CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations

Giovanni Liboni, Julien Deantoni

CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations

Giovanni Liboni *Safran Tech*
Modeling & Simulation
Rue des Jeunes Bois, 78114 Magny-Les-Hameaux, France
giovanni.liboni@safrangroup.com Julien Deantoni *I3S/INRIA Kairos*
Université Côte d'Azur
Sophia Antipolis, France
julien.deantoni@univ-cotedazur.fr

***Abstract*—The development of Cyber-Physical Systems involves several disciplines and stakeholders, which use heterogeneous models and formalisms to specify the system and make early validation and verification. In order to understand the behaviour emerging from the heterogeneous models, a collaborative simulation (co-simulation) can be used. To make it happen, the system engineer must define a correct coordination of the different executable models, which can be distributed over different enterprises. This is an important but difficult (and error prone) task that can not be done without information about the behavioral semantics of each model. In this paper, we introduce an integrated development environment which allows 1) to import different executable models (named simulation units), 2) to graphically connect them with rich connectors and 3) to generate a dedicated, accurate and efficient distributed co-simulation. The framework is based on Eclipse EMF for the modeling part and on ∅MQ for the deployment. It is named CoSim20.**

***Index Terms*—Coordination, Co-Simulation, Master Algorithm, Language Engineering**

I. Introduction

The increasing complexity of Cyber-Physical Systems (CPS) makes their development a challenging multi-expert effort. Each expert focuses on a specific aspect of the system and uses dedicated tools and languages tailored to her or his domain of expertise [1]. The cooperation among these different stakeholders is necessary due to the strong inter-dependency between every parts of the system. Additionally, the emergence of extended enterprises [2], where enterprises take advantage of their network organization with many partners to outsource some expertise, introduces Intellectual Property Rules between the different experts and their artifacts. Orthogonally, in order to speed up the time-to-market, Model-Based System Engineering promotes early verification and validation (typically by using simulation) [3]. In this context, using distributed collaborative simulation is more and more common since it enables the coordination of heterogeneous models and simulators (*i.e.*, simulation units), possibly without revealing Intellectual Properties (*e.g.*, by using black-box simulation units like in the FMI standard [4]).

Work partially founded by the GLOSE bilateral project between INRIA and Safran.

However, the heterogeneity of the languages and tools used by the different experts (from either the cyber and physical domain) makes the coordination between different simulation units a difficult task. A simple error in this task can lead to timing bias that corrupts the co-simulation results [5]–[13].

Several approaches, mostly based on the FMI standard, were proposed to define the coordination of heterogeneous simulation units [14]–[20]. However, due to the time-triggered nature of the co-simulation framework, they failed in ensuring a correct co-simulation embracing both cyber and physical models [9], [11]. To embrace the cyber and physical models, some knowledge about the behavior of the models is essential. Such information allows to explicitly define a correct coordination, *i.e.*, a coordination that does not introduce simulation time delays that can corrupt the results.

In this paper, we introduce CoSim20, an Integrated Development Environment (IDE) based on coordination interfaces and rich connectors, supported by advanced textual and graphical editors; and from which it is possible to automatically generate a distributed executable code of the co-simulation.

The next Section overviews other existing approaches and state the problem. In the Section III, we introduce the language and tool proposed to define an explicit coordination model. Then, in Section IV, we detail the distributed algorithm generated from the models. Finally, in Section V, we show a use case study, which shows the correctness and efficiency of the approach.

II. Problem Statement and Related Work

A collaborative simulation focuses on the orchestration among different simulation units that represent different parts of the same system, in order to better understand the emerging behavior of the system. A simulation unit is an executable entity, usually a black box, which may for instance encapsulate a model and its solver, a binary executable process or a proxy to a hardware device. The orchestration is of prime importance

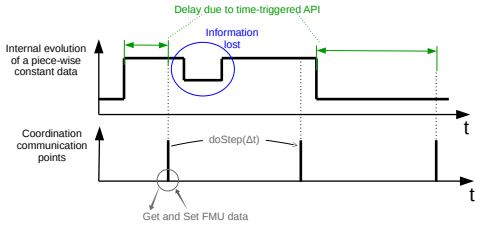

Fig. 1: Sources of errors in a time-triggered co-simulation algorithm.

because it defines the instant when a simulation unit exchanges data with other simulation units, *i.e.*, when it synchronizes its internal time in order to produce or consume data at the right timing.

Several approaches proposed different orchestrations (mainly based on the FMI Standard). A majority of them are variants of well-known Jacobi or Gauss-Seidel methods and are dedicated to (continuous) system of equations [14], [16]–[21]. All these algorithms implement a time-triggered coordination that does not correctly support the integration of heterogeneous simulation units as found in Cyber Physical Systems; *i.e.*, they do not encompass *Cyber* simulation units which are based on discrete time / discrete event models. These simulation units usually embed so-called piece-wise constant data where sampling creates bias. For instance, Figure 1 shows that time triggered sampling can create either information loss or delays. As shown in [9]–[12], such bias may invalidate the results of the co-simulation. Consequently, we consider a coordination as *correct* when no timing bias like the ones of Figure 1 are introduced.

