A Semantic-Aware, Accurate and Efficient API for (Co-)Simulation of CPS

Giovanni Liboni, Julien Deantoni

A Semantic-Aware, Accurate and Efficient API for (Co-)Simulation of CPS

Giovanni Liboni[1,2] [0000−0002−0981−0750] and Julien Deantoni[1] [0000−0001−6962−7846]

[1] Université Cote d'Azur, CNRS/INRIA Kairos, Sophia Antipolis, France
julien.deantoni@univ-cotedazur.fr
[2] Safran Tech, Modeling & Simulation, Rue des Jeunes Bois, 78114 Magny-Les-Hameaux, France
giovanni.liboni@safrangroup.com

Abstract. To understand the behavior emerging from the coordination of heterogeneous simulation units, co-simulation usually relies on either a time-triggered or an event-triggered Application Programming Interface (API). It creates bias in the resulting behavior since time or event triggered API may not be appropriate to the behavioral semantics of the model inside the simulation unit. This paper presents a new semantic-aware API to execute models. This API is a simple and straightforward extension of the Functional Mock-up Interface (FMI) API. It can be used to execute models in isolation, to debug them, and to co-simulate them. The new API is semantic aware in the sense that it goes beyond time/event triggered API to allow communication based on the behavioral semantics of internal models. This API is illustrated on a simple co-simulation use case with both Cyber and Physical models.

Keywords: co-simulation · API · behavioral semantics

1 Introduction

Cyber-Physical Systems are a class of systems where computation parts (cyber) and the plant parts (physical) can not be developed in isolation since the behavior of one impacts the others. In this context, it is of prime importance that orchestration of software and physical processes are based on semantic models that reflect properties of interest in both [14]. Echoing this, such systems are usually developed by multiple stakeholders, which use domain-specific languages, tailored both syntactically and semantically to the domain of expertise [10].

One solution to address the orchestration of different processes is the use of co-simulation, where processes are computed/solved by dedicated tools and kept in synchronization by a coordination algorithm. Usually, co-simulation makes use of a common Application Programming Interface (API) to communicate more easily with the various tools. For instance, the Functional Mockup Interface (FMI [18]) proposes a homogeneous time-driven interface to realize a computation step on a solver; this is well suited to the simulation of continuous physical

processes. Another well-known co-simulation approach is HLA [2], which proposes a homogeneous publish-subscribe event-based API well suited to discrete event processes.

However, in order to keep during the orchestration the behavioral semantics specificity of each modeling language, it is important to avoid using an API that hides semantic specificity in order to homogenize. Not taking care of internal semantics during the orchestration of such processes may lead to wrong results, lack of accuracy, and bad simulation performance [16,25,19,9,11].

In this paper, we presented a versatile API, which can be tailored to the internal model under simulation. By using this API, it is possible to communicate with the simulation unit according to different semantics (e.g, time-triggered, event-triggered, mix); according to the internal semantics of the simulation unit. Additionally, it is also possible to ask the simulation unit to stop under specific conditions like for instance when crossing a threshold or when it reaches breakpoint (for debugging purpose).

The next section explains the problems and overviews of the solutions proposed by other approaches. In Section 3, we present the semantic-aware API and in Section 4, we illustrate its use and benefits on a case study. Finally, before to conclude in Section 6, we propose a small discussion about the approach in Section 5.

2 Problem Statement and Related Work

Collaborative simulation of Cyber Physical Systems relies on the data and time synchronizations between various simulation units; where a simulation unit is, generally speaking, an encapsulation of a system part execution. Depending on the co-simulation, a simulation unit can encapsulate different entities amongst which (but not limited to): a model and its solver, a program together with its virtual machine, a compiled executable code, a proxy to an existing system part (hardware/software in the loop). A co-simulation actually implements the coordination that should ensure a *correct*[3] synchronization between the data produced and required by different simulation units. The goal is to be able to understand the behavior emerging from the coordination of the different parts of the system; either for simulation or analysis. Usually, such parts were developed by different domain experts, who are using domain specific languages and tools tailored syntactically and semantically to their needs. Also, most of the time, a simulation unit is seen as a black box by the coordination to ensure Intellectual Property preservation.

In this context, various algorithms were proposed to realize the coordination; well known classics like the Jacobi or Gauss-Siedel algorithms but also many others variants [23,3,29,8,28,7,22,4,20]. It is worth noticing that all the proposed algorithms are time-triggered, i.e., simulation units are all executed for a specific predefined time-step. This is a surprising fact since from the 90's work

[3] this notion is defined later in this section.

