CPO for integers with an additional `absent` value. In flat CPO, every ordered subset has at most two elements and has consequently an upper bound. Over a flat CPO, a monotonic function is (Scott) continuous[1] and the Kleene fixed-point theorem ensures it has at least one fixed point.

For more on CPOs, fixed point theorem, see [31] and the first chapter of [29].

[1] This is false for non flat CPO

Directors This sections aims at given a overall idea on how directors work and give an detailed example of the discrete events director. The semantics for the other directors can be found in [35].

How works a director A director is a composition actor. From a given set of actors, it will return a new actor that behaves like an atomic actor but whose output depends on the initial set of actors.

The dependency relation between the output and the initial set of actors defines the director's type. In particular, they choose when to call the different functions of the actor interface and they also manage the data exchanges between the actors. Therefore, the same set of actors with two different directors may have different timed behaviors. We usually say that directors implement the model of concurrency and communication.

Discrete events director semantics Let's $H = (A_1, \ldots, A_n)$ be a set of atomic actors (with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$) defined like in 2.1.2

We introduce the following sets :

$$V = \bigcup_{i=1}^{n} O_i \tag{12}$$

$$I = \bigcup_{i=1}^{n} I_i \setminus V \tag{13}$$

$$O = V \setminus \bigcup_{i=1}^{n} I_i \tag{14}$$

$$S = \bigcup_{i=1}^{n} S_i \tag{15}$$

$$s_0 = (s_{0,1}, \ldots, s_{0,n}) \tag{16}$$

V is an intermediate set for listing all outputs variables, I is the set of input variables that are not linked to an input port while O is the set of output variables that are not connected to input variable.

Then the discrete events director is

$$DE(H) = (I, O, S, s_0, F, P, D, T) \tag{17}$$

To define its functions, let's be $s = (s_1, \ldots, s_n)$, $s_i \in \hat{S}_i$ a state and $x \in \hat{I}$ be an input of $DE(H)$. We define the function $\tilde{F}_{s,x} : \hat{V} \to \hat{V}$ such has

$$\forall y \in \hat{V}, \forall j \in \{1..n\}, \forall o \in O_j,\ (\tilde{F}_{s,x}(y))(o) = F_j(s_j, (x, y) \restriction_{I_j})(o). \tag{18}$$

The above definition states that to compute the value of a given output variable o of A_j, function $\hat{F}_{s,x}$ uses the fire function of A_j , which is, F_j . This function takes as its inputs the local state s_j of A_j and its local inputs, which are the ones in $(x, y) \restriction_{I_j}$.

Every F_j is designed to be monotonic over a flat CPO over $\mathscr{U}$, consequently the $\hat{F}_{s,x}$ function is monotonic and so continuous. Thus, as explained before, $\hat{F}_{s,x}$ has a least fixed point. Let's $y^*_{s,x}$ be that fix point. Then we can define :

$$F(s, x) = y^*_{s,x} \restriction_O \tag{19}$$

$$P(s, x) = (P_1(s_1, (x, y^*_{s,x}) \restriction_{I_1}), \ldots, P_n(s_n, (x, y^*_{s,x}) \restriction_{I_n})) \tag{20}$$

$$D(s,x) = \min(D_1(s_j, (x, y^*_{s,x}) \restriction_{I_j}), j = 1..n) \tag{21}$$

$$T(s,x,d) = (T_1(s_1, (x, y^*_{s,x}) \restriction_{I_1}, d), \ldots, T_n(s_n, (x, y^*_{s,x}) \restriction_{I_n}, d)) \tag{22}$$

Those functions define the formal semantics for the DE Director : events are processed in time stamp order.

2.1.3. Ptolemy's domains

As explained above, a director is the key component that rules over a model and give an overall meaning to it.

The main domains in Ptolemy (other than DE) are describe below :

- *Dataflow*, a domain where a stream of data flows through different actors without any notion of time. It is often called pipe and filter system.
- *Synchronous Reactive* is close to the Dataflow but there is here a notion of time. It is useful for describing logically timed systems.
- *Process network* is a domain where actors are processes exchanging messages with a blocking read and asynchonous write. That way, the actors can be processed concurrently but in a deterministic manner.
- The *Continuous Time* domain is used when the time must be seen as a continuum.
- A *Modal Model* is a system with a finite set of behaviors and defined transitions between them.

There more specialized domains, such as for quantized state systems, petri nets or cellular Automata for instance.

2.2. HLA

2.2.1. Definition

HLA is standard for interoperability between distributed simulations. The driving idea beyond HLA is it is simpler and more cost efficient to build several small simulators distributed over a network than a monolithic simulator on a supercomputer. With several small simulators, the computing power needed for a single machine is lower, and a single simulator can be more easily reused from one simulation to another.

Initiated by the *Department of Defense*, it became an IEEE's norm. HLA services are officially divided into seven service groups but we can we categorize them into 3 broader categories :

- Data exchange
- Time management
- Information and synchronization

The first norm was released in 1998 [16], the following in 2000 [21] and the latest in 2010 [22].

Wording HLA has some specific vocabulary. A simulator is called a *federate*. A set of federate is a *federation*. They exchange data through a software bus called *Run Time Infrastructure*. Figure 5 sums it up.

2.2.2. Simulation architecture and data exchange

They are usually two ways to integrate different services together. The first one is a pair-wise integration. With that approach, each simulator knows about all the other simulators which it needs to know about to work. This works well for a few systems but quickly becomes rather complicated for more. And adding a new simulator may disrupt the existing architecture.

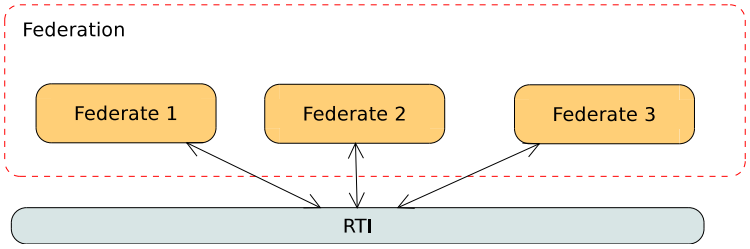
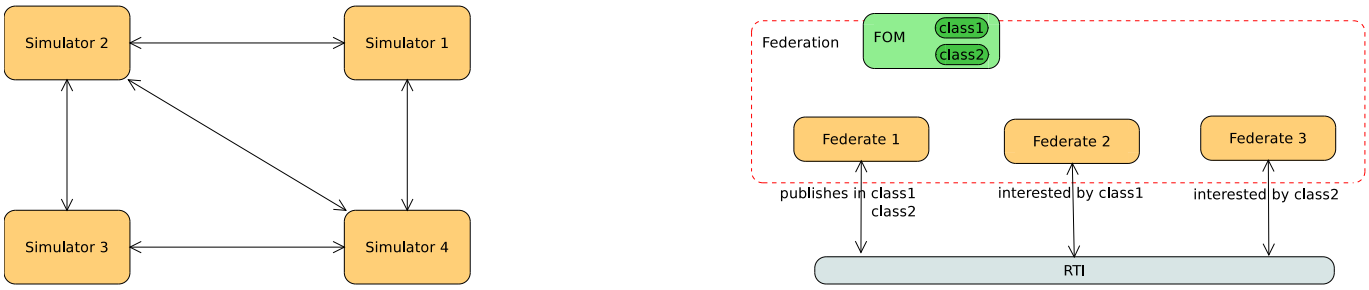


Figure 5: HLA wording



(a) Example of a pair-wise architecture

(b) Service approach from HLA

Figure 6: Different simulation architectures

HLA uses a service bus approach. For a given simulation, there is a file[2] that describes what can be exchanged between the simulators within the simulation. Then each federate states what il will publish and subscribe to. The data exchange is done by the RTI in anonymous publisher-subscriber manner. A federate does not know to whom it is giving data, neither where come the data it is receiving. This is more flexible and scalable. Figure 6 explains the two approaches.

2.2.3. Time management

Time in simulation Time in simulations can be confusing because there are three timelines involved.

- The *physical time* which refers to the time of the system being simulated
- The *simulation time*, which is how the simulator represents the physical time.
- The *wallclock time* which refers to time when the simulator is executed

For instance, we may want to simulate a chemical reaction for 10 seconds (physical time) and have simulation time unit of 1 micro second, then simulation time will go up to 10^7. That simulation could take one hour to complete on the wall clock time.

Time in distributed simulations

Causality In a distributed context, time management is a key feature, because there are several simulators involved and there is no such thing as a unique clock[3]. Among a federation, each federate refers to the same time line but with its own clock.

[2] Called fed file

[3] Even if some projects with specific conditions, such as CERN's White Rabbit, achieved a clock synchronization below the nanosecond.

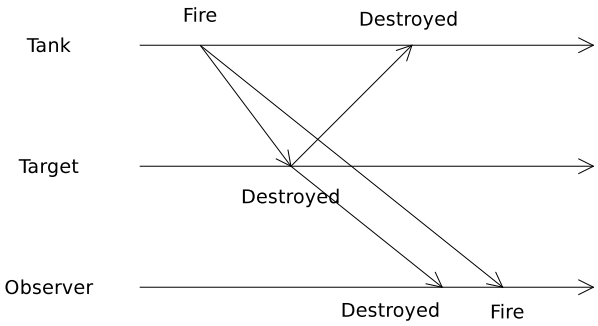


Figure 7: A simulation where causality is violated. For the observer, the target is destroyed before the tank has fired

We want the simulation to be causal: on every federate, a message must arrive before any another event that it might have triggered on another federate. Figure 7 shows a simulation which is not causal, a tank fires on a target, but according to the observer, the target is destroyed before the tank fired.