Additionally, the emergence of extended enterprises implies that simulation units may be distributed in different locations, in particular where the expertise is. This is consequently important to reduce communication between the different simulation units to better support distributed coordination.

The reader should note that the problem in Figure 1 is not related to the sample rate but rather on an inappropriate communication schema between the simulation unit and the coordination. One can increase the sample rate but it decreases the performances (by introducing more message exchanges between simulation units) and reduces the delays due to the time-triggered API without removing them (see section V).

In order to define a correct communication schema, the system engineer must have some information about the behavioral semantics of the simulation units. To thwart such problem, several works on component based architecture description languages and coordination languages proposed to exhibit in an interface partial information from the (black box) components [22]–[26]. As an example, a nice proposition of behavioral interface has been recently published in [27]. Such interface must give enough information with respect to the use of the interface, without disclosing the internal model (to avoid Intellectual Property violation). In our case, it must provide enough information on how to coordinate the component. In the context of CPS, we proposed an assisted tool to easily import and define coordination specific interface,

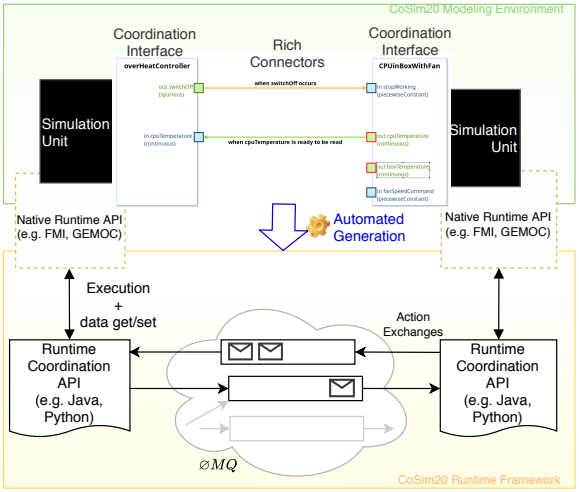

Fig. 2: Overview of the Cosim20 framework.

i.e., an abstraction of the simulation unit behavioral semantics which is suitable to define an appropriate coordination among simulation units. This coordination interface can be exploited to define a coordination algorithm. Coordination languages such as [26], [28]–[32] proposed sophisticated techniques to define a correct and efficient coordination among software components. However, they focused only on discrete-event coordination and this make these approaches unsuitable to the coordination of CPS simulation units. In this paper, we proposed a language based on rich connectors explicitly tailored to the specification of CPS simulation unit coordination.

The proposed language and framework are detailed in the next sections.

III. CoSim20 Modeling Environment

Figure 2 gives an overview of the proposed framework. There are two distinct parts: one part dedicated to the modeling of the co-simulation by using the language provided in the CoSim20 modeling environment; and one part dedicated to the co-simulation itself in the distributed CoSim20 Runtime framework. In the modeling environment, the development starts by importing a set of existing simulation units with their native runtime API. A coordination interface is then defined for each simulation unit, exposing appropriate properties like input/outputs and their nature (see next subsection). Then, the interface is used to define rich connectors which specify the intention of the coordination (see section III-B). Based on such model, the framework automatically generates for each simulation unit a Runtime Coordination API and an algorithm that handles the model execution through its native runtime API. The runtime coordination API makes use of a set of distributed queues (generated and configured during the automatic generation phase) to exchange actions to be realized by other simulation units in order to realize the coordination. A distributed master algorithm emerges from such exchanges.

A. Coordination Interface

The proposed Coordination Interface consists in a set of *ports* with their properties *e.g.*, direction, type and nature,

and a *temporal reference*. Each *port* represents a point of interaction between the internal variable of a simulation unit and the external environment *i.e.*, the coordination algorithm.

By looking at the various existing approaches and by realizing various systems, we realized that the information which is important from the coordination point of view depends on how and when the inputs and outputs are internally used (*i.e.*, read and/or written) by the simulation unit. Such information is abstracted by the *data nature* of the simulation units inputs/outputs. The different natures of the data found in CPS have been introduced in several approaches but are nicely summed up in [10], [33]. We adapted such specification and ended up with the following data natures:

Continuous : A variable is defined as *continuous* if its value is present for all $t \in T$, continuous and differentiable at any points of its range of definition;

Piecewise-continuous : A variable is defined as *piecewise-continuous* if its value is present at each instant but it is not continuous and not differentiable at some points (it presents discontinuities);

Piecewise-constant : The value is present at each instant but it presents discontinuities (typically due to internal or external assignments) and the value is constant between two discontinuities;

Transient : The value is present only at specific points in time and absents at other points in time. Transient data are usually associated with the notion of event or signal as found in synchronous languages.