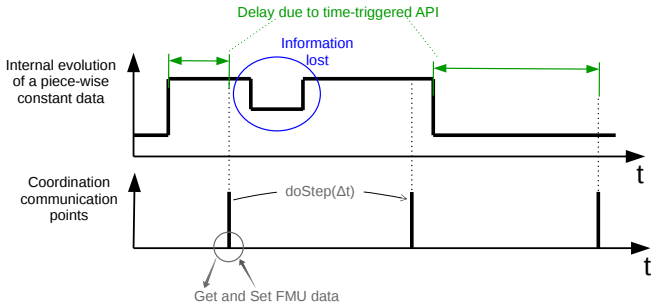


Fig. 1: Time-triggered simulation considered as incorrect.

about coordination languages and architecture description languages (ADLs) proposed more sophisticated techniques for the correct and efficient coordination among software components [21,17,13]. Actually, using the same API, being time-triggered or event-triggered on all the simulation units creates bias in the understanding of what the simulation units do. For instance, [25,9,16,26] identified the problem introduced by the use of a time-triggered API on *Cyber* simulation units where sampling the behavior at specific point in times creates artificial delays and loss of information from the coordinator point of view. Such delay can lead to error in results, to bad performances of the co-simulation, or both [16].

In this paper, *we consider a coordination algorithm as correct if it does not introduce any delays or lose information during the communication with the simulation unit.* Consequently, delays and information loss that appear when using a time-triggered API on a piece-wise constant data are considered incorrect (see Figure 1). Three important things must be noticed at this point. First, sampling a piece-wise constant value can make sense and does not necessarily introduce major problem; however, this should be done on purpose and not be the result of an inappropriate API. Second, there exists in many API (e.g., the FMI standard [18]) the possibility to avoid such delay, typically by roll-backing the simulation to a previous state and trying to locate the actual value change. This can be done only if the simulation can actually be rolled-backed; also this is costly in terms of simulation-time. Finally, third, it is worth noticing that the problem is broader than the simple illustrative case. As illustrated in [26], the coordination algorithm can have an impact on the correctness of the system.

The core of the problem was identified in several papers: it is not appropriate for any simulation unit to communicate only through a time-triggered or event-triggered API. In the literature, some approaches proposed to extend some existing API to fix a particular problem. This was for instance the case in [25] where they proposed to add a new parameter to the FMI time-triggered *doStep(Δt)* function. The new parameter is $nextEventTime$, a placeholder to store the time at which unpredictable events occurred. [16] went further by proposing to extend the FMI API with new functions that simulate until input

and output ports are respectively ready to be read or just written. Finally, the new features of FMI3.0 for hybrid co-simulation tries to aggregate such propositions (see chapter 5 of FMI3.0 development version `https://fmi-standard.org/docs/3.0-dev/#fmi-for-hybrid-co-simulation`).

However, in all these related works, the problem is not handled in its generality and they make specific cases of something that should be straightforward. In order to speak *correctly* with a simulation unit, you should be aware of its behavioral semantics and adapt the way to realize the *doStep* accordingly. As an abstraction of a simulation unit behavioral semantics, previous works proposed to focus on the nature of the inputs and outputs of the simulation units [8,24] like for instance *continuous*, *piecewise-continuous*, *piecewise-constant* or *spurious*. We believe this abstraction is very interesting and can be used as a basis for a semantics-aware API.

3 Proposition

We propose to consider the FMI time-triggered interface *doStep(Δt)* as a specific case where we ask a simulation unit to simulate until a specific predicate characterized by an amount of time spent in the simulation unit[4]. Following the same rationale, the proposed semantics-aware API can ask to a simulation unit to execute until a specific coordination predicate holds. The predicate must be expressed according to the information from the simulation unit behavioral interface (typically containing input/output nature, time representation, and solver capability [8,24,16,15]). Consequently, the general form of the proposed doStep API is:

`StopCondition` doStep(`CoordinationPredicate` p)

; where p expresses a condition under which the execution should pause, *i.e.*, the condition under which the doStep function returns. For instance, considering the input and output nature as defined in [24] (i.e., continuous, piece-wise constant, piece-wise continuous or spurious), the concept of predicate for a correct coordination can be defined as shown in the class diagram Figure 2

If the simulation unit supports only temporal predicates, then it corresponds to the FMI API. However, other coordination predicates have been defined. Here is a brief description of their meaning and their typical use case.