Ensuring causality To ensure causality, a time stamp (in simulation time) is assigned to each event and each federate processes the incoming messages in time stamp order. For advancing in time, a federate must ask for the permission to the Run Time Infrastructure (RTI). The RTI will grant that access if all time stamped messages with a lower time stamp than the given time have been delivered.

The granting process relies on two mechanisms :

- Time regulation aspects
- The lookahead

The lookahead is the time needed by a federate to produce an output after receiving a message. That means if a federate has lookahead of L and its local time is T, the earliest message it can produce is at $T + L$.

The time regulation aspects are *time regulating* and *time constraint*. A federate can be time regulating, time constrained, both or neither. If a federate is not time constrained, the RTI will grant him unconditional time advances. If a federate is not time regulating, its messages are timestampless.

Lower bound on time stamps A federate must determine if it can safely process the next queued message (what was called *deliver* process on the above section). That requires the knowledge of time stamps from messages which will arrive in the future, which is obviously impossible. Instead, it computes a lower bound on time stamps, which is a lower bound on time stamps of Time stamp ordered (TSO) messages that could later be placed into the TSO queue for a given federate. Thus, the RTI ensures logical time of federate i will never exceeds $LBTS_i$. Messages with a time stamp lower than the LBTS can be safely processed.

If T_j is the federate j's current time and L_j its lookahead, then federate i's lower bound on time stamps $LBTS_i$ is

$$LBTS_i = \min_{i \neq j}(T_j + L_j) \tag{23}$$

Some algorithms to compute the LBTS can be found in [18].

Time advance mechanisms There are two main mechanisms for advancing in time

- Next Event Request (NER) : the federate is event driven
- Time advance Request (TAR) : the federate is time steped

Both needs the lookahead to be greater than zero[4].

The Next Event Request (up to time T) requests the federate's logical time be advanced to the time stamp of the next incoming TSO message, or to time T if there is not such a message. If the next incoming TSO message is not unique, then they all are delivered.

The Time Advance Request (to time T) requests the federate's logical time be advanced to T. All messages up to T are delivered to the federate. The delivery is done in time stamp order.

2.2.4. RTI providers

As a standard, HLA only describes the components, the expected behaviors and properties. There are many implementations of the HLA standard, from commercial companies, governments, academia or open source developers [4]. This work has been using CERTI ([3]), an RTI implementation provided by the Onera.

3. Related work

Simulation has always been a active domain, bellow there are listed some other works and how they are related to this project.

3.1. Functional Mockup Interface

The functional mockup interface [7] is standard which is gaining momentum in the simulation community. The idea is that any product is a combination of more basic components, each of which is following a set of physical rules. Then it should be possible to build a virtual representation of the overall system by assembling basic virtual components. FMI is mainly used for model exchange and for co-simulation.

The FMI for Model Exchange interface defines an interface in C programming language to the model of a dynamic system. With that interface a different simulation environment should be able to evaluate these equations and thus simulate the component. This component can be described by differential, algebraic and discrete-time equations.

With the co-simulation aspect, the intention is to couple two simulating tools in an overall environment. Each subsystem is still independent and its equations are solved by its own tool. The data exchange between the different subsystems is restricted to discrete communication points. The overall consistency is ensured by some master algorithm that are left to the user to implement.

FMI for co-simulation and HLA have close goals but they are not exactly the same. They both bring together several simulator in order to build a larger simulation. But HLA also tackles that *master algorithm* and all the related aspects through the time advance mechanisms and the conditional delivery of timestamped messages, whereas all of this has to be done for FMI. Some people suggest we could reuse HLA's RTI as master algorithm in a FMI co-simulation [9].

[4]There are two other mechanisms called TARA and NERA when the lookahead is zero. We do not discuss them here.

3.2. Other efforts for distributed simulation

3.2.1. Easing HLA distributed simulation

ForwardSim [5] enables Matlab/Simulink simulations to operate within a federation. But the product is closed-source and licenses can be expensive. Another HLA framework for Matlab can be found in [36]. Unlike these two projects, this project provides an open source HLA framework for the Ptolemy II simulation software.

[20] introduces a simulation framework named *Transparent_DS* (TDS), which enables the use of distributed environments while making affordable the development of agent-based simulators upon JADE [6] (a Multi-Agent System platform) with a HLA back end.

3.2.2. Easing distributed simulation for Ptolemy

There have been some experiments for doing some distributed simulation with Ptolemy.

Automatic Distribution System (ADS) [15] introduces a platform called ADS which allows a model designer to ignore distribution concerns while greatly benefiting from its advantages. ADS describes an architecture that deals with mapping of processes, communication and synchronization in a manner transparent to the user. That work has only been used with the Synchronous Dataflow (SDF) domain. On the opposite, this work focus on enabling several simulator to communicate and exchange data. The end user has to distribute the different components on the nodes and configure them to be in the same simulation. Moreover, the current framework mainly uses the Discrete Events (DE) domain.

Distributed discrete-event (DDE) The DDE domain [11] provides distributed simulation for Ptolemy and uses similar algorithms to the ones used by CERTI (Chandy and Misra). But DDE serves as a framework for studying distributed discrete event systems, it has no intends for improving execution speed.

The current framework implements a layer of compatibility for Ptolemy with a HLA, a standard which was designed for tackling specific issues (performances, reusability). Thus the end goals are not the same and the framework has broader use cases. Moreover, the DDE is a new model of computation and is more intrusive than our framework, which relies on a handful of actors which can be embedded in the DE domain.

3.2.3. Other work with HLA and Ptolemy

Another work presented in [10] uses HLA and Ptolemy for distributed simulation. To do so, they introduced two new actors *SlaveFederateActor* and *MasterFederateActor* with a new domain lead by a *HLADEDirector*. The master is responsible for the initialization of the simulation, and then it stops the simulation until all the slaves begin and the simulation is released. This director is responsible for coordinating the order execution of the actors within the domain, checking when to send data received from RTI to actor, and also from actor to RTI. Unfortunately, the work is not available for others to test.

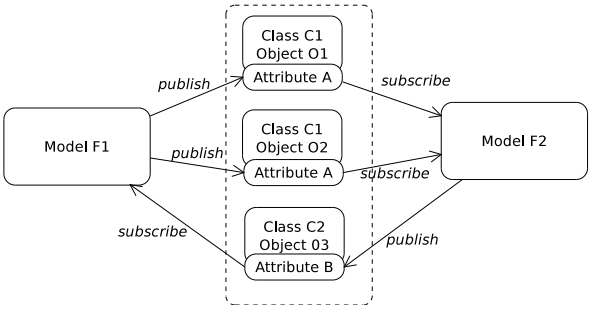


Figure 8: Explanation of the HLA inputs/outputs sets

4. Semantics of the HLA/Ptolemy framework's actors

This section aims at giving a formal semantics for the framework's main components. As explained before, this is intended to be the reference behavior for the actors and can be used to understand more easily the framework's code and the algorithms involved.

4.1. HLA's actors' semantics

In this section, we will assume that there is a global parameter called EventBased which is true if the federate is using the NER mechanism. This parameter is constant during the execution and can be freely used by the actors.

4.1.1. Accessing HLA variables

In the following sections, space variables that have a meaning from a HLA point of view will be written with a different font, such as $\mathscr{c}$ or $\mathscr{a}$.

The set $I_{HLA,F}$ denotes all the variables that the federate F can subscribe to while $O_{HLA,F}$ denotes the one he can update. An HLA variable is made up of two components : an object handle $\mathscr{o}$ and an attribute handle $\mathscr{a}$. The set of all object handles is O_h and the set of attribute handles is A_h. These sets are established once per model[5].

We introduce the functions $\mathscr{N}_{r,\mathscr{F}}$ and $\mathscr{N}_{w,\mathscr{F}}$ for accessing the HLA variables defined above.

$$\mathscr{N}_{r,\mathscr{F}} \begin{cases} O_h \times A_h \to I_{HLA,F} \\ \mathscr{o}, \mathscr{a} \mapsto (o, a) \end{cases} \tag{24}$$

$\mathscr{N}_{w,\mathscr{F}} : O_h \times A_h \to O_{HLA,F}$ has the same definition. We will omit the federate F in the following paragraphs.

The RTI ensures that each element in O_h is unique thus $\mathscr{N}_r$ and $\mathscr{N}_w$ are one-to-one functions. Both functions are not total functions because not all combinations are valid (see example below)[6].

Let's explain this on situation described on the figure 8. There are two models *F1* and *F2*, each being a federate. There are 2 classes *C1* and *C2*. *C1* has an attribute *A* and two instances *O1* and *02*. *C2* has an attribute *B* and one instance *O3*. *F1* stated that it publishes *C1' A* and has subscribed to *C2' B* whereas *F2* did the opposite.

Then the sets are the following :

[5] when the model joins the federation

[6] Using a invalid combination would yield an exception in the code.

O_h	$O1, O2, O3$
I_h	A, B

Federate	Set / value	
F1	I_{HLA}	$(O3, B)$
	O_{HLA}	$(O1, A), (O2, A)$
F2	I_{HLA}	$(O1, A), (O2, A)$
	O_{HLA}	$(O3, B)$

We introduce two tagged signals

- $UAV : O_{HLA} \times \mathbb{R}^+ \to \mathscr{U}$
- $RAV : I_{HLA} \times \mathbb{R}^+ \to \mathscr{U}$

which denote messages send to (respectively received from) the HLA variables O_{HLA} (respectively I_{HLA}) through the HLA primitives UAV (respectively RAV). $\mathscr{U}$ is the set of values which can be exchanged through HLA augmented with the `absent` value and the $\perp$ value.