Associated with the direction of the data, the data nature, as defined above, gives information on important points in time at which coordination should be done. Typically, if the output of a simulation unit is declared as *piecewise-constant*, then the coordination model is interested in the points in time at which the data is written; so that it can be updated in the connected simulation unit input port(s). If the data nature is *transient*, then important points in time are the instants when the data is present. If the data nature is continuous, then there is no real interest in any point in time; what is important is to sample data fast enough to avoid losing quick variation (*i.e.*, Nyquist-Shannon law should actually hold). Finally, in addition to the previous case, if the data nature is *piecewise-continuous*, then it is important to be noticed about any discontinuity in the signal. All in one, the data nature gives hints to the coordination of what is the event of interest on the data. Of course one can decide knowingly to sample *piecewise-constant* data, aware of potential problems as illustrated in Figure 1.

Due to the heterogeneity of formalism participating in the system, the time representation can be different across the simulation units. Even if physical time (with or without relativity effects) is mainly used in CPS simulation units it can also be polychronous. The temporal reference specifies the magnitude used to measure time (*e.g.*, the angle of a camshaft [34] or a distance).

Other (less important) properties like, for instance, the initial value of a port can also be defined.

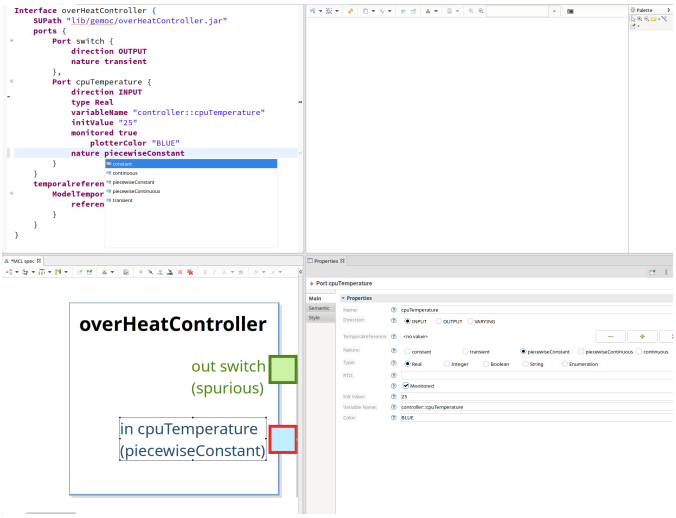

Fig. 3: Screenshot of the interface definition in the tool, with properties of ports.

An example of a coordination interface in the proposed framework is represented in textual and graphical form in Figure 3, together with the property windows of a port. Note that the properties are automatically imported if the *SUPath* point to a standard Functional Mockup Unit (FMU) [4]. Also in the definition of this interface, the user can decide if the value must be plotted during the co-simulation or not (*isMonitored* attribute on Figure 3).

B. Model Coordination Language

Once the interfaces are defined, we propose to use rich connectors to specify how the coordination between the different simulation units must be realized. A rich connector defines the conditions and constraints under which data exchanges have to occur. Inspired by coordination languages like REO [28] and Wright [26], the coordination model is defined by the behaviours defined in the rich connectors between ports from the coordination interfaces.

This section describes the basic syntax and semantics of these rich connectors, showing how they are combined to specify the coordination between simulation units in an unambiguous way.

The main element of MCL is a $MCLSpecification$ that imports Coordination Interfaces and contains a set of $Connectors$. A connector is defined based on the ports defined in the interfaces of different simulation units. More precisely, a connector defines three main concepts: the actual *interaction*, defined as a directed-graph topology of the system, the *triggering conditions*, defined as a set of conditions that trigger the interaction, and the *synchronization constraint*, defined as a temporal constraint that must hold before to perform the interaction.

An **Interaction** defines a relation between two or more ports defined in different interfaces. Each interaction can specify one or more assignment defined as a directed edge that connects a source port (defined as *output*) with one or more target ports

(defined as *inputs*). The global set of interactions is then used to generate the global topology of the system.

A **Triggering Condition** defines the condition at which the interactions must be realized. It is based on the data natures of the ports involved in the related interactions. For instance, if the data involved are all of *continuous* nature, so the triggering condition can specify a sample rate at which the data exchange occurs. As another example, if an interaction involves a *piecewise-constant* output port, the triggering condition can refer to each update of this port (instead of using a time-triggered approach). This coordination consequently removes delays or information losses (see Section V). The triggering conditions proposed in the framework are aligned with the expressiveness of predicates defined in [35] and appropriated to the data nature defined earlier in this section. More precisely, we defined the following triggering conditions:

1) *time condition*: the interaction is (classically) triggered periodically. It can be used on data of any nature;
2) *event condition*: the interaction is triggered when an event occurs on the output port. This allows to react at the exact time an event occurs in the simulation unit. It can be used on *transient* and *piecewise-continuous* data;
3) *data update condition*: the interaction is triggered each time the data of the output port is updated (*i.e.*, written) by the simulation unit. This allows to propagate fresh data to the connected simulation units. It can be used on *piecewise-constant* data;
4) *data read condition*: the interaction is triggered each time the data of the input port is ready to be read by the simulation unit. This allows to provide fresh data to the simulation unit. It can be used on *piecewise-constant* data.

