Toward a Formalism to Study the Scheduling of Cyber-Physical Systems Simulations

Henrick Deschamps and Gerlando Cappello
Modelling and Simulation department
Airbus Operation SAS, Toulouse, France
Email: {firstname.name}@airbus.fr

Janette Cardoso and Pierre Siron
Complex systems engineering department
ISAE-SUPAERO, University of Toulouse, France
Email: {firstname.name}@isae.fr

***Abstract*—This paper presents ongoing work on the formalism of Cyber-Physical Systems (CPS) simulations. These systems are distributed real-time systems, and their simulations might be distributed or not.**

In this paper, we propose a model to describe the modular components forming a simulation of a CPS. The main goal is to introduce a model of generic simulation distributed architecture, on which we are able to execute a logical architecture of simulation. This architecture of simulation allows the expression of structural and behavioral constraints on the simulation, abstracting its execution.

We will propose two implementations of the execution architecture based on generic architectures of distributed simulation:

- **The High Level Architecture (HLA), an IEEE standard for distributed simulation, and one of its open-source implementation of RunTime Infrastructure (RTI): CERTI.**
- **The Distributed Simulation Scheduler (DSS), an Airbus framework scheduling predefined models.**

Finally, we present the initial results obtained applying our formalism to the open-source case study from the ROSACE case study.

***Index Terms*—Aeronautics, CPS, Modelling, Simulation, Scheduling, HLA, DSS, CERTI.**

I. Introduction

Aircraft are systems including control loops to stabilize a vehicle in a physical environment, and are good examples of Cyber-Physical Systems (CPSs) [1].

In a CPS, computational resources are tightly interacting with physical elements through conversion subsystems, such as sensors and actuators.

The increasing CPS complexity has an impact on the complexity of their simulations. Aircraft simulations might integrate hundred models, and hundreds of thousands of communication channels between the models. In order to receive the full benefit of a test executed in a simulated environment, one must prove that the simulated system is sufficiently valid, regarding the given needs.

Moreover, some simulation usages impose a reproducibility guarantee on the tests done on simulated aircraft. However, it must be pointed that the real aircraft has non-reproducible behaviors. Thus, fidelity and reproducibility may appear contradictory, but they are not necessarily incompatible, depending on the type of tests that need to be done, and the assumptions made.

In addition, some uses of simulation for training purpose and validation, respectively involving human and machine in the loop, add to the simulation soft and hard real-time constraints.

"Scheduling" is a solution to a scheduling problem [2]. A scheduling describes, through a given formalism, the execution of tasks, and the resources allocation over time, in order to be compliant with objectives, while respecting constraints. Generally, in computer architecture, tasks are threads or processes and the goals of scheduling is to minimize latency, maximize throughput or minimize response time.

In this paper, we propose a definition for simulation scheduling in the context, as well as the method we use to formalize this scheduling. The method formalisms are developed in § IV and V, and their usage in § VI. Finally, in § VII, we illustrate this method with a case study.

II. Context

We call the cyber and physical models in CPS simulation **components**. We call **scheduling of CPS simulation** the assembly of simulation components, namely the temporal organisation of component execution and the synchronisation of their interactions. Currently, a scheduling of simulation of CPS is obtained experimentally. Obtaining a scheduling does not allow to estimate the validity of this scheduling regarding the targeted tests a priori, and the iterations between simulation and real world are expensive.

In order to analyse the impact of the integration of modular components while reducing the costs, we need a formalism of the execution of a simulation of the CPS.

Our objective consists in defining a formal method to determine a priori the validity of CPS simulation scheduling, for a distributed simulation of a cyber-physical system. The goal is to prove by analytic methods that depending on the objectives of a given test on a given platform, an assembly of models will be sufficiently representative.

The components and their interactions form the **simulation logical architecture** (sLA). The sLA allows the expression of logical structures with requirements for the targeted simulation, but not the simulation execution. The **simulation execution architecture** (sEA) is a generic execution platform allowing the execution of any sLA. Thus, the sEA allows the verification of execution properties, independently of the

SLA. The binding of the SLA with an instance of SEA is the simulation.

III. Related work

The Discrete Event System Specification (DEVS) formalism by Bernard P. Ziegler [3] is a modular and hierarchical formalism for modeling and analyzing general systems that can be discrete event systems or continuous state systems. Discrete event systems might be described by state transition tables, and continuous state systems which can be described by differential equations. *A priori*, DEVS seems adapted for CPS simulation modelling. However, our experiments with analysing CPS simulation scheduling with DEVS and its parallel extension shows us that this formalism is not adapted to this situation. For example, in order to analyse the delay on a data path, using the period of models execution is more simple than the time advancement function in parallel-DEVS. This point is detailed in § IV.