The two signals are linked by the following relation :

$$\forall e \in I_{HLA} \cap O_{HLA}, \forall t,\, RAV(e,t) = UAV(e,t) \tag{25}$$

This simply states for all attribute which is published and subscribed, then at all time, all messages sent must be forwarded to the federates which have subscribed to it and that any incoming message comes from another federate.

4.1.2. HLASubscriber

HLASubscriber is one side of the link between the model and the HLA federation. Its job is to bring into the model events from a given attribute $\mathscr{a}$ of a given object $\mathscr{o}$ from a given class $\mathscr{c}$. To do so, it needs to state it will subscribe to the pair $(\mathscr{c}, \mathscr{a})$. $\mathscr{c}$ and $\mathscr{a}$ are set up by the end user whereas the RTI will give $\mathscr{o}$ the model. It is a source actor, it has no input and a single output e. Space variable t tracks time.

It is formally described by

$$\text{HLASubscriber} = (\varnothing, \{e\}, \{t\}, s_0, F, P, D, T) \tag{26}$$

Its functions are :

$$F(s,x) = \begin{cases} \{e \mapsto RAV(\mathscr{M}_r(\mathscr{o}, \mathscr{a}), t)\} & \text{if EventBased = true} \\ \bigcup\limits_{(k-1)\cdot T_s \le t \le k \cdot T_s} \{e \mapsto RAV(\mathscr{M}_r(\mathscr{o}, \mathscr{a}), t)\} & \text{if EventBased = false and } \frac{t}{T_s} \in \mathbb{N} \\ \{e \mapsto \texttt{absent}\} & \text{otherwise} \end{cases}$$

with $k = \lfloor \frac{t}{T_s} \rfloor$ where $\lfloor\ \rfloor$ denotes the floor function.

$$P(s,x) = s \tag{27}$$

$$D(s,x) = +\infty \tag{28}$$

$$T(s,x,d) = \{t \mapsto t + d\} \tag{29}$$

The semantic is different if we used it in the $EvenBased$ mode or not.

In case of $EventBased$, the actor will produce a token on the output channel for each RAV message with the appropriate time.

If the federate is not event based, it is time stepped[7]. This means it will ask the RTI for time $k \cdot T_s$, with k an integer (see HlaManager's semantics in 4.1.4). On time $k \cdot T_s$, the idea is to send on the output channel all the messages that have been received between $(k-1) \cdot T_s$ and $k \cdot T_s$ and time stamp them with the current time. Messages' order is preserved. If the actor is fired on time where there is no value for $\mathcal{N}_r(\mathscr{o}, \mathscr{a})$, then the `absent` value is produced. More details about time stepped federates can be found in [37]

4.1.3. HLAPublisher

A *HLASubscriber* is on the other side of the link between the model and the HLA federation. Its job is to register several objects in the federation and then send updates values which have been computed in the model. The federate can only send updates to given object, attribute pair $(\mathscr{o}, \mathscr{a})$ if it has acquired ownership on it. That requires the federate to previously state it will publish updates for that pair. It needs to know the federate's lookahead L and the amount of time t that has passed to send the update with the right time stamp. It is a sink actor, it has no output and several inputs. An given input is dedicated for a given object. In the state space the $\mathscr{o}_j$ are the objects' handles. It is formally described by :

$$\text{HLAPublisher} = (\{i_{1 \leq j \leq n}\}, \varnothing, \{\mathscr{o}_{1 \leq j \leq n}, t\}, s_0, F, P, D, T) \tag{30}$$

Its functions are

$$F(s,x) = \begin{cases} UAV(\mathcal{N}_w(\mathscr{o}_j, \mathscr{a}), t + L) \mapsto i_j\} & \text{if EventBased = true} \\ UAV(\mathcal{N}_w(\mathscr{o}_j, \mathscr{a}), \max(kT_s + L, t)) \mapsto i_j\} & \text{if EventBased = false} \end{cases}$$

with k defined as in the *HLAPublisher* paragraph

$$P(s,x) = s \tag{31}$$

$$D(s,x) = +\infty \tag{32}$$

$$T(s,x,d) = \{t \mapsto t + d\} \tag{33}$$

with k defined as in the *HLAPublisher* paragraph.

An input at this current time will only be send after adding the lookahead in case of NER. In case of TAR, the idea is the RTI has granted the federate the time kT_s and thus it cannot send a message until $kT_s + L$. So we pick the max between the current time and that time. See [37] for more on that.

From a HLA point of view, the federate will have the ownership on the (o_i, a). The list o_i is built when registering all connected actor to the *HLAPublisher* as objects in the federation. See [14] for more details.

[7] It could be neither of them but that case is not relevant for that analysis

4.1.4. HLAManager

The *HLAManager* is a special actor. It is not here to process messages, but to ensure the model's time is consistent with the RTI's time. Thus, it has no input nor output. It is parameterized by two strings: the name of federate it will register and the name of federation it will join. It will try to create the federation if it does not exist and join it afterward.

It can be described by the following tuple :

$$\text{HLAManager} = \{\varnothing, \varnothing, \{t, T_s\}, s_0, F, P, D, T\} \tag{34}$$

Its state space is

- T_s, the time step used when EventBased is false
- The current time t

Given that time flows in the HLA simulation, the actor has to keep trace of it to return the adequate value for the deadline. T_s is set by the user.

Its functions are defined as followed :

$$F(s, x) = * \tag{35}$$

$$P(s, x) = s \tag{36}$$

$$D(s) = \begin{cases} (\lfloor \frac{t}{T_s} \rfloor + 1) \cdot T_s - t & \text{if EvenBased = false} \\ \min(\text{LBTS}, t_m) - t & \text{if EvenBased = true} \end{cases} \tag{37}$$

$$\text{with } t_m = \begin{cases} \min(\text{timestamp of all incoming messages}) & \text{if any} \\ +\infty & \text{otherwise} \end{cases}$$

$$T(s, d) = \{s | t \mapsto t + d\} \tag{38}$$

With a NER, the main idea is to not advance the model's time beyond the next potential HLA event, which is the minimum between federate' LBTS or the timestamp of its next incoming TSO message. With a TAR, the model simply states that the time can not advance beyond the next multiple of T_s.

4.2. Improvement

We introduce a choice on the time advance mechanism to use. A given federate, for a given run, is either event based or time stepped. This separation does not exist in HLA. The standard says a federate can alternatively use event based or the time stepping mechanism. This could be improved to be more compliant towards the HLA standards.

5. Scalability

One of HLA's end-goals is to have reusable simulators that can be used from one simulation to another. This requires to make no assumptions on the number of objects a simulator will have to handle. Whereas this is not an issue with classical programming languages such as C++ or Java which can use dynamic memory allocation, there is no such thing within Ptolemy models. Thus, there is a mandatory scaling phase : the end-user will have

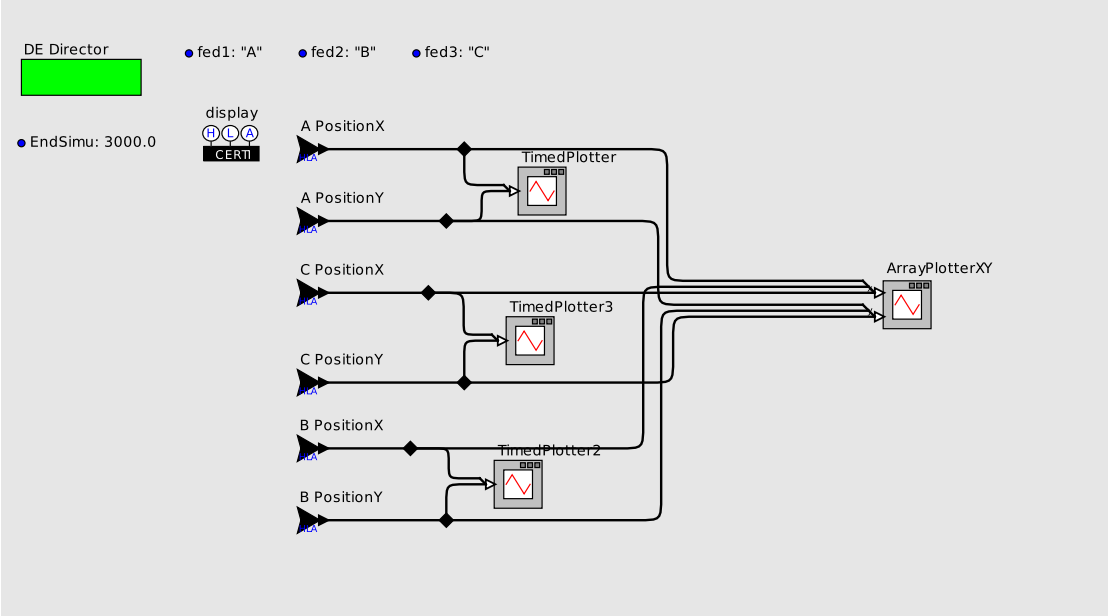


Figure 9: A non scalable flat approach for distinguishing objects in a HLA Federation. A, B and C are placeholder name for future objects not yet discovered.

to manually tune the model in order to make it fit the targeted simulation for handling the right number of objects. This phase is dull, error-prone and can be automated. We want the end user to only design a minimal model that will automatically scale up.

We describe in the following section the solution we used, its rationale, associated abstract semantic and its implementation.