Based on the triggering information, the framework is able to statically know if there exists a suitable coordination or not. For instance, if there is a loop between two simulation units linked with two connectors, both stating data update condition, then it is not possible to determine a suitable schedule for the simulation without forcing a rollback. Instead, the designer is proposed to sample one of the data, consciously introducing a possible coordination related delay. We believe this is an interesting feature to guide the designer in the coordination process. However, elaborated recommendations beyond non-correct coordination models are left for future works.

The **Synchronization Constraint** defines the relation between the temporal references in different simulation units. It specifies when the actual interaction can be accomplished. Usually, the most used synchronization constraint specifies that the internal time must be equal *i.e.*, synchronized, in order to exchange data between two models.

An example of rich connectors in the proposed framework are represented in textual and graphical form, as shown in Figure 4. In addition, the editor proposes completion, syntactic checking, etc.

From an implementation point of view, the modelling environment is defined on top of technologies from the Eclipse Modeling project [1].

[1]https://www.eclipse.org/modeling

```
load "overHeatController.mbi"
load "fanController.mbi"
load "CPUinBoxWithFan.mbi"

Connector cpuTemp ( from CPUinBoxWithFan.cpuTemperature to fanController.cpuTemperature)
    when every 5 CPUinBoxWithFan.t
    sync fanController.t = CPUinBoxWithFan.t
    do
        CPUinBoxWithFan.cpuTemperature -> fanController.cpuTemperature

Connector fanSpeed ( from fanController.fanSpeedCommand to CPUinBoxWithFan.fanSpeedCommand)
    when every 5 CPUinBoxWithFan.t
    sync fanController.t = CPUinBoxWithFan.t
    do
        fanController.fanSpeedCommand -> CPUinBoxWithFan.fanSpeedCommand

Connector cpuTemp2   ( from CPUinBoxWithFan.cpuTemperature to overHeatController.cpuTemperature)
    when value on overHeatController.cpuTemperature is ReadyToRead
    sync overHeatController.t = CPUinBoxWithFan.t
    do
        CPUinBoxWithFan.cpuTemperature -> overHeatController.cpuTemperature

Connector switchCPUState ( from overHeatController.switch to CPUinBoxWithFan.stopWorking)
    when event on overHeatController.switch occurs
    sync overHeatController.t = CPUinBoxWithFan.t
    do
        CPUinBoxWithFan.stopWorking -> not CPUinBoxWithFan.stopWorking
```

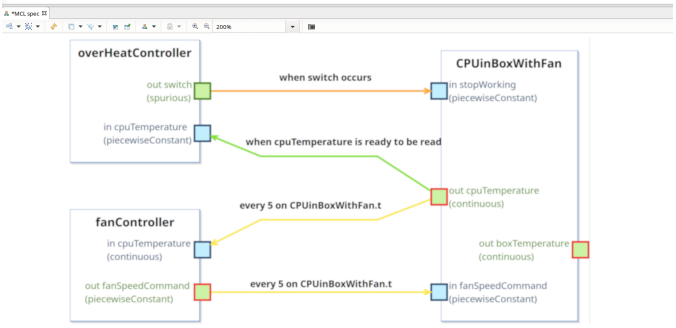

Fig. 4: Screenshot of the definition of rich connectors in the proposed tool.

IV. CoSim20 Runtime Framework

In this section, we present a distributed algorithm generated from the information of the modeling environment introduced in the previous section. More specifically the source code implementing the algorithm is generated using the coordination specification contained in the rich connectors and the coordination interfaces.

Before falling into the details of the algorithm presented in Listing 1, we introduce the general idea.

The main idea is that a connector creates an *Initiator*/*Follower* relationships on the ports it connects, depending on its triggering conditions. An initiator port initiates the data exchange and choose the time at which it occurs. On the opposite, a follower port waits for the initiator and does not know about the time at which the data exchange must occur. For instance, the triggering condition of the connector *cpuTemp2* in Figure 4 is "**when value on** `overHeatController.cpuTemperature` **is ready to be read**". This means that each time the overHeatController simulation unit internally reads the cpuTemperature variable, it pauses just before the reading so that a fresh value can be provided to it (see [35] for details). In this case, the target of the connector initiates the data exchange by querying for a value at a specific point in time (to the related follower ports, *i.e.*, the source of the connector). For this mechanism to work correctly, it requires that the simulation unit which exposes the follower ports did not simulate beyond the point in time at which the value is queried. More generally it means that each simulation unit that exposes a follower port

should not simulate beyond the minimum point in time of the simulation units exposing the initiator ports. We name this point in time the *Temporal Horizon* of the simulation unit. Rephrased, it means that a simulation unit which has at least a follower port can never simulate beyond its temporal horizon to be able to correctly handle queries from its connected initiator ports. Sometimes, it is possible to take advantage of *deterministic triggering conditions*. For instance, the triggering condition of the *fanSpeed* connector on Figure 4 is "**every** 5 **on** `CPUinBoxWithFan.time`". In this case, statically, the follower ports know when the data exchange will happen (every 5 time units of the CPUinBoxWithFan simulation units). We refer to such ports as *DeterministicFollowerPort*.