Our goal is to model a CPS simulation in order to analyse its scheduling. The simulated CPS is tightly linked to the execution platform. In order to capitalize on the scheduling analysis, this analysis must be the most independent possible from the implementation of the execution platform. Trying to define a scheduling of tasks without a scheduler is not possible, thus an abstract version of the execution platform must be defined. Other formalisms than DEVS are most suitable to describe the execution platform, for instance Architecture Description Languages (ADLs).

One of the ADLs is the Architecture Analysis & Design Language (AADL), coming from the avionics domain. An AADL model comprises software and hardware components The software components regroup the data, thread, process and subprogram, while the execution platform components regroup the memory, bus, processor and devices. Recent works proved the worthiness of AADL for prototyping distributed systems [4] through an extension: REAL (Requirement Enforcement Analysis Language), and there are some efforts from the industry to use AADL in order to describe an execution platform of simulation [5].

In this paper, we propose a simple ADL has a support to describe the SEA. In further works, we might try to bring our formalism closer to the AADL.

IV. Atomic model of a simulation component

CPS simulations are composed of two kinds of components, the physical components, and the cyber components.

A. Characterizing cyber components

The cyber systems modelled in CPS simulations are, for instance, Flight Control Systems (FCS), Human Machine Interface (HMI) or network devices (routers, switches...). These are processes executed by computers.

A process is a computational entity, often referred to as a task. Today, most Operating Systems (OS) do not allow direct communication between processes, in particular, this is the case of avionics [6]. In these systems, the different processes are segregated in space and time.

The partitioning in time exists because of the scheduling of process, and the partitioning of space because two processes do not use the same memory without using OS mechanisms.

These notions of partitions are important for our components, they allow defining subsystem models which are totally isolated, apart from explicit communication.

In terms of formalism, this allows us to use a notion of bus for communication between our components, while the scheduling of tasks can be modelled by a periodical execution of models.

Finally, some models require a set of "previous" states to compute a new one. Considering our components as periodical processes with total time and space partition, we can use the common definition of discrete system modelling, such as the state advancement, and output: $x_{k+1} = F(x_k, u_k, t_k)$, $y_k = G(x_k, u_k)$.

With these characteristics, we can propose a formalism to express the cyber components:

$$cyber\_component =< I, O, S, S_0, \Delta_{in}, \Delta_{out}, f >$$

The set of inputs I are the data consumed by a component. For instance for an altitude controller, the first input could be the current altitude, while the second input could be the altitude reference. It should be noted that these two inputs are not necessarily consumed together, we can imagine a component using the reference less regularly than the current altitude. Thus, inputs might be consumed by a component in our formalism with different frequencies.

The set of outputs O are data produced by a component. For instance for an altitude controller, it could be commands for propulsion and elevator.

The states S and initial states S_0 of a component are different depending on the nature of the cyber components. For a controller, this is straightforward, but in general, its state is the vector of variables it manipulates, and the initial condition, their initializations.

The transition functions Δ_{in} are used to calculate new state based on previous ones, at the component frequency f. The output functions Δ_{out} use a set of recent states to compute the data to produce.

Certain processes do not use memory, the sets of states, initial states, and transition functions are empty, so their output functions only depend on the inputs.

B. Characterizing physical components

According to their characteristics, continuous-time models are described by ordinary differential equations (ODEs), partial differential equations (PDEs), differential algebraic equations (DAEs) or partial differential algebraic equations (PDAEs). In the following, we will focus on ODEs, because of their simplicity, but the methodology is the same for the other class of continuous-time models. ODEs are described by: $\dot{x} = f(x, u, t)$.

The ODE characterization for distributed simulation with real-time constraints is addressed in [7]. In this paper, we do not distribute the solving of ODE between multiple components, and we address the problem of an algebraic loop in the SEA.

Since our simulations of CPS are subjected to real-time constraints, we will only focus on numerical methods that can adapt to the limitation of the computing resources (in space and time).

We consider time discretization of continuous dynamic system. With a constant discretization interval of Δt, we have the following approximation: $x_{k+1} \approx x_k + \Delta t \times F(x_k, u_k, t_k)$, $y_k = G(x_k, u_k)$.

Those characteristics allow us to define the following formalism:

$$physical\_component =< I, O, S, S_0, \delta_{in}, \delta_{out}, f >$$

The elements of a physical component are the same as defined for the controller in the cyber components. The differences are that the physical components consume and produce all their data at their own specific frequency f, with $f = \frac{1}{\Delta t}$, and they only need one transition function δ_{in} and output function δ_{out}.