1. `TemporalPredicate` is a predicate that becomes true when the internal time of the simulation unit reaches the value of the predicate. This is the classical FMI predicate.
2. `UpdatedPredicate` is a predicate that becomes true as soon as the referenced variable, which must be a piece-wise constant output, has been assigned. It typically corresponds to example from Figure 1, which can then be managed without data loss, delays, or very small communication step size; i.e., in a correct way.

[4] Note that, in reference to study on Model of Computations [27] that this may be done only for timed simulation units.

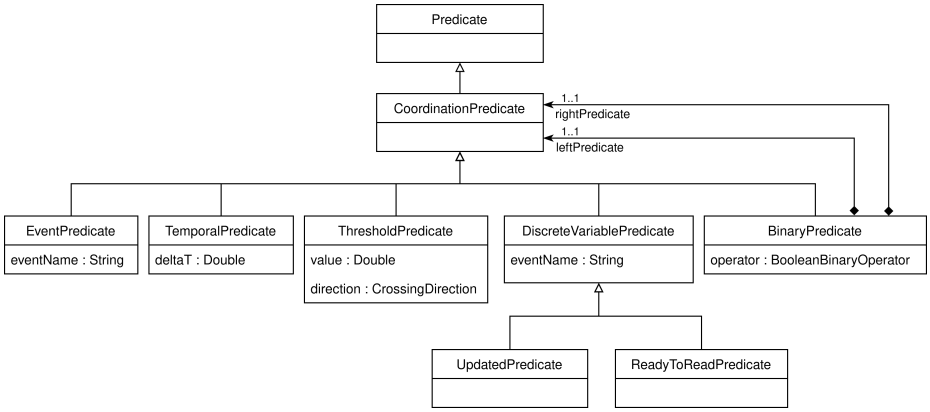


Fig. 2: Minimal but extendable set of predicates.

3. `ReadyToReadPredicate` is a predicate that becomes true just before the simulation units actually read the referenced variable. It is typically used if there is a need to provide an input to a simulation unit that actually reads (non necessarily in a deterministic way) this input at specific points in time. Instead of periodically providing the input data (consequently with unavoidable delays), the data is provided only when needed by the simulation unit.
4. `ThresholdPredicate` is a predicate that becomes true when the referenced variable crosses the defined threshold (according to the crossing direction[5]). It is typically used when a simulation unit is waiting for a specific threshold on a value from another simulation unit. Instead of periodically providing the input data (consequently with unavoidable delays) to be tested and possibly using rollback for more precision, the data is provided only when the condition is reached.
5. `EventPredicate` is a predicate that becomes true when the referenced event occurs. While this is in our implementation only used for cyber events, it may also be extended to encompass discontinuities or other kinds of events on (piecewise) continuous signals.
6. `BinaryPredicate` defines the disjunction of other predicates.

Finally, the proposed API also provides the classical function like for instance `loadModel`, `get/set Variable`, `get/set State` and `terminate`.

What is important is the (preliminary) definition of the coordination predicate, which is, according to our experiments, the minimal set of predicates to have an accurate coordination *i.e.*, without loosing any data, events or signals. Note that for now, we are only using the disjunction of predicates since it is not clear about the meaning of their conjunction. For instance, existing works about Event constraints suggest using Union or Inf/Sup constraints instead of AND

[5] It can be either from above to below, from below to above or both.

since they intrinsically embed a notion of order which is not existing into the classical Boolean operators [1,12].

To these predicates, many others could be added like for instance a discontinuity predicate that stops when a discontinuity is detected on a piece-wise continuous variable (see description of the Event predicate). Another more complex predicate could be a Büchi predicate, which is verified when a specific state-based observation occurs. There is no real reason to limit the kind of predicate that can be defined, as long as it makes sense according to the simulation unit execution semantics.

In other words, based on the simulation unit behavioral interface, one can speak about the simulation in terms of predicates which are relevant in the particular simulation units used in the co-simulation. For instance, considering a simulation unit interface of an untimed simulation unit, no temporal predicate can be used. In the same idea, if the simulation unit exposes only (piece-wise) continuous variable, then it should not be possible to refer to these variable updates (since it creates an undesired connection with the internal simulation unit discretization step). In short, the acceptable predicates for a specific simulation unit can be inferred from the simulation unit behavioral interface of such simulation unit. However, it is also important that each tool specifies the predicates it supports.