The proposed algorithm runs in parallel for each simulation unit (in the runtime coordination interface of Figure 2). The list of initiator ports, follower ports and deterministic follower ports are parameters of this algorithm, generated from the modeling environment (see line 1 in Listing 1). The algorithm also relies on (1) the native runtime interface of the simulation unit to perform the computational concern and (2) on a distributed communication technology to exchange actions to be performed across the system by other simulation units. It means that an initiator port will send actions to be done by the simulation unit of the associated follower port. These actions are stored in the simulation *todo* list, which is a list of actions to be done, sorted according to the time at which action must be done. The next action in the *todo* list is the action with the smallest time. An action is a request to the simulation unit. There are three kinds of action: *publish(data, time)* to ask a simulation unit to publish a data on a port at a specific time, *set(data, value, time)* to ask a simulation unit to internally assign a data to $value$ at a specific point in time and *updateTH(time)* to update the temporal horizon of the simulation unit.

Before to actually run the co-simulation, it computes, according to the list of its initiator port triggering conditions, the predicate at which it is mandatory to pause the simulation. For instance, if we consider the triggering condition of connector *cpuTemp2* already explained, the predicate requires that the simulation must pause when the cpuTemperature variable is (internally) ready to be read. The conjunction of such predicates required by the initiator ports is then constructed (line 2).

After the initialization and while the simulation is running, the next action in the *todo* list is taken (line 4). Note that, a simulation unit can assign itself actions to do, typically if it possesses initiator ports for which actions to be done are a priori known (*e.g.*, to publish a data on a port at every 5 time units).

Each action must be done at a specific point in time, this is the current action time attribute. If the time of the current action is the current simulation time (*i.e.*, the time at which the simulation unit is actually paused) then it performs the action (see lines 5 and 6 of Listing 1).

If the current action is in the future, the simulation unit has to check if it can actually simulate or not (lines 7 and 8). For that, we first check, according to the current action, initiator predicates and deterministic follower ports, what we have to do in the future and how much we can simulate. If we cannot advance in time (the max step size is 0) it means that we have to wait for a new temporal horizon from another simulation unit (*i.e.*, we wait for the temporal horizon from a simulation unit connected to the actual simulation unit through an initiator/follower relation) (line 10). Also, before to wait, we have to reschedule the current action by putting it in the *todo* list (line 9).

If the max step size is greater then 0, then it is add to the initiator predicates (line 12) and the simulation is restarted ($doStep$ line 13). When the simulation pauses, the now variable is updated according to the stop condition of the doStep call. This stop condition makes explicit the reason why the simulation actually stopped. Pragmatically it refers to the part of the predicate that became true and the algorithm can consequently determine the actions to be submitted to other simulation units. For instance, if the stop condition tells that the simulation units are ready to read the $cpuTemperature$ (see connector $cpuTemp2$) then the $publish(cpuTemperature, now)$ action is sent to the simulation unit that contains the CPU temperature variable. Thanks to the temporal horizon mechanism (lines 8 to 10), the time in the simulation unit that will receive the action will be lower or equals to the local now sent in the $publish$ action. Eventually, the emitter of the action will receive a set action with the requested value of the variable at the correct time. Finally, depending on the stop condition reason, the current action may be accomplished or not. If not, it is rescheduled (lines 16 and 17).

Algorithm 1 Pseudo-Code for the Wrapper Coordination Algorithm.

```
1:  function COSIMULATE(Set<Port>initiatorPort, Set<Port>followerPorts, Set<Port>deterministicFollowerPorts)
2:      initiatorPred ← setOwnedInitiatorPredicate()
3:      while simulationIsRunning do
4:          currentAction ← todo.getNextAction()
5:          if now = currentAction.TH then
6:              realize(action)
7:          maxStepSize ← getNextStepSize(action)
8:          if maxStepSize = 0 then
9:              todo.add(action)
10:             waitTH()
11:         else
12:             predicate ← maxStepSize ∪ initiatorPred
13:             stopCondition ← su.doStep(predicate)
14:             now ← stopCondition.time
15:             submitActionsToOtherSU(StopCondition)
16:             if currentAction! = done then
17:                 todo.add(action)
```

If the coordination defined by the rich connector does not violate constraints (see the previous section) then this algorithm ensures that no action in the past will be present in the *todo* list of a simulation unit. In other words, there is no need to rollback. Additionally, as shown in the next section,

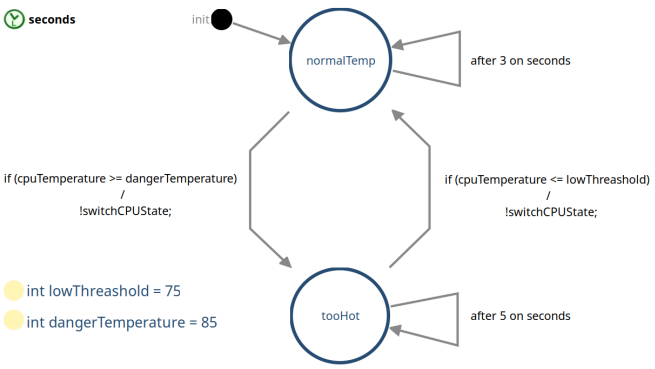

Fig. 5: Internal State Machine of the *ctrl* Simulation Unit.

such an algorithm drastically reduce the number of required communication between simulation units and avoid timing bias like the one of Figure 1.