C. Extension to generic atomic model

We propose a generic atomic model of component, from the characteristics of cyber and physical components, with the perhaps naive propositions of formalisms, and the limitations of inputs/outputs. In order to simplify this generic atomic model of component, we do not address the problem of intercompatibility yet, and consider a syntactical level of intercompatibility, as described in [8]. The following formalism is largely inspired from DEVS, however, we introduced the component frequency, and adapt the DEVS concepts to our context. As, we want to introduce requirements depending on the simulated CPS, for instance the minimum and maximum latencies on data paths, the expression of periods simplify the estimation of time taken by a set of components in order to produce a data. Moreover, the definition of component in the early phases of simulation design can evolve quickly, and modifying the frequency of a component is faster than redefining the time advancement function. Furthermore, the coupling of component has to be flexible. At the component scale, input and output event are not considered, but ports. Two ports connected on a channel are producing and consuming data at frequencies defined by the connected component frequencies. Finally, components of simulation can interact differently, with different set of other components, at different rates. In contrary to DEVS considering one internal and one external function, we introduce sets of transition functions, and output function. We express the generic model of components c illustrated with some simplifications in fig. 1 as the following tuples:

$$c =< P_{in}, P_{out}, S_0, S, \Delta_{in}, \Delta_{out} > \quad (1)$$

where:

- P_{in} is the set of input ports, a port being data produced (or consumed) at a given frequency.
- P_{out} is the set of output ports.
- S_0 is a set of initial states.
- S is a set of states, depending on the definition of the component.
- Δ_{in} is the set of transition functions, with a transition function δ_{in} defined as how a set of data extracted from input ports changes the state of the component, at a given frequency:

$$\delta_{in} =< I, S_{in}, s_{out}, \delta, f > \quad (2)$$

  where:
  - I is the set of inputs of the function.
  - S_{in} is the set of current and previous states.
  - s_{out} is the computed state.
  - δ is the main function, taking n inputs and m previous states in order to compute a new state: $I^n \times S^m \rightarrow S$.
  - f is the frequency of the transition function.
- Δ_{out} is the set of output functions, with an output function δ_{out} defined as how a value for an output port p_{out} is calculated from the current states and a set of data from input ports:

$$\delta_{out} =< o, I, S, \delta > \quad (3)$$

  where:
  - δ_{out} is the output function.
  - o is the calculated output.
  - I is the set of inputs of the function.
  - S is the set of current and previous states.
  - δ is the main function, taking n inputs and m states to compute an output: $I^n \times S^m \rightarrow o$.

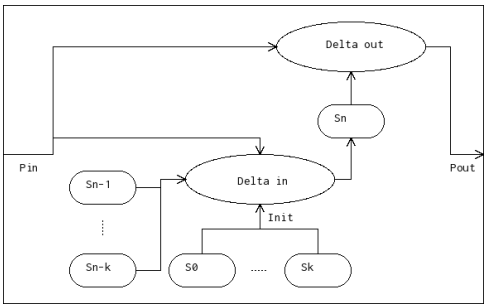

Figure 1: Simplified view of a component.

The component model defined is used in the SLA. In this section we introduce new elements in the SLA, the channel and the requirements. The channel allows connecting an output port of a component to an input port of another one.

We propose to model an SLA as the set of its components, the channels between its components, and requirements, as the following:

$$\text{SLA} =< C, \Lambda, R > \quad (4)$$

Where:

C is the set of simulation components c.
R is the set of requirements.
Λ is the set of channels λ used by components to exchange data, defined as:

$$\lambda =< p_{in}, p_{out}, l_{\min}, l_{\max}, R > \quad (5)$$

where:

p_{in} is the channel input port.
p_{out} is the channel output port.
$l_{\min}$ is the minimum latency in the channel.
$l_{\max}$ is the maximum latency in the channel.
R is the set of requirements.

At this point, the requirements that have to be taken into account in the SLA and the channels are not fully identified.

We can verify some properties, for instance every input port needed by components is supplied, or requirements are consistent. Nevertheless, the SLA does not allow verifying properties on the CPS simulation scheduling, we must now model the execution architecture able to run instances of the logical architecture of simulation.

V. Modelling an execution architecture of simulation

A. The simulation execution architecture

In this section, we want to define a model of SEA. This SEA can be implemented in several ways from a simple program with a single thread to a complex distributed architecture, thus our model must be abstract enough to represent this diversity.

An execution architecture of simulation can be viewed as a non-empty set of logical processors, with logical processors being able to execute the components defined in § IV-C, as periodic tasks.

Multiple models can be executed by a single logical processor, nevertheless, the logical processors respect the notion of time and space partition mentioned in § IV-A. In this work, we will call the process of binding multiple components to a logical processor a clustering. Running the SEA implies the distribution and clustering of components.