5.1. Issue with the framework as it was

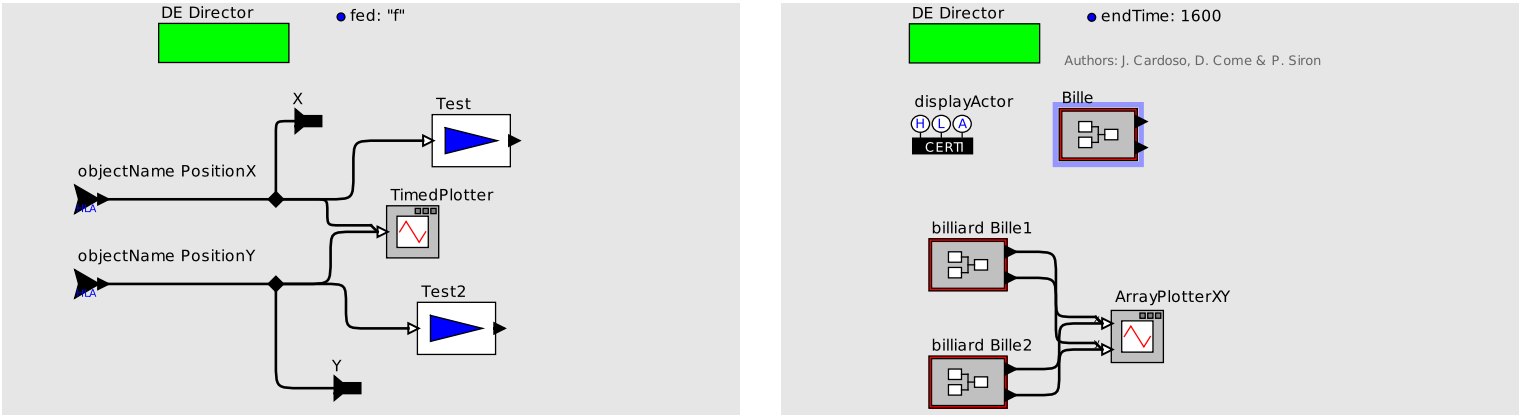
A previous work in [14] made possible for a HLA/Ptolemy model to distinguish several objects from the federation in the model. But it was based on a flat-approach : all *HLASubscribers* have to be in the top level of the model and have to be individually setup. The setting part is here to make sure all the messages from a given HLA object go to a predefined set of actors and are not interleaved across several actors with no way to tell where does a message come from.

But this does not scale well. If you had designed a model for dealing with at most three objects, there was no way you could use it for dealing with four objects. Figure 9 shows a such model. If you were to want to extend your model for dealing with an extra object, one had to add new *HLASubscriber* actors, configure them and plug them into the model. This was tedious and error prone. Moreover, using this extended model in federation with fewer objects to deal with would result in several actors being not used, making it harder to read and debug.

5.2. Solution

To overcome this difficulty, the main idea was to

- Use Ptolemy classes and objects to simplify the design phase.
- Be able to dynamically create a new instance of a Ptolemy class in the model if needed upon an HLA object discovery.



(a) Content of a class

(b) Overall model

Figure 10: Using Ptolemy classes for subscribing to classes and attributes

5.2.1. Mandatory classes

Within Ptolemy, one can design a *class*. That class that can be instantiated (in the same way a class is instantiated in object oriented programming) for creating new actors[33].

To simplify the process of handling new objects from the federation, we made the design choice to use classes in a mandatory manner for receiving data from the federation.

Rule 1 For each class you want to subscribe to, there must be a class in the model. Its name must match the name given in the Federation Object Model (FOM) file.

Rule 2 The framework will list all the *HlaSubcriber* inside a given Ptolemy class and subscribe to the parameter they are set up with.

With that approach, we can create a new actor by simply creating a new instance of that Ptolemy class and the setup phase of the *HLASubscribers* can be automatically done at the actor level by the framework. Moreover, all computation that is object specific is done within that class while all computation that operates on all actors is done in the overall model. Figure 10 shows such a model.

5.2.2. Scalable and reusable models

If a designer wants his model to be reusable from one simulation to another, in a drop and use manner (ie without a single edition to it), he can not do any assumption on the number of object the model will have to handle. With usual programming languages, this is done by using some dynamic memory allocation creating a new object.

There, the driving idea is similar : to dynamically instantiate a new actor when an object is discovered. This would allow Ptolemy models to be more scalable. The only design that is needed is for a single object and then the model automatically scales for handling the exact number of objects.

Existing work Model transformation has been around for a while in the *model driven* community. For instance, we can cite [23], a domain specific language for specifying model-to-model transformations, or [8]. Run time changes are available in some actor orientated framework, see the Fractal [12] with its *LifeCycleController* which allow explicit control over the main behavioral phases, for supporting dynamic reconfiguration.

For a specific Ptolemy approach, [17] uses an offline transformation based on pattern-matching analysis for creating scalable model from basic models. Its analysis is applied to a map-reduce problem. The *MultiCompositeActor*, as described in [33], is merely an automatic cloning device. This actor will clone its content (an

other actor) a given number of times during the *pre-intialize* phase. But in both cases, the transformation is done before the model is running. This works focus on model transformation at run time.

Mechanism's semantics This mechanism only works with DE as the level domain.

Change Request Let be $H = (A_1, \ldots, A_n)$ a set of actors. We introduce a *change request* mechanism which can either add a new actor to the list of actors or remove an actor from it.

Adding an actor A to the set H is done by using

$$CR_A^+ : H \mapsto H \cup \{A\} \tag{39}$$

Removing the actor A_j from the set H is done by the following function

$$CR_{A_j}^- : H \mapsto \{A_i \in H, i \neq j\} \tag{40}$$

Introducing a change in a model Let's $A = (I, O, S, s_0, F, P, D, T)$ be an actor. We defined the *augmented actor* $A\prime = (I, O, S', s_0', F', P', D', T')$.

A describes the actor's core function while A' extends A to take into account the ability to introduce a new actor within a block diagram.

The changes inside a model are triggered by the current actors within model. Introducing a new actor will be done by the director, but the request itself is queued inside the actor in the CR set. We will not describe *why* an actor would like to change the model, just it wants to. Thus, we will not describe the explicit creation of the *change requests*, only it happens in the transition function.

$$S' = S \cup \{CR\} \tag{41}$$

$$s_0' = s_0 \cup \{CR \mapsto \varnothing\} \tag{42}$$

$$F'(s, x) = \begin{cases} F(s, x) & \text{if } \mathbf{CR} = \varnothing \\ \forall o \in O, o = \bot & \text{otherwise} \end{cases} \tag{43}$$

$$P'(s, x) = \begin{cases} P(s, x) & \text{if } \mathbf{CR} = \varnothing \\ s & \text{otherwise} \end{cases} \tag{44}$$

$$D'(s, x) = \begin{cases} D(s, x) & \text{if } \mathbf{CR} = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{45}$$

$$T'(s, x, d) = \begin{cases} T(s, x, d) & \text{if } \mathbf{CR} = \varnothing \\ (s, \mathbf{CR} \mapsto \varnothing) & \text{otherwise} \end{cases} \tag{46}$$

The main idea beyond the augmented semantics is that actor which wish to change the model's topology produces no output at the current time, ask to stay at that time with a zero deadline and do not change its state in the postfire method.

Composition operator We introduce a composition operator. For a given set of functions $A = \{f_1, \cdots, f_n\}$ which all commute, we define $\odot$ operator by

$$\bigodot_{f \in A} f(H) = \begin{cases} f_1 \circ \cdots \circ f_n(H) & \text{if } H \neq \varnothing \\ Identity & \text{otherwise} \end{cases} \tag{47}$$

Director's semantics This works only with DE as the top level director.

Let an initial set of actors $H_0 = (A_1, \ldots, A_n)$ (with $A_i = (I_i, O_i, S_i, s_{0,i}, F_i, P_i, D_i, T_i)$) describing the initial block diagram.

The discrete event director is described as the following

$$DE = (I, O, S, s_0, F, P', D, T) \tag{48}$$

.

with I, O defined as in 2.1.2 and $s_0 = (s_{0,1}, \ldots, s_{0,n}, H \mapsto H_0, CR \mapsto \varnothing)$

In S, H represents the current extended actors that are in the model and CR represents the queued change requests at the current time.

The extended version of the director only changes its *postfire* function.

$$P'(s, x) = \{P(s, x) | H \mapsto \bigodot_{f \in \bigcup_{i=1}^{n} A_i.CR} f(H)\} \tag{49}$$

The idea is to apply to the set of actors all the change requests that currently stored by the different extended actors.

Match between semantics and implementation Ptolemy's code semantics differs slightly from the one described above because it has a *prefire* function. That function returns true if the actor is ready to be fired.

This can be implemented in the semantics by adding a *prefire* variable in an actor's state, and if that variable is true, make sure that *fire* does not produce any outputs and *P* does not change the state.