From an implementation point of view, the data exchanges are realized by using the ∅MQ library[2]. This allowed us to define runtime coordination API in different languages and to abstract the network stack by using ∅MQ distributed queues. Moreover, since the topology is static and a priori known (defined in the modeling environment), we chose a point-to-point communication to avoid having a router that may introduce performance bottleneck due to the single-point-of-routing.

V. Validation Of The Approach And Results

As a representative case study, we used a CPU cooling system composed of 3 simulation units (see Figure 4). The *BoxWithCPU* and the *FanController* have been developed in OpenModelica[3] and then exported as two different FMUs. They define, respectively, a CPU in a box which is cooled by a fan, and the proportional controller of the fan speed. The heat is transferred according to the fan speed between the CPU and the box. These units represent the *Physical* part of our system. The *overHeatController* has been developed as a state machine in the GEMOC studio[4] and then exported as a simulation unit (see Figure 5). This unit represents the *Cyber* part of the system. This cyber part evaluates guards of outgoing transitions each time it enters a state. Consequently, in the $normalTemp$ state the $cpuTemperature$ variable is read every 3 seconds and in the $tooHot$ state the CPU temperature is read every 5 seconds.

We defined for each simulation unit its coordination interface (according to their internal semantics) and the coordination by using rich connectors. In this experiment, we defined two different coordination models: a (classical) time-triggered coordination, in which data are exchanged periodically between the simulation units; and a hybrid coordination model that takes advantage of data nature and rich connector triggering conditions proposed in our approach.

[2]https://zeromq.org/
[3]https://openmodelica.org/
[4]http://eclipse.org/gemoc

The hybrid coordination is defined in Listing V. There are four $Connectors$. The two first ones define a time triggered data exchange between the $plant$ and the pid simulation units. Both sampling periods are 5 units of time (*i.e.*, 5 seconds) as defined lines 6 and 12. For the third connector (line 17) the data exchange is done only when the $ctrl$ is ready to read the $cpuTemperature$. Finally, for the fourth connector (line 23) the interaction is realized each time the $switch$ internal event of the $ctrl$ occurs.

```
1   load "overHeatController.mbi" as ctrl
2   load "fanController.mbi" as pid
3   load "CPUinBoxWithFan.mbi" as plant
4
5   Connector cpuTemp
6   when every 5 plant.t
7   sync pid.t = plant.t
8   do
9     plant.cpuTemperature -> pid.cpuTemperature
10
11  Connector fanSpeed
12  when every 5 plant.t
13  sync pid.t = plant.t
14  do
15    pid.fanSpeedCommand -> plant.fanSpeedCommand
16
17  Connector cpuTemp2
18  when value on ctrl.cpuTemperature is
      ReadyToRead
19  sync ctrl.t = plant.t
20  do
21    plant.cpuTemperature -> ctrl.cpuTemperature
22
23  Connector switchCPUState
24  when event on ctrl.switch occurs
25  sync ctrl.t = plant.t
26  do
27    plant.stopWorking -> not plant.stopWorking
```

In opposite in the time triggered coordination, all connectors exchange data with a fixed sample rate. We realized experiments with a sample rate (Δt) of 1 second, 5 seconds and 10 seconds. The framework automatically generated the executable codes (see https://project.inria.fr/icise2020/ for videos and screenshots). Table I reports some results from the experiments: the number of communication points between *BoxWithCPU* and *OverHeatCTRL*, and the associated $CPUtemperature$ delays between the production of the value by the $plant$ and its reading by the $ctrl$. The CPU temperature delay is a cumulative delay caused by the non-synchronization between the internal sample rate of the $ctrl$ and the delay introduced by the sampling rate. The elapsed time is the clock time spent to simulate 30'000 seconds of the system (about 8 hours and 20 minutes of simulated time). What we can learn from this table is that the delay decreases when the sampling rate increases. This is expected since the data is fresher when sampled internally by the controller. However, the number of communication points and the time needed to simulate increases since more time is spent to communicate. In the opposite, the proposed approach, taking advantage of the rich connector, allow to remove the temporal delay by exchanging data only when required by the internal semantics of the simulation unit. In this case, we can see that the number of communication between the simulation units is reduced to its strict minimum to have 0 delay in the $CPUTemperature$.