B. Introducing scheduler and communications

The tasks in a logical processor exist in a sequential domain, while logical processors exist in a concurrent domain. Moreover, depending on the implementation of the architecture, the concurrent domain might be a parallel domain, where logical processors can run totally simultaneously[1].

The components in logical processors must be able to communicate:

- Intraprocessor communications, direct between components.
- Interprocessor communications, occurring during logical processors synchronisation phases.

[1]We borrowed those notions of sequential and concurrent domain from the VHDL [9].

The different kinds of communication can have different natures, e.g. shared memory or network communication, implying a difference of performances.

The distribution/clustering of components on logical processors, and the communications create the notion of resources needed to express a scheduling, and the components are the schedulable tasks.

We are able to express the SEA as a two-level scheduler:

- A global scheduler: scheduling local schedulers.
- Local schedulers: scheduling tasks.

The global scheduler has no view on local schedulers tasks. There is no direct synchronisation between two tasks when they are on two different logical processors.

More specifically, we define a simple ADL considering logical processors and tasks, depicted in fig. 2, as the following:

$$\text{SEA} =< P, gs, c > \quad (6)$$

Where:

P is the set of logical processors, with a logical processor p defined as the following:

$$p =< T, ls, c > \quad (7)$$

where:

T is the set of periodical tasks.
ls is the local scheduler.
c is the type of intraprocessor communication between tasks in the same logical processor.

gs is the global scheduler.
c is the type of interprocessor communication between tasks in different logical processors.

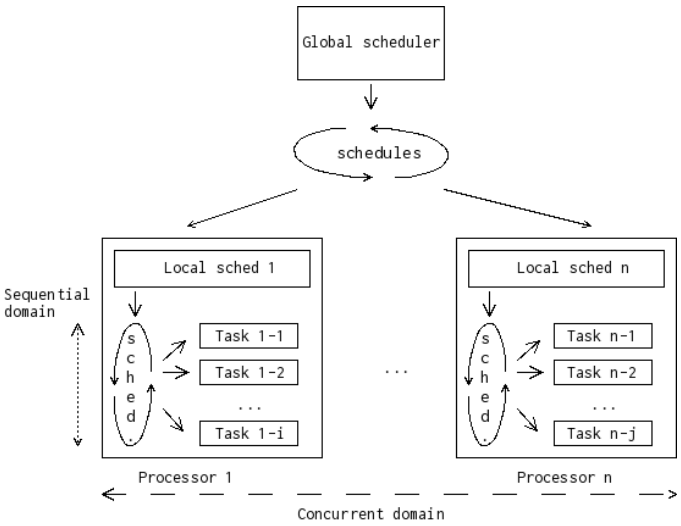

Figure 2: The SEA with its double level of scheduling.

VI. From sLA and sEA to simulation implementation

The distributing and clustering of schedulable components from an sLA on an sEA implementation is done in two stages: partitioning and mapping.

In this section, we discuss about the impact of partitioning the sLA, and mapping this partition on an sEA, and we will illustrate the implementation of the execution architecture with two general purpose architectures for distributed computer simulation systems, HLA/CERTI and DSS.

A. sLA *partitioning and* sEA *mapping*

In § IV-C we stated that our components have ports connected through channels. We want to divide our set of components into subsets, allocating a logical processor for each subset, and each one of the channels will be an interprocessor or intraprocessor communication when ported in an sEA, depending on partition and mapping.

In [10], the notions of partitioning and mapping on modular parallel literature are treated in order to reduce the number of switching elements and to minimize communication times.

We reuse these notions in this paper, with different constraints and objectives. For instance, due to the minimum and maximum latencies on channels, we are not looking for methods to reduce communication times, but for ensuring the consistency between latency constraints and partitioning/mapping.

The partitioning of sLA consists in splitting the set of sLA components into unordered subsets. From this partitioning, we are able to identify the type of channels that will be instantiated between components.

During this step, if we already have information about the sEA implementation, we can eliminate some partitions.

Fig. 3 illustrates some possible partitions for a single set of components. Ultimately, we can use set notation in order to represent the partitions. Regarding fig. 3, the partitions are:

- partition 1: $\{\{a,b,c,d\}\}$;
- partition 2: $\{\{a,b\},\{c\},\{d\}\}$;
- partition 3: $\{\{a\},\{b\},\{c\},\{d\}\}$.

Nevertheless, these partitions are not yet linked to an execution. This occurs during the mapping to sEA step.

As stated in § V-A, the set of tasks in a logical processor is ordered. Mapping components from a partition to tasks from a logical processor implies the definition of a sequence. Considering partition 2, the ordering of set $\{c\}$ and $\{d\}$ are straightforward, but there are multiple solution for $\{a,b\}$: $\{a,b\}$ or $\{b,a\}$. This is where the problem of the algebraic loop discussed in § IV-B is treated.