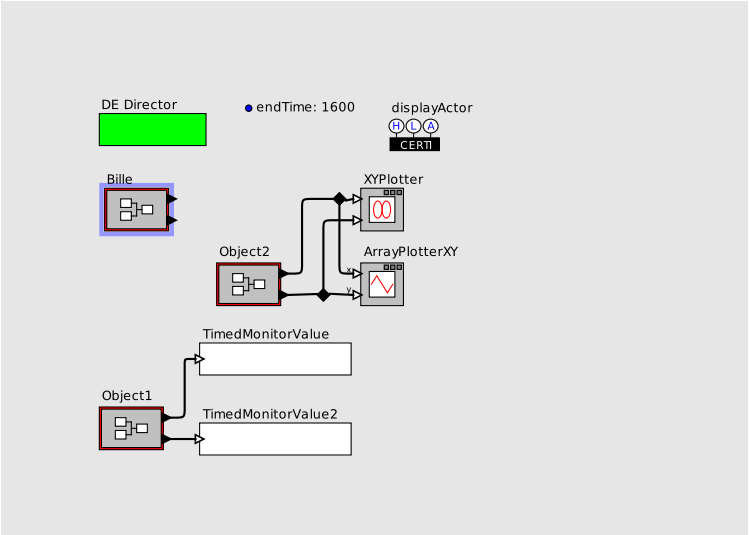
Mechanism's implementation Ptolemy has a mechanism called *ChangeRequest* that enables modification of the model while it is running. Given that two threads are running for each model (one for the GUI, one for the computation), the change requests are not instantaneously executed but rather queued and executed at the beginning of a new iteration after acquiring a write lock on the model.

Objects are discovered when the *DiscoveredObjectInstance* callback is called. That callback is called after the application yields some CPU time to the RTI through the *tick* method. This is done in *proposeTime*, which is called in *director.prefire* and if there are no actors to fire.

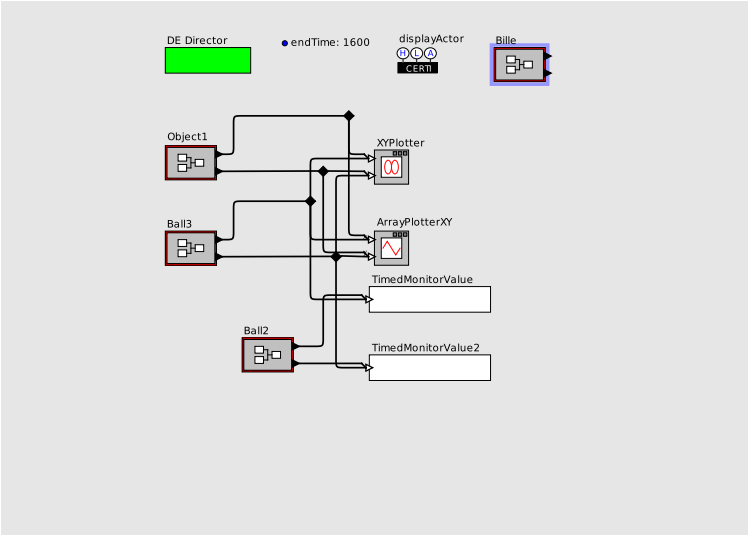
The *proposeTime* method's purpose is to give to back to a director the time the RTI has granted to the federate. It is strictly analogous to the *Deadline* function described in 4.1.4, even if the HlaManager is *not* an actor.

Below is the pseudo code for an iteration

```
1  Manager.iterate
2      execute Change Requests
3      pre-initialize new actors
4      initialize new actors
5      pre = container.prefire
6          director.prefire
7              proposeTime
8                  rti.tick
```



(a) Before the execution



(b) After the execution and auto layout

Figure 11: Visual explanation for outport policy

```
9       if(pre = true)
10          container.fire
11              director.fire
12              for each actor A to fire
13                  if(A.prefire)
14                      A.fire
15                      A.postfire
16          container.postfire
17              director.postfire
```

Issue The issue with that mechanism was that even if the RTI ensures that DOI $\prec$ *UAV*, Ptolemy can't process the *UAV* messages until the object are created. But, before doing so, the fire method would have been called, advancing the model's time to a greater time than the one in the UAV messages. On the next iteration, the objects are created and the messages processed but an exception is triggered because the model tries to post messages in the past.

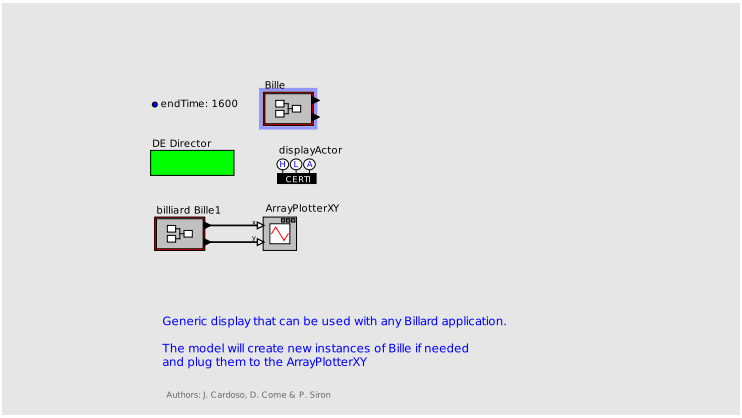
The solution was to make *prefire* returning false if an object has been discovered. That way, the current iteration stops and neither *fire* nor *postfire* are executed. Then the change requests are executed when the new iteration starts the queued messages can be processed.

How to know what to do with new objects A key question that had to be tackled was to find a way to know what to do with newly created objects, especially how should we connect the ports.

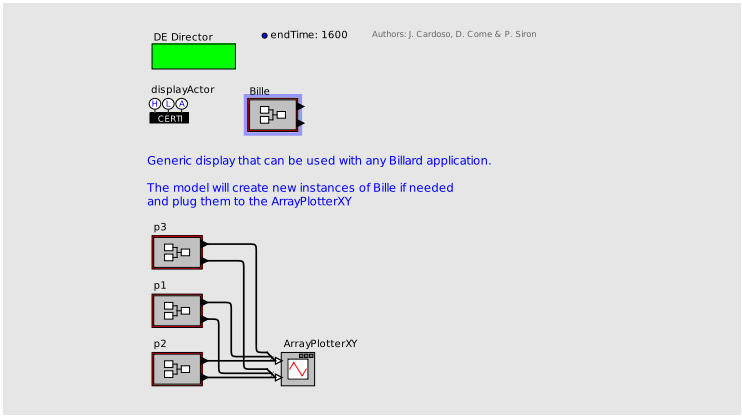
We choose to rely on instances that exist before the model is run, they are called *free instances*. These instances are used as models for the potential new instances. Formally, let's be C a class, p one of its output ports and $C_{1\leq i\leq n}$ be n free instances of C (whose port has been renamed p_i). Let's be $R(a,b)$ a function from $O \times I \rightarrow \{0,1\}$. For any outport a and any input port b, R yields true if a is connected to b.

For any dynamic instance, its port p will be connected to the ports in $\bigcup_{i=1}^{n} \{x | R(p_i, x) = 1\}$.

The figures on 11 visually explains what is described above. When a third object is discovered, there are no free instance. A new actor is created and mapped to that object its output ports are connected to all input ports which are connected to any free instance output port.



(a) Model before execution



(b) Model after execution and autolayout

Figure 12: Generic scalable model

Remarks on design We keep the same symbols as in the paragraph above.

Design and repeatability For each port p it is strongly advised that

$$\forall i, j, 1 \leq i, j \leq n, \forall x, R(p_i, x) = R(p_j, x) \tag{50}$$

.

If this is not true, it means at least two free instances have different receivers for that port p. Since the mapping between free instances and the HLA objects is done according to the discovery order of the former, different simulations runs may have different results if the objects are not discovered in the same order.

Practical matters The end user needs to ensure that :

$$\forall p, \forall x \; R(p_j, x) = 1 \Rightarrow x \text{ is a multiport} \tag{51}$$

Indeed, if we were to create a new object, then its port will be connected to x. If x is not a multiport then, it cannot handle more than one input and will throw an error at run time if we try to do so.

5.2.3. Results

The simulation was tested against a distributed simulation of a pool table which consist of several ball bouncing on table. A ptolemy model will be used a generic display for that simulation. Here, each Ball was updated a single C++ federate[8]. We could have used three others Ptolemy models or any other setup.

The FOM used is listed below on listing 1

6. Others functionalities

6.1. Timeline adaptability

A given federate works on a given time scale (seconds, , milliseconds..), called K. This timeline is given the computation the federate does. For simulating a system for a duration T, the simulation's end of time must be equal to $\frac{T}{K}$. But a federate works within a federation and that federation may have a different time scale. That value is defined inside the FOM by the *TIM_Units* parameter. Thus, to comply with this paramater the federate

```
(Fed
  (Federation Test)
  (FedVersion v1.3)
  (Federate "fed" "Public")
  (Spaces
        (Space "Geo"
                (Dimension X)
                (Dimension Y)
        )
  )
  (Objects
    (Class ObjectRoot
      (Attribute privilegeToDelete reliable timestamp)
      (Class RTIprivate)
        (Class Bille
                (Attribute PositionX RELIABLE TIMESTAMP)
                (Attribute PositionY RELIABLE TIMESTAMP)
                (Class Boule
                    (Attribute Color RELIABLE TIMESTAMP)
                )
        )
    )
  )
  (Interactions
    (Class InteractionRoot BEST_EFFORT RECEIVE
      (Class RTIprivate BEST_EFFORT RECEIVE)
    )
  )
)
```