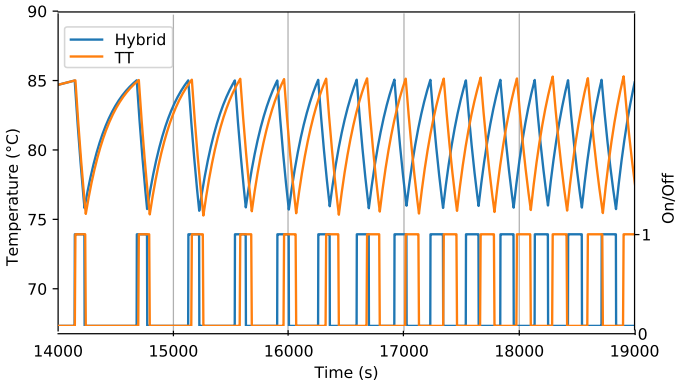

Fig. 6: Comparison betweent the results obtained by using Time-Triggered (TT) and Hybrid coordination. The Δ step size is set at 5 seconds.

TABLE I: Experimental number of Communication Points and Delay on the CPU temperature variable for the Time-Triggered (TT) coordination and the Hybrid coordination.

		CPU Cooling System Simulation				
		Communication Points Per Variable (#)		CPUTemperature Delay (seconds)		Elapsed Time
	$\Delta Size$	$isStopped$	$Temp$	$Avg.$	Max	(seconds)
TT	1	30000	30000	82	190	335
	5	6000	6000	792.7	890	138
	10	3000	3000	2361.8	1969	89
CoSim20	1	100	9022	0	0	210

Since communication points are reduced to their minimum, the performance is the one required to have correct results.

Additionally, Figure 6 gives a graphical representation of the $CPUTemperature$ delay on variables that measure the temperature (top) and the switch of the fan (bottom). Compared to the values obtained by the hybrid approach, Time-Triggered values are shifted on the right due to the introduced cumulative delay. As already explained, to minimize this issue it is possible to reduce the Δ step size, but the number of communication points and the simulation time will respectively increase. It is important to see here that a coordination which does not consider the data nature of input/output can introduce delay and consequently provide erroneous results.

VI. Conclusion

In this paper, we proposed an integrated modeling environment dedicated to the modeling of co-simulation. It provides rich editors (with completion, syntax check, graphical editing, etc) to support a language based on rich connectors. Rich connectors enable the definition of correct coordination between simulation units of Cyber Physical Systems. Additionally, the IDE automatically generates code for distributed co-simulation. The generated code is based on point-to-point coordination between simulation units. A case study shows the different benefits of the proposed framework. First, by proposing appropriate connectors it allows the designer to define a correct coordination, *i.e.*, coordination for which the co-simulation does not introduce unexpected delays. This is important since early V&V should not be biased due to the co-simulation framework. Second, it reduces the communication between the units to their strict minimum to ensure correctness. Less communication means better simulation performance. Finally, the high degree of automation in the framework removes the time-consuming task of writing a correct coordination. Many future works are envisioned for the framework. First, we are discussing with Safran to put the framework open source so that it helps to create a community beyond the company itself. Second, we plan to add features in the IDE to represent the actual network and help in the deployment of the different simulation units on the different nodes (according to the expected number of communications between nodes). Finally, amongst other future work, we would like to investigate how to embed rich connectors in existing model-based system engineering tools.

References

[1] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J. Jézéquel, and J. Gray, "Globalizing modeling languages," *Computer*, vol. 47, no. 6, pp. 68–71, June 2014.

[2] H. Jagdev and J. Browne, "The extended enterprise—a context for manufacturing," *Production Planning & Control - PRODUCTION PLANNING CONTROL*, vol. 9, pp. 216–229, 04 1998.

[3] J. A. Estefan *et al.*, "Survey of model-based systems engineering (mbse) methodologies," *Incose MBSE Focus Group*, vol. 25, no. 8, pp. 1–12, 2007.

[4] Modelisar, "FMI for Model Exchange and Co-Simulation," July 2014. [Online]. Available: https://fmi-standard.org/downloads#version2

[5] C. Thule, C. Gomes, J. Deantoni, P. G. Larsen, J. Brauer, and H. Vangheluwe, "Towards the Verification of Hybrid Co-simulation Algorithms," in *Workshop on Formal Co-Simulation of Cyber-Physical Systems (SEFM satellite)*, Toulouse, France, Jun. 2018. [Online]. Available: https://hal.inria.fr/hal-01871531

[6] S. Mustafiz, C. Gomes, H. Vangheluwe, and B. Barroca, "Modular design of hybrid languages by explicit modeling of semantic adaptation," in *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, April 2016, pp. 1–8.

[7] F. Cremona, M. Lohstroh, D. Broman, M. Di Natale, E. A. Lee, and S. Tripakis, "Step Revision in Hybrid Co-simulation with FMI," in *14th ACM-IEEE International Conference on Formal Methods and Models for System Design*. Kanpur, India: IEEE, Nov. 2016.

[8] F. Cremona, M. Lohstroh, S. Tripakis, C. Brooks, and E. A. Lee, "FIDE: An FMI integrated development environment," in *31st Annual ACM Symposium on Applied Computing*. Pisa, Italy: ACM New York, NY, USA, 2016, pp. 1759–1766.

[9] J.-P. Tavella, M. Caujolle, S. Vialle, C. Dad, C. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol, "Toward an accurate and fast hybrid multi-simulation with the FMI-CS standard," in *21st IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Berlin, Germany: IEEE, Sep. 2016, pp. 1–5.