The value returned by the `doStep` function must allow the coordinator to understand why the simulation was actually paused, so that it can do the appropriate action. For instance, if the simulation unit was paused due to an `UpdatedPredicate`, then the variable that has been updated should be communicated to the appropriate simulation unit input (after being sure that the receiving simulation unit is at the same time than the emitting simulation unit, aligning the time if needed). For now, we used a simple form a `StopCondition` but it might be aligned with the `Predicate` class diagram. The Figure 3 shows a minimal proposition for a simple `StopCondition`. The *StopReason* is a predicate type defining why the simulation was paused; the *elementName* defines the referenced element link with the stop reason and the *stopTime* stores the internal time of the simulation unit when paused.

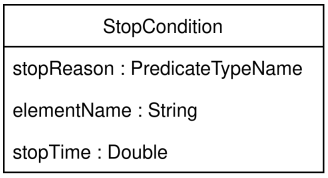


Fig. 3: Simple `StopCondition`, returned by the `doStep` function.

Remarque 1: This is not clear yet how the link should be made between the name of an exposed variable in the simulation unit behavioral interface and the actual variable inside the model under simulation. For now, we are using

qualified names instead of simple names like in the simulation unit behavioral interface. Similarly, for experimental facilities, we are using a Double to encode time in the co-simulation. It does not mean that the time is internally a double (since it may be encoded by super dense time for instance) but it provides a helpful homogenization of the time from the coordination point of view.

Remarque 2: According to our definition, FMI is a specific mold of our interface since it defines only (piece-wise) continuous variables and (and it does not allow for Threshold predicate injection). Consequently, the only acceptable predicate is a Temporal predicate.

We show in the next section how this API, implemented for language developed in the GEMOC studio [6], provides a simple way to gain in term of accuracy and performance during the coordination of multiple simulation units. However, in the next subsection, we overview how it can be used for other usages, typically debugging.

Example of Extension of the API for Debugging

In this subsection, we show an implementation experimented in the GEMOC studio to use the very same API for debugging. Our goal was to implement the functionality of an API as defined in the usual debugger. We consider this useful for the developer of one simulation unit when she/he wants to debug the simulation unit in the context of the other simulation units. For this reason, we considered that breakpoints are defined with another interface and considered only the way to execute the simulation unit. To define the new use of the interface, we simply defined the necessary Predicate for debugging (see Figure 4) and implemented the corresponding management of the Predicate in a wrapper. Details can be found here: `https://github.com/jdeantoni/cosimulationOfCpuHeatManagement`. Furthermore, it is interesting to realize that debugging equational simulation units could use a totally different notion of breakpoint. For instance, one could want to pause the simulation when the derivative of a specific output reaches a symptomatic threshold, in order to check different values in the system and try to understand what actually happens. In this case, Predicates should be defined accordingly.

Once again, we tried to provide an extendable simulation API, focused on co-simulation but suitable for different activities.

4 Case Study

We used the management of a CPU temperature as a simple but representative case study[6]. This system is made up of 3 simulation units (see Figure 5). *CPUinBoxWithFan* and *fanControler* have been developed in the OpenModelica tool[7] to respectively define the CPU in a box which is cooled by a fan and the controller

[6] the associated code can be retrieved from `http://i3s.unice.fr/~deantoni/cosim-cps2020`.

[7] `https://openmodelica.org`.

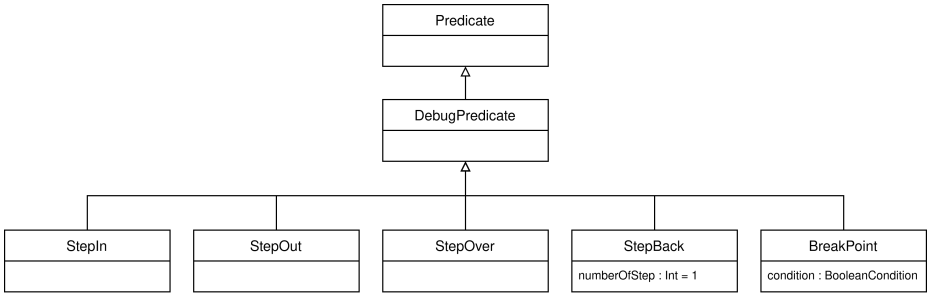


Fig. 4: Simple `StopCondition`, return by the `doStep` function.