Listing 1: Federation Object Model used for the tests

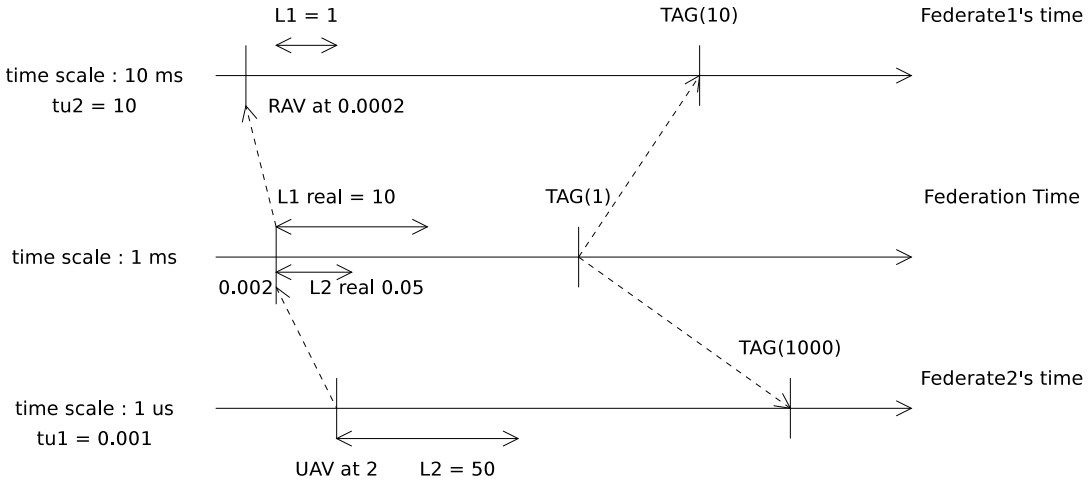


Figure 13: Time line adaptability illustrated

Former action	Real action
Lookahead value L	Effective lookahead is $L \cdot t_u$
Incoming message at time T	local token is timestamped at $\frac{T}{t_u}$
Outgoing message timestamped at T	token is send with time $(T + L) \cdot t_u$
RTI grants time T	model advances to $\frac{T}{t_u}$
Model wants to move to time T	model really asks RTI for to time $T \cdot t_u$

Table 1: Mapping between fundamental actions and real ones

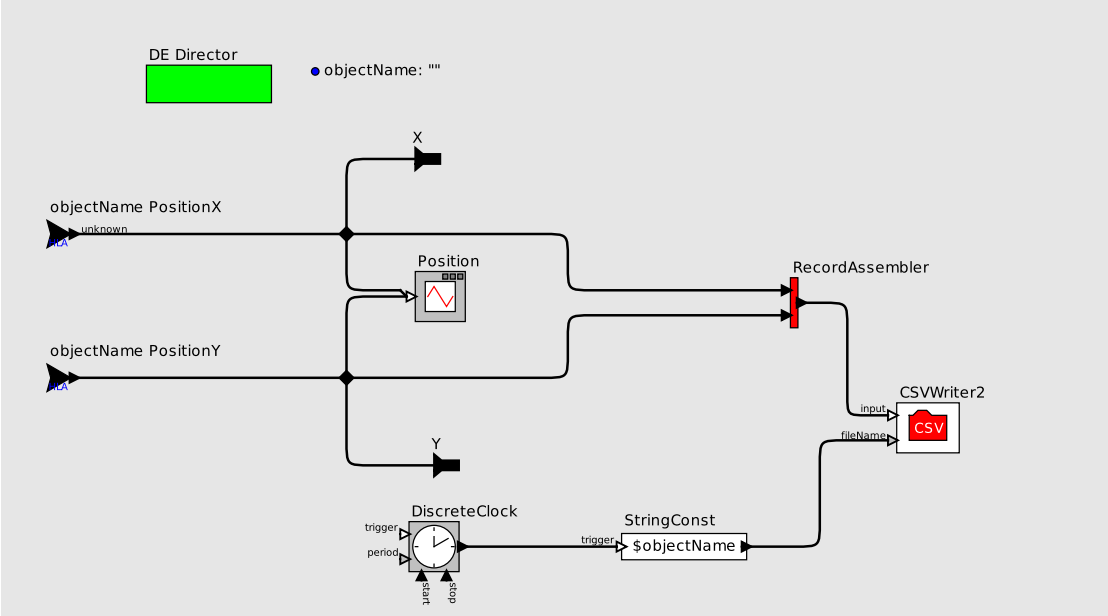


Figure 14: Using the object's name in a model

has to shift his events by a certain amount. This is the purpose of a newly introduced *time unit* parameter into the *HLAManager*.

The transformations induced by time unit parameter t_u are described on table 1.

6.2. Using an object's name

An object within a federation always has name : either is was provided on its registration or by the RTI when another federate asks for it[9]. This value was used to set the actor's *displayed name* and in the actor's *HLASubscribers* but the user could not use it.

Now, this value can be retrieved inside an actor if the former defines a *objectName* parameter. When designing the model, in any context where that parameter is used, Ptolemy will try to evaluate it. Thus the parameter must not be null.[10]. The figure 14 shows a model where the object name is used to write the received values in a CSV file whose name is the object name.

[8]CERTI's binary billard or billard-nogui

[9]when another federate uses *GetObjectInstanceName*

[10]empty string is a good default value

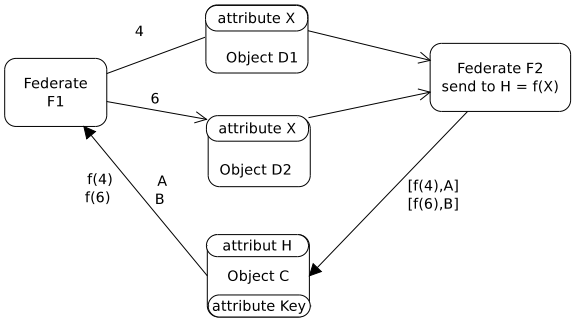


Figure 15: Issue when using a single object

6.3. Realtime synchronization scaling

An user may want to have the model's time synchronized with the wall clock's time. This is useful when doing a real time simulation or a simulation with some hardware in the loop. Ptolemy had a the *SynchronizeToRealTime* attribute, which implements the *TimeRegulator* interface and returns the proposed time once the real time has catch up with the model time. This attribute was created to have a one to one mapping between real time and model's time.

There are cases where this is not desirable, one might want to have 2 seconds of model's time for each 1 second of wall clock's time, or the other way around, have spending 2 seconds in wall clock's time for one second of model time. The *scale* attribute was introduced in *SynchronizeToRealTime* for that purpose. With a scale factor of 0.5 will make the model time passes twice as fast as real time, while make it equals 2 means that 2 seconds in wallclock are needed for a single unit of time in the model.

6.4. Sending some information to a particular object

6.4.1. Situation

Let's imagine we have inside a federation two federates handling a set of drones. The first one controls the drones whereas the other one compute the drones height and send them back this information. A drone is only interested in its own height. We need to be able to send it the right height and that turns out be harder than expected.

6.4.2. Using publish subscribe pattern

The primary data exchange mechanism in HLA is the publish/subscribe pattern. But that requires having an object in the HLA world. That means having at least one object for sending the height back to the first federate.

One global object Using a single object across the whole federation is not possible for sending back the heights to the first federate with the current framework.

A single object implies to send with the value some kind of key to identify the right drone. But the HLA standard explicitly states that one update attribute values invocation could result in multiple reflect attribute values invocations in a subscribing federate. This implies that two UAV (with a key and the height) could result in 4 RAV messages all with the time stamp in the first federate and with no way to know how to match the different parts. The figure 15 illustrates that situation. Thus, the federate F1 can receive up to four RAV messages for object C (two for the attribute H, two other for the key) with no way no know which value goes with witch key.

There is an user-supplied tag on many service provided by HLA. The standards explicitly state that *these arguments shall be provided as a mechanism for conveying information between federates that could be used*

to implement priority or other schemes. Thus, we would need to encode into the tag the intended receiver and be able to process that information on the receiver side to route the token inside the model.

One object per drone To overcome the issues described above with today's framework, we can use one new object per drone. We still need a key to identify the channel, but federate F1 cannot mix the incoming messages. We have to distribute all the pair $(value, key)$ to all the actors, which will discard pairs with the wrong key.

Such models are shown on figures 16 and 17. The major drawback of this is that with the current framework implementation, we can only register an object if there is a corresponding actor *before* executing the model. That means all models have to be statically configured. Even if this limitation was lifted, there is still a problem : we need a least an actor we can use as a model to know what to do with potential new objects. But such an actor will trigger the registration of an object if connected to an *HlaPublisher*, even if no "source "object is registered. To overcome this, we need to review the way we specify how new objects are inserted into the model.

6.4.3. Using shared ownership

After a federate has stated what it will publish, it has ownership of the corresponding attributes for any object it registers. Any other attribute is *not owned* and can be owned by another federate. If a drone had 3 attributes (X,Y and H), the first federate could publish X,Y and subscribe to H and the other way around for the second federate, as explained on figure 18.

This approach could be used but the code base needs some changes.

7. Bugs and soundness

7.1. Distributed simulation's intrinsic difficulty

In any project, tracking down a bug can be difficult. In this project, any bug could be :

- Within the framework's code
- Within CERTI's code
- Or even within Ptolemy's code

And with N federates, there are N RTIAs and one RTIG. That means each debug session, we had to cover at least three pieces of software, usually four or five. With hindsight, we can say that is one aspect that makes distributed simulation hard.