The ordered sets of tasks are sequentially executed by the local schedulers of logical processors, while the unordered set of logical processors is executed by the global scheduler.

To be more formal, we can describe the partitioning and mapping as the following function definitions:

- Let C be the set of sLA components.

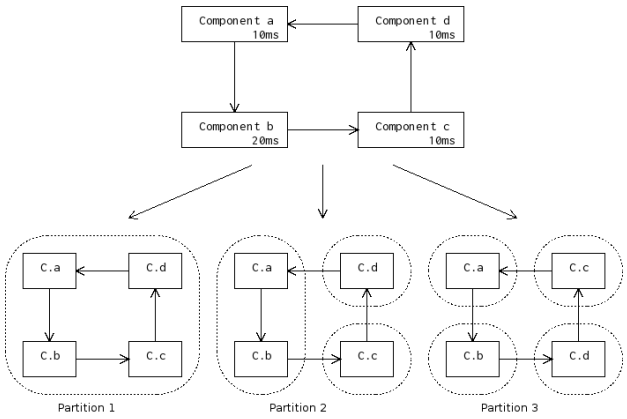

Figure 3: Example of partitions from a single set of components

- Let S be the set of C powerset cartesian squares $\wp(C)^2$, such that, for any S element s, the junction of s elements is C, and the superposition of s elements is empty.
- Let O be an ordered set of tasks.
- Let P be an unordered set of O.

$$partitioning : C \rightarrow S \tag{8}$$

$$mapping : S \rightarrow P \tag{9}$$

Depending on the sEA implementation, we are now able to verify requirements, such as the channel latencies requirements, and to adapt our partitioning and mapping.

Specifically, the decision of clustering components in a same subset are indirectly driven by the sEA implementation limitations, and component implementations, since the sLA does not consider execution. The mapping of partitions in the sEA implementation might lead to identification of partitions that are impossible to execute.

For instance, different components will have different Worst Case Execution Times (WCETs). Depending on these WCETs, and the sEA logical processor capabilities, we are able to check that a given partition is executable or not. Another example is that interprocessor and intraprocessor communications have different costs. Once the sEA implementation is identified, we know the cost of these communications, we can then verify the latency requirements.

If the mapping is theoretically possible, but technically impossible, then we have to reiterate at the partitioning step.

B. sEA *implementation*

1) Implementation with HLA: High Level Architecture (HLA) is a standard from the IEEE, for software architecture [11]. This standard defines methods and a framework to build global simulations comprised of smaller simulations, the federates. The HLA federates communicate through a RunTime Infrastructure (RTI) [12], and using publication/subscription mechanisms to exchange data. In this paper, we consider the CERTI implementation [13] for the RTI.

Model instances in the same federate run sequentially. Model instances in different federates run concurrently.

- The models are components of simulation hardcoded or imported into a federate from a library.
- The logical processors and local schedulers are the federates.
- The global scheduler is composed of the RTI Gateway (RTIG) and RTI Ambassadors (RTIAs).
- The intraprocessor communication is shared memory.
- The interprocessor communication is network communication through RTIG and RTIA.

Fig. 4 illustrates the implementation of the execution architecture of simulation with HLA/CERTI, with the partition 2 of fig. 3, considering the use of one computer for the two first logical processors, and one computer for the third one.

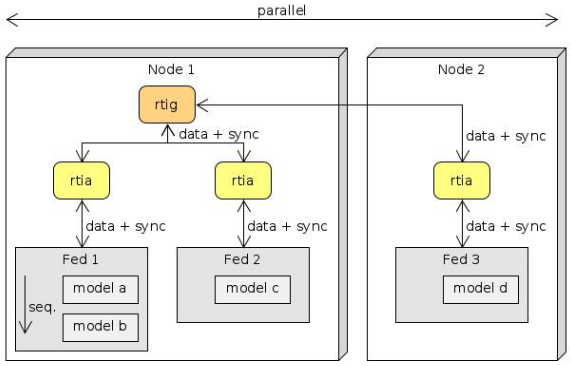

Figure 4: Illustration of the HLA/CERTI execution architecture.

2) Implementation with DSS: Distributed Simulation Scheduler (DSS) is an Airbus framework, scheduling AP2633 models: Airbus simulation model containing entry points, state machine, and variable needed for scheduling.

The major component of a DSS simulation is its configuration file. This file contains the AP2633 models used, with their location and execution frequency.

DSS implements two kinds of actors:

- a Central controller (CC) computing short cycles and long cycles from information given by the local schedulers, choosing the synchronization period, and managing the local scheduler executions.
- Local controllers (LCs) driven by the global scheduler, executing sequentially each AP2633 model provided depending on the configuration.