[10] J.-P. Tavella, M. Caujolle, C. Tan, G. Plessis, M. Schumann, S. Vialle, C. Dad, A. Cuccuru, and S. Revol, "Toward an Hybrid Co-simulation with the FMI-CS Standard," apr 2016. [Online]. Available: https://hal-centralesupelec.archives-ouvertes.fr/hal-01301183

[11] S. Centomo, J. Deantoni, and R. De Simone, "Using SystemC Cyber Models in an FMI Co-Simulation Environment," in *19th Euromicro Conference on Digital System Design 31 August - 2 September 2016*, ser. 19th Euromicro Conference on Digital System Design, vol. 19, Limassol, Cyprus, Aug. 2016. [Online]. Available: https://hal.inria.fr/hal-01358702

[12] G. Liboni, J. Deantoni, A. Portaluri, D. Quaglia, and R. De Simone, "Beyond Time-Triggered Co-simulation of Cyber-Physical Systems for Performance and Accuracy Improvements," in *10th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, Manchester, United Kingdom, Jan. 2018. [Online]. Available: https://hal.inria.fr/hal-01675396

[13] C. Gomes., B. J. Oakes., M. Moradi., A. T. Gámiz., J. C. Mendo., J. Denil., S. Dutré., and H. Vangheluwe., "Hintco – hint-based configuration of co-simulations," in *Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH,*, INSTICC. SciTePress, 2019, pp. 57–68.

[14] T. Schierz, M. Arnold, and C. Clauß, "Co-simulation with communication step size control in an fmi compatible master algorithm," in *Proceedings of the 9th International MODELICA Conference; Munich; Germany*, no. 076. Linköping University Electronic Press, 2012, pp. 205–214.

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

[16] B. Wang and J. S. Baras, "Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems," in *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, ser. DS-RT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 33–40. [Online]. Available: http://dx.doi.org/10.1109/DS-RT.2013.12

[17] B. Van Acker, J. Denil, H. Vangheluwe, and P. De Meulenaere, "Generation of an optimised master algorithm for fmi co-simulation," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, ser. DEVS '15. San Diego, CA, USA: Society for Computer Simulation International, 2015.

[18] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press, 2013, p. 2.

[19] V. Savicks, M. Butler, and J. Colley, "Co-simulating event-b and continuous models via fmi," in *Proceedings of the 2014 Summer Simulation Multiconference*. Society for Computer Simulation International, 2014, p. 37.

[20] J. Bastian, C. Clauß, S. Wolf, and P. Schneider, "P.: Master for co-simulation using fmi," in *8th International Modelica Conference*, 2011.

[21] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar, "Model-based integration platform for fmi co-simulation and heterogeneous simulations of cyber-physical systems," in *Proceedings of the 10 th International Modelica Conference; Lund; Sweden*, no. 096. Linköping University Electronic Press, 2014, pp. 235–245.

[22] I. Crnkovic and M. P. H. Larsson, *Building reliable component-based software systems*. Artech House, 2002.

[23] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 60–76, 1997.

[24] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages," *Advances in Computers*, vol. 46, no. C, pp. 329–400, 1998.

[25] D. Luckham, *Rapide: A language and toolset for causal event modelling of distributed system architectures*, 11 2006, pp. 88–96.

[26] R. J. Allen, "A formal approach to software architecture," Carnegie Mellon University, Tech. Rep. CMU-CS-97-144, 1997.

[27] D. Leroy, E. Bousse, M. Wimmer, T. Mayerhofer, B. Combemale, and W. Schwinger, "Behavioral interfaces for executable DSLs," *Software and Systems Modeling*, Apr. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02565549

[28] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, p. 329–366, Jun. 2004. [Online]. Available: https://doi.org/10.1017/S0960129504004153

[29] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in computers*, vol. 46, pp. 329–400, 1998.

[30] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011, cited By 164. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-79955558541&doi=10.1109%2fMS.2011.27&partnerID=40&md5=760d39c68c54d3989b85d37c252bad9a

[31] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, p. 96, 1992.

[32] N. Busi, R. Gorrieri, and G. Zavattaro, "On the expressiveness of linda coordination primitives," *Information and Computation*, vol. 156, no. 1-2, pp. 90–121, 2000.

[33] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for hybrid cosimulation standards," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 179–188.

[34] C. André, F. Mallet, and R. De Simone, "Modeling Time(s)," in *ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS/UML)*, ser. Lecture Notes in Computer Sciences, vol. LNCS 4735. Nashville, TN, United States: Springer, Oct. 2007, pp. pp. 559–573, the original publication is available at www.springerlink.com (http://dx.doi.org/10.1007/978-3-540-75209-7_38). [Online]. Available: https://hal.inria.fr/inria-00204489

[35] G. Liboni and J. Deantoni, "A semantic-aware, accurate and efficient api for (co-)simulation of cps," *4th Workshop on Formal Co-Simulation of Cyber-Physical Systems – conjointly with the 18th edition of the International Conference on Software Engineering and Formal Methods*, pp. 1–16, 2020.