7.2. CERTI's bugs

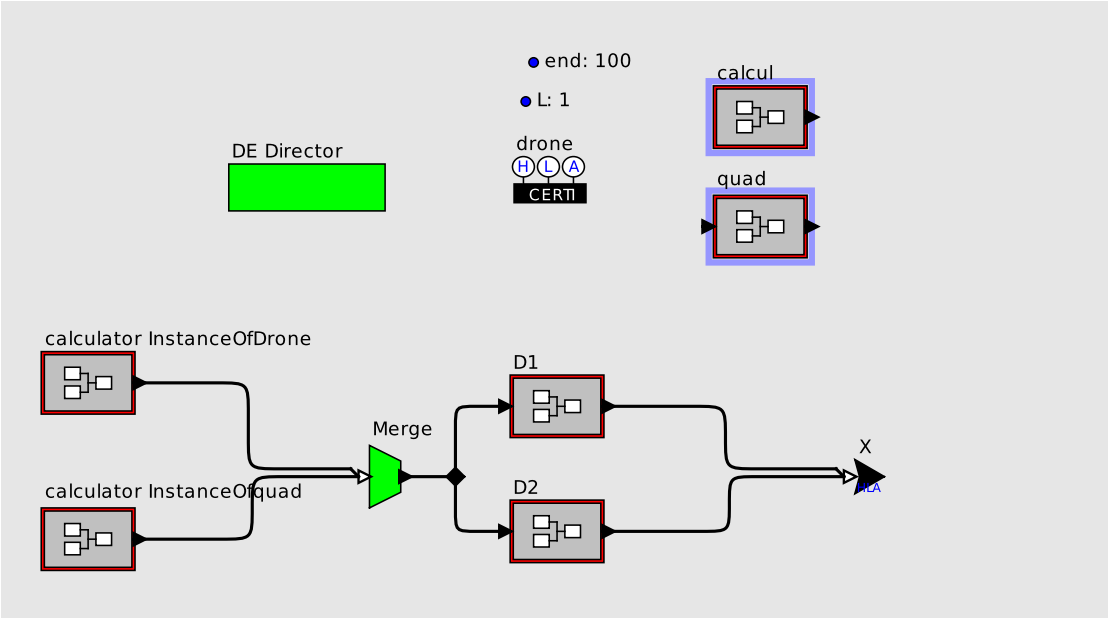
Like any software, CERTI is not bug free. The project has uncovered several bugs in CERTI.

The main one (and which is so far unresolved) is described below.

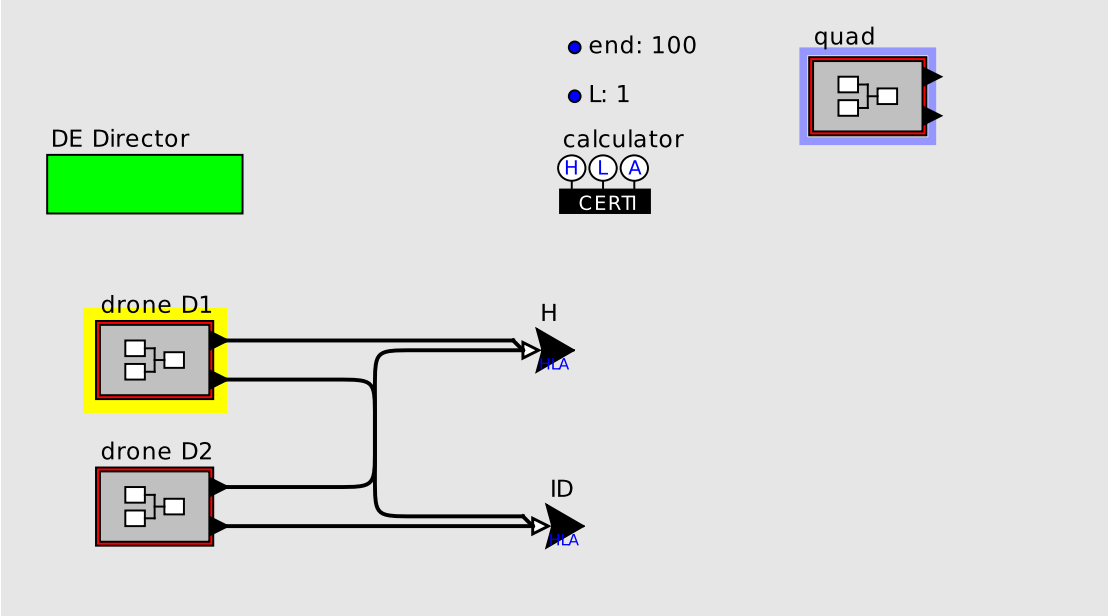
A problem arose when using the simulation described on figure 19 : 2 federates, each updates an object and subscribes to the other, so that there is a feedback loop in the simulation. A single message from one of the federates should spawn many messages thanks to the loop until the end of simulation's time is reached. In the simulation, the federate F1 was the one sending that first message.

With a lookahead of 1 for each federate and an initial message send at 0, the output should have been like the one on table 2. But the simulation stopped after F1 receives its first message. A closer look to the logs shows that F1 sends the UAV message at time 3 but that message was never received by F2.

To pinpoint the exact the issue we reproduce the the issue with sheer C++ code and it was successfully reproduced.



(a) Model which will register the objects and receive the feedback (F1)



(b) Model F2 : it computes f(X) for each object

Figure 16: Feedback with one channel per object

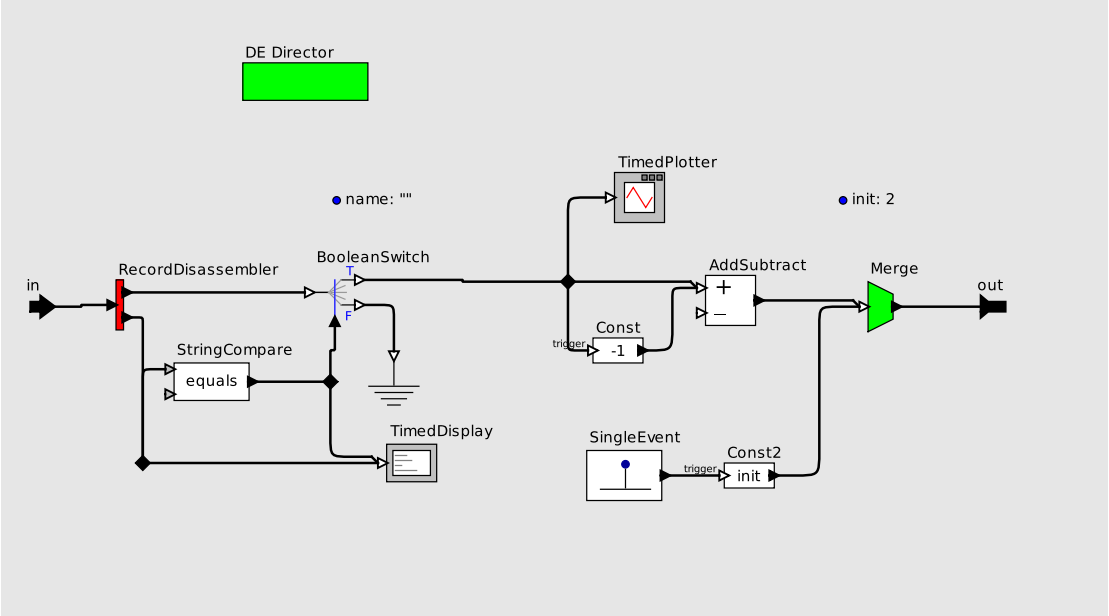


Figure 17: Content of class quad shown on figure 16a

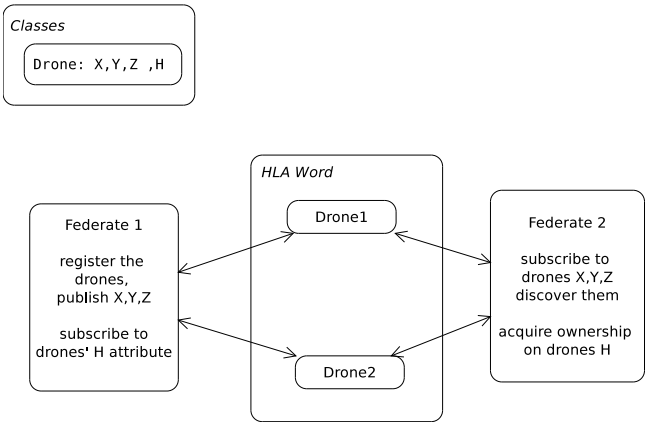


Figure 18: Shared ownership accros federates

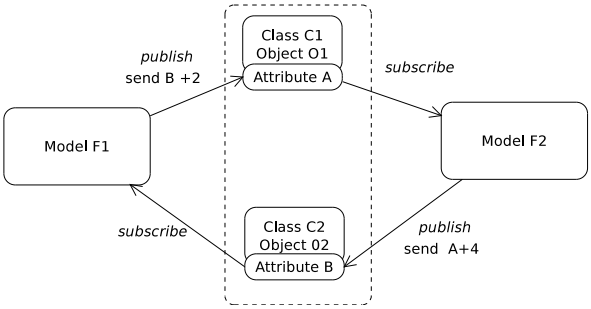


Figure 19: Simulation used for discovering CERTI's bug

Model time	F1'output	F2's output
1	UAV to O1.A : 42	RAV from O1.A : 42
2	UAV to O2.B : 46	RAV from O2.B : 46
3	UAV to O1.A : 48	RAV from O1.A : 48
4	UAV to O2.B : 52	RAV from O2.B : 52
5	UAV to O1.A : 54	RAV from O1.A : 54

Table 2: Expected output

The problem disappeared when the compile-time option *NULL_PRIME_MESSAGES* was not enabled in CERTI. This option enables *NULL Message Prime algorithm* described in [13]. The algorithm's purpose is to fight well known time creep issue which happens when the lookahead is not well chosen with a RTI that uses the classical algorithm from Chandy and Misra for deadlock detection. So far, we have not been able to tell if there is an issue with the algorithm itself, or with its implementation in CERTI.