AP2633 models in a same LC are executed sequentially, and LCs are running concurrently.

The LCs periodically rerun the models, depending on the model periods. This execution leads to two concepts:

- The concept of short cycles (or minor frames), the maximum period needed to ensure each models can be run when needed.
- The concept of long cycles (or major frames), the minimum period needed for a model execution pattern.

Depending on the given configuration, the CC of DSS will compute the synchronization periods (eq. 12).

To be more specific, we have:

$$short\ cycle(LC) = GCD(model.period | \forall model \in LC.models) \quad (10)$$

$$long\ cycle(LC) = LCM(model.period | \forall model \in LC.models) \quad (11)$$

$$syncro\ period(CC) = LCM(long\ cycle(LC) | \forall LC \in CC.LCs) \quad (12)$$

Where GCD is the Greatest Common Divisor and LCM is the Least Common Multiple.

Thus, a DSS distributed simulation with poor model distribution can lead to a long synchronization period, and potentially degraded results.

The implementation of an SEA with DSS is straightforward.

- The models are the AP2633 models, embedding implementations of components of simulation.
- The logical processors and local schedulers are the LCs.
- The global scheduler is the CC.
- The intraprocessor communication is shared memory.
- The interprocessor communication is P2P network communication and shared memory, on synchronization period defined by eq. 12.

Fig. 5 illustrates this implementation with the partition 2 of fig. 3, considering the use of one computer for the two first logical processors, and one computer for the third one.

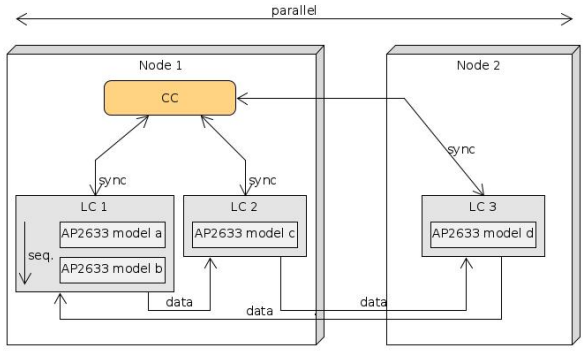

Figure 5: Illustration of the DSS execution architecture.

Table I summarize DSS and HLA/CERTI implementations of SEA.

SEA		HLA/CERTI	DSS
Sched.	Global	RTI	CC
	Local	Federate	LC
Comm.	Interproc.	Publication/Subscription	P2P on synchro period
	Intraproc.	Shared memory	Shared memory

Table I: DSS and HLA/CERTI SEA implementations comparison

VII. Illustration with a case study: R-ROSACE

A. Introducing the case study

Research Open-Source Avionics and Control Engineering (ROSACE) is a case study covering different steps from the conception to the implementation of a longitudinal flight controller [14][15]. A major challenge on designing the ROSACE controller is the need of interactions between control and software engineers.

This case study was extended in order to add redundancy on the controller, and the possibility to inject errors to test the redundancy, we named this case study R-ROSACE, for Redundant-ROSACE [15].

1) The sLA *of R-ROSACE:* In this case study, multiple aircraft functions are divided into components. For readability, we will use FCC for Flight Control Computer, and FCU for Flight Control Unit. We also directly use h for inertial altitude, V_z for inertial vertical speed, V_a for true air speed, q for pitch rate and a_z for body vertical acceleration.

We depict the full R-ROSACE sLA in table II, beside the Δ_{in} and Δ_{out} functions in details, that are too large for this paper. In this first case study, we do not consider requirements on the sLA or channels yet. Moreover, channels can be deduced from component ports (by homonymy), and latencies are 0s for minimum, and $+\infty$ time for maximum.

These components are then implemented into a library of models, following the Object Oriented Programming (OOP) paradigm.

B. R-ROSACE implementation with DSS

DSS schedules AP2633 models, thus we bind the models from the models library with AP2633 models as illustrated in fig. 6.

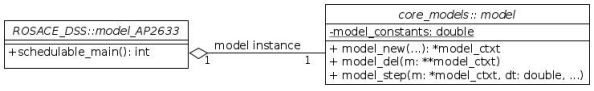

Figure 6: Structure of the binding of a model from the library with AP2633 model.

In run mode, AP2633 models receive and send data with global variables, the binding is straightforward.

1) The inputs have been updated by the local scheduler, we pass them to the model step function.
2) We write the model step function output in the outputs.
3) The scheduler retrieves our outputs and provides them to other AP2633 models, or other local controllers.

C. R-ROSACE implementation with HLA/CERTI

The mechanisms are divided into two levels: The CommonFederate, and the SpecializedFederate inheriting the CommonFederate.

The CommonFederate holds lists of subscribable and publishable objects, bound to a list of attributes, following recommendations from [16]. The CommonFederate advances its logical time depending on the minor frames through HLA services, and executing library models depending on their frequencies. SpecializedFederates, inheriting from CommonFederate, initialize the minor frame and the previous list, depending on the ports in and out linked to extraprocessor communications. The intraprocessor communication is a simple shared memory, instantiated with SpecializedFederate attributes.

Fig. 7 depicts the structural bindings of models in SpecializedFederates.

Component	P_{in}	P_{out}	f
Flight dynamics	δ_e, T	h, V_z, V_a, q, a_z	$200Hz$
Engine	δ_{th_c}	T	
Wiring	partial_$\delta^1_{e_c}$, partial_$\delta^2_{e_c}$, relay_$\delta^1_{e_c}$, relay_$\delta^2_{e_c}$, partial_$\delta^1_{th_c}$, partial_$\delta^2_{th_c}$, relay_$\delta^1_{th_c}$, relay_$\delta^2_{th_c}$,	δ_{e_c}, δ_{th_c}	
Elevator	δ_{e_c}	δ_e	
FCC1A	h_f, V_{z_f}, V_{a_f}, q_f, a_{z_f}, h_c, V_{z_c}, V_{a_c}, mode	partial_$\delta^1_{e_c}$, partial_$\delta^1_{th_c}$	$50Hz$
FCC1B	partial_$\delta^1_{e_c}$, partial_$\delta^1_{th_c}$, h_f, V_{z_f}, V_{a_f}, q_f, a_{z_f}, h_c, V_{z_c}, V_{a_c}, mode	relay_$\delta^1_{e_c}$, relay_$\delta^1_{th_c}$	
FCC2A	h_f, V_{z_f}, V_{a_f}, q_f, a_{z_f}, h_c, V_{z_c}, V_{a_c}, mode	partial_$\delta^2_{e_c}$, partial_$\delta^2_{th_c}$	
FCC2B	partial_$\delta^2_{e_c}$, partial_$\delta^2_{th_c}$, h_f, V_{z_f}, V_{a_f}, q_f, a_{z_f}, h_c, V_{z_c}, V_{a_c}, mode	relay_$\delta^2_{e_c}$, relay_$\delta^2_{th_c}$	
FCU	—	h_c, V_{z_c}, V_{a_c}	
Flight mode	—	mode	
h filter	h	h_f	$100Hz$
V_z filter	V_z	V_{z_f}	
V_a filter	V_a	V_{a_f}	
q filter	q	q_f	
a_z filter	a_z	a_{z_f}	

Table II: R-ROSACE sLA components

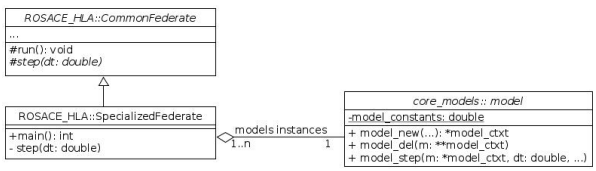

Figure 7: Structure of the binding of models from the library with HLA/CERTI federates.

D. R-ROSACE mapped partitions and executions

With both DSS and HLA/CERTI, we were able to run multiple mapped partitions (partially ordered sets), specifically:

- All components in a single logical processor, in arbitrary order: { Elevator, Engine, Flight dynamics, h filter, V_z filter, V_a filter, q filter, a_z filter, FCU, Flight mode, FCC1A, FCC1B, FCC2A, FCC2B, Wiring }
- All components in a single logical processor, in reverse order: { Wiring, FCC2B, FCC2A, FCC1B, FCC1A, Flight mode, FCU, a_z filter, q filter, V_a filter, V_z filter, h filter, Flight dynamics, Engine, Elevator }
- One component per logical processor: { {Elevator}, {Engine}, {Flight dynamics}, {h filter}, {V_z filter}, {V_a filter}, {q filter}, {a_z filter}, {FCU}, {Flight mode}, {FCC1A}, {FCC1B}, {FCC2A}, {FCC2B}, {Wiring} }
- One logical processor for physical components, and one for cyber ones: { {Elevator, Engine, Flight dynamics}, { h filter, V_z filter, V_a filter, q filter, a_z filter, FCU, Flight mode, FCC1A, FCC1B, FCC2A, FCC2B, Wiring } }

All these mapped partitions gave similar results without errors with DSS and HLA/CERTI, as expected, especially since we use this case study with a negligible set of requirements.