7.3. Simultaneous events

The current framework had a open question on simulteanous events for a given *(object,attribute)* pair. At a given time, is it possible to receive several updates (RAV messages) for single *(object,attribute)* pair ?

The standard does not say anything about that topic, making possible for a federate to send several updates at the same time for a single attribute as long as the others preconditions are fulfilled. Thus we had to adapt the code to handle that case, even if none of our simulations is using it.

Future work and conclusion

Both HLA and Ptolemy are interesting technologies and the HLA-CERTI cosimulation framework tries to bring the best of it to people. Through this internship, the framework has been improved on theoretical and practical aspects. But there are still several aspects that can be improved such as :

- Designing a better mechanism for inserting new objects
- Adding missing features
- Exploring the attributes' ownership sharing and its consequences
- Generalizing the semantics used for scalability for decorating actors with dynamic behaviors

On a more personal point of view, this internship was a marvelous experience and I have learned many things on a personnal and professional point of view. This internship also concludes my scholarship at the ISAE/Supaero and the end of four unforgettable years.

Thanks

First of all, I would like to thank J. Cardoso, without her this experience would not have been possible. Then E. Lee for hosting me in his group and P. Siron for his tutoring work. Also C. Brooks, M. Lohstroh and C. Shaver for their advices on Ptolemy. And E. Noulard for his inputs on CERTI.

I also would like to thank the *Fondation ISAE-Supaero*, *région Midi Pyrénées* and my parents for their financial support. Without them, none of this would have happened.

References

[1] URL: http://chess.eecs.berkeley.edu/ptexternal/ (page 6).

[2] URL: http://www.nongnu.org/certi/.

[3] URL: http://www.nongnu.org/certi/certi_doc/Install/html/build.html (page 14).

[4] URL: https://en.wikipedia.org/wiki/Run-time_infrastructure_(simulation) (page 14).

[5] URL: http://www.forwardsim.com/products/hla-blockset/ (page 15).

[6] URL: http://jade.tilab.com/ (page 15).

[7] URL: https://www.fmi-standard.org/ (page 14).

[8] A. Agrawal et al. "The Design of a simple language for graph transformations". In: *Journal in Software and System Modeling* (2005) (page 21).

[9] Muhammad Usman Awais et al. "The High Level Architecture RTI as a master to the Functional Mock-up Interface components". In: *International Conference on Computing, Networking and Communications, Workshops Cyber Physical System* (2013) (page 14).

[10] Alisson V. Brito et al. "Development and Evaluation of Distributed Simulation of Embedded Systems Using Ptolemy and HLA". In: *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*. DS-RT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 189–196. ISBN: 978-0-7695-5138-8. DOI: 10.1109/DS-RT.2013.28. URL: http://dx.doi.org/10.1109/DS-RT.2013.28 (page 15).

[11] C. Brooks et al. *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains*. Tech. rep. University of California, Berkeley, June 2004 (page 15).

[12] Eric Bruneton et al. "The Fractal Component Model and Its Support in Java". In: *Software : Practice and Experience* (2003) (page 21).

[13] Jean-Baptiste Chaudron, Eric Noulard, and Pierre Siron. "Design and modeling techniques for real-time RTI time management". In: *Simulation Interoperability Workshop* (2011) (page 32).

[14] David Come. "Improving Ptolemy-HLA co-simulation by allowing multiple instances". In: *ISAE-Supaero* (2015) (pages 18, 20).

[15] Daniel Lazaro Cuadrado. *Automated distributed simulation in Ptolemy II* (page 15).

[16] U.S. Department of defense. *High-Level Architecture Rules Version 1.3*. 1998 (page 11).

[17] Thomas Huining Feng. "Engineering Structurally Configurable Models with Model Transformation". MA thesis. Electrical Engineering and Computer Sciences University of California at Berkeley, 2008 (page 21).

[18] Richard M. Fujimoto. *Parallel and Sistributed Simulation Systems*. A Wiley-Interscience publication, 2000 (page 13).

[19] Richard M. Fujimoto. "Time Management in the High Level Architecture". In: *Simulation* 71 (1998), pp. 388–400.

[20] Daniele Gianni et al. *HLA-Transparent Distributed Simulation of Agent-based Systems*. University Campus STeP Ri Slavka Krautzeka 83/A 51000 Rijeka, Croatia: InTech, 2010 (page 15).

[21] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules". In: *IEEE Std. 1516-2000* (2000), pp. i–22. DOI: 10.1109/IEEESTD.2000.92296 (page 11).

[22] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules". In: *IEEE Std. 1516-2010* (2010), pp. i–38. DOI: 10.1109/IEEESTD.2010.5553440 (page 11).

[23] Frédéric Jouault et al. "ATL: A model transformation tool". In: *Science of Computer Programming* (2008) (page 21).

[24] Gilles LASNIER. *Toward a Distributed and Deterministic Framework to Design Cyber-Physical Systems*. ISAE-Supaero, 2013.

[25] Gilles Lasnier et al. "Distributed Simulation of Heterogeneous and Real-time Systems". In: *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium* (2013), pp. 55–62. DOI: 10.1109/DS-RT.2013.14.

[26] Gilles Lasnier et al. "Environnement de coopération de simulation pour la conception de systèmes cyber-physiques". In: *JESA* 47 (1-3) (2013), pp. 13–27. ISSN: 1269-6935.

[27] E. A. Lee and Zheng. "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems". In: *EMSOFT '07: Proceedings of the 7th ACM and IEEE International Conference on Embedded Software* (2007) (page 7).

[28] Edaar A. Lee. "Cyber Physical Systems: Design Challenges". In: *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)* (2008) (page 5).

[29] Edward A. Lee. *Concurrent Models of Computation - An Actor-Oriented Approach*. http://Ptolemy.org/books, 2011 (page 9).

[30] Edward A. Lee and Alberto Sangiovanni-Vincentell. "The tagged signal model, a preliminary version of a denotational framework for comparing models of computation". In: *Department of Eelectrical Engineering and Computer Science, University of California* (1996) (page 7).

[31] X. Liu and E. A Lee. "CPO semantics of timed interactive actor networks". In: *Theoretical Computer Science* 409 (2008), pp. 110–125 (pages 7, 9).

[32] Xiaojun Liu. "Semantic Foundation of the Tagged Signal Model". PhD thesis. Department of Eelectrical Engineering and Computer Science, University of California, 2005 (page 7).

[33] Claudius Ptolemaeus. *System design, Modeling and simulation using Ptolemy II*. UCB, 2014 (page 21).

[34] Ragunathan Rajkumar et al. "Cyber-Physical Systems: The Next Computing Revolution". In: *Design Automation Conference* (2010) (page 6).

[35] Stavros Tripakis et al. "A modular formal semantics for Ptolemy". In: *Mathematical Structures in Computer Science* (2013) (pages 7, 10).

[36] Wei Xiong, Pengshan Fan, and Hengyuan Zhang. "HLA Based Collaborative Simulation with MATLAB Seamlessly Embedded". In: *International Journal of Machine Learning and Computing* 2 (2012) (page 15).

[37] Li Yanxuan. "A Distributed Simulation Environment for Cyber-Physical Systems". In: *ISAE-Supaero* (2015) (page 18).

A. HLA Quick and dirty manual

Quick and dirty Ptolemy-HLA manual up to date on September 2015.

Receive value Lets assume you want to subscribe to a class named *C* which has 2 attributes *A* and *B*.

Class design

- Create a *CompositeActorDefinition* named C. The name has to match the one in the fed file !
- Put inside the class two *HlaSubscriber*, one per attribute you want to subscribe to. Double-click on one to set up its type and the parameter's name, in our case *A* in one case, *B* in the other. If the out port's type is wrong for any reason wrong, set it up directly.
- If you dont want to subscribe to an attribute, don't put a *HlaSubscriber* for it in the class
- The class has to have a director (and thus be opaque) if you want to use dynamic instances because otherwise the actors with the model wont be initialized.
- If your class is opaque, make sure it has a stop time.
- You may retrieve the object's name if the class has a parameter called *objectName*. You can use it with the following syntax *$objectName*.
- Design the rest of the class as you like.

Using that class

- You may create instances of that *C* class. Instances that exist before running the model are *free instances*.
- Upon an object discovery (*discoverObjectInstance*), the newly discovered object will be assigned to a free instance of the *C* class.
- If there are no free instances when discovering an object , then the framework will create a new instance and maps the object to that instance, called a *dynamic instance*

Behavior specification

- For each output port, the framework will compute all the destination ports for all instances and the output port of any dynamic instance will be linked to all ports which have been listed previously.

See 5.2.2

Sending values Let's assume there is a class named *C* which has 2 attributes *A* and *B* and you want to create 3 objects of that class in the federation.

- Create 2 HlaPublisher. Renamed (F2) one in *A* and the other on *B*
- Double-click on each of them to set object class name (*C* here.
- create 3 **opaque** actors and link their output ports to the right HLASubscriber. You can create a generic class and instantiate 3 actors, copy and paste an existing actor, does not matter.

The executive director must be a DE Director.

B. Getting log with JCerti

If you want to log with JCerti, you need to add into Ptolemy's classpath[11] a *certi.properties* file.
Useful variables are :

- logLevel whose values are in java.util.logging.Level. Don't put it or use OFF to disable the logging.
- enableHtmlLogging , true or false
- htmlLogFileName, if you want to customize the file's name for the HTML log
- enableTextLogging, true or false
- textLogFileName, if you want to customize the file log ¿

[11] When running Ptolemy, you can find the classpath with jconsole. On my system, ~/.ptolemyII belongs to the class path