Nevertheless, even with quasi-nonexistent requirements, we found partitions that are not schedulable when interprocessor and intraprocessor communication have different cost, or when the mapping adds different latencies on similar data paths. More precisely, these mapped partitions are executable, but lead to visible errors in simulation. This situation happens when the FCC1 and FCC2 have different latencies between A and B. For instance, mapped partitions with the following partially ordered sets:

- ...{FCC1A,FCC1B,FCC2A},{FCC2B}...
- ...{FCC2B,FCC1A,FCC1B,FCC2A}...

Similar errors also occur when pairs of FCCs and wiring have different latencies. We identify here a coincidence constraint.

VIII. Conclusion

In this paper, we proposed a method to model CPS simulation components, in order to analyze the CPS simulation scheduling. We proposed a method based on two formalisms, the sLA, and the sEA, and functions between them. We prove that we can apply our early formalism to a case study, with an implementation on two distributed architectures. Using these early formalisms, we can express a CPS simulation scheduling as a partially ordered set of components, partitioned and mapped on logical processors, and deterministically estimate the time used in data paths for a given logical on a given execution architecture. With these formalisms, we noticed that some partitioning and mappings can lead to different delays on different data paths, independently of the execution platform implementation.

The R-ROSACE case study helped us to understand the impact of those delays, and we were able to identify some structures that do not allow different latencies on multiple data paths (for instance, the FCCs), resulting in untruthful simulations. We have to clearly identify how this phenomenum can happen, and to express it as requirements in our formalism.

Lastly, we will work on the identification of the requirements that can apply to the different level of our formalism, with more sophisticated case study.

In the long term, we will try to bring our formalism closer to the AADL or similar formalisms that can help us to integrate tools for static check. We expect to generate code and configuration, and being able to quickly iterate through schedulings of CPS simulation in order to find efficient ones, such as in [5], but with an abstract execution platform.

Acknowledgement

The work described in this paper is supported through an Industrial Agreement for Research Training — CIFRE — financed by the National Association for Research in Technology (ANRT). This work is also financed and supervised by Airbus, and supervised by the ISAE-SUPAERO, University of Toulouse.

References

[1] C. Landauer, "Flight Systems are Cyber-Physical Systems," Nov. 2012.
[2] E. L. Lawler, J. Karel Lenstra, A. H. G. Rinnoy Kan, and D. B. Shmoys, "Sequencing and scheduling: algorithms and complexity," in *Handbooks in OR & MS*. Elsevier Science, vol. 4, pp. 445–521.
[3] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*, 2nd ed. San Diego: Academic Press, 2000.
[4] M. Y. Chkouri and M. Bozga, "Prototyping of distributed embedded systems using aadl," *ACESMB 2009*, p. 65, 2009.
[5] J. Casteres and T. Ramaherirariny, "Aircraft integration real-time simulator modeling with AADL for architecture tradeoffs," in *Automation Test in Europe Conference Exhibition 2009 Design*, Apr. 2009, pp. 346–351.
[6] S. Han and H.-W. Jin, "Kernel-level ARINC 653 Partitioning for Linux," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, pp. 1632–1637.
[7] J.-B. Chaudron, D. Saussié, P. Siron, and M. Adelantado, "How to solve ODEs in real-time HLA distributed simulation," in *SISO (Simulation Interoperability Standards Organization)*, 2016.
[8] A. Tolk and J. A. Muguira, "The levels of conceptual interoperability model," in *Proceedings of the 2003 fall simulation interoperability workshop*, vol. 7. Citeseer, 2003, pp. 1–11.
[9] J.-M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard, *VHDL Designer's Reference*. Springer Science & Business Media, Dec. 2012.
[10] V. David, C. Fraboul, J. Y. Rousselot, and P. Siron, "Partitioning and mapping communication graphs on a modular reconfigurable parallel architecture," *Parallel Processing: CONPAR 92—VAPP V*, pp. 43–48.
[11] Institute of Electrical and Electronics Engineers and IEEE-SA Standards Board, *IEEE standard for modeling and simulation (M & S) high level architecture (HLA): object model template (OMT) specification*. New York: Institute of Electrical and Electronics Engineers, 2010.
[12] C. Gervais, J.-B. Chaudron, P. Siron, R. Leconte, and D. Saussié, "Real-time distributed aircraft simulation through HLA," in *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2012, pp. 251–254.
[13] B. Bréholée and P. Siron, "Certi: Evolutions of the onera rti prototype," in *Fall Simulation Interoperability Workshop*, 2002.
[14] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The ROSACE case study: from Simulink specification to multi/many-core execution," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 309–318.
[15] SchedMCore | Easy MultiCore Scheduling Analysis and Simulation. [Online]. Available: http://sites.onera.fr/schedmcore/
[16] F. Kuhl, J. Dahmann, and R. Weatherly, *Creating computer simulation systems: an introduction to the high level architecture*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.