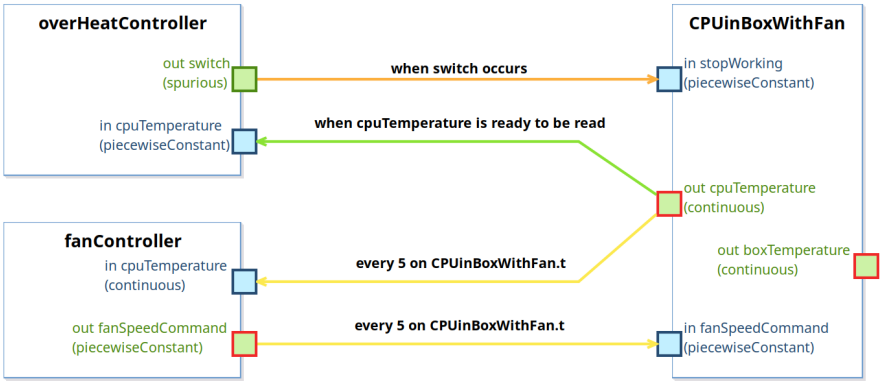


Fig. 5: Simple but representative case study for co-simulation.

of the fan speed (a simple Proportional controller). The heat between the box and the CPU is transferred according to the fan speed. The *overHeatController* has been developed as a state machine in the GEMOC studio[8].

In the *CPUinBoxWithFan* simulation unit, the CPU is activated as long as the *stopWorking* input is equal to false. When activated, the CPU produces heat, which is exchanged with the air of its box more or less rapidly depending on the *fanSpeedCommand* input ($\in [0..10]$ where at 0 the fan is stopped and at 10 the fan is at full speed).

In the *overHeatController* simulation unit, a state machine is defined. It monitors periodically (every 3 seconds) the *cpuTemperature* and if it exceeds a specific threshold, the *switch* event occurs and the state machine enters in a new state where it monitors the CPU temperature every 5 seconds. If it goes above a specific threshold, the *switch* event occurs and the state machine enters the first state (see Figure 6).

To connect the different simulation units we relied on strategies defined in [16]. Consequently the temperature from *CPUinBoxWithFan* to *overHeat-*

[8] `http://eclipse.org/gemoc`.

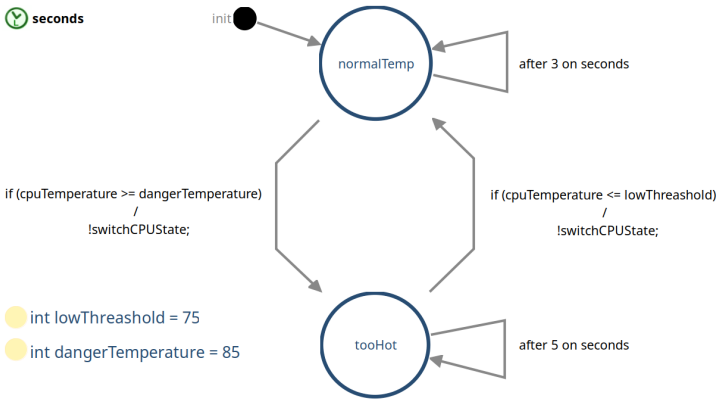


Fig. 6: Over Heat Controller state machine.

Controller is only exchanged when the later simulation unit is ready to read the data. Similarly, the change of the *stopWorking* input is only done only when the *switch* event occurs. Between the two simulation units obtained from Modelica, the connectors define classical time trigger communication.

Of course, we handled these different cases by using different Predicates in the *doStep* function call. However, one can notice that the coordination algorithm will not be generic anymore but dedicated to the topology of simulation units and the information on the connectors. For this specific use case, the coordination algorithm is provided on Listing 1.1. Lines 4 to 6, the predicate for the *overHeatController* simulation unit is defined as "the variable cpuTemperature is ready or the switch event occurs". Line 7, the *dostep* function is called and lines 8 to 16 the result of the function is managed. If the simulation was paused due to the variable cpuTemperature which is ready to be read, then a function (*simulateBoxAndFanControl* defined line 19) is called to set the *CPUinBoxWithFan* simulation unit at the same time as the over heat controller simulation unit. Once done, the expected value is exchanged between the FMU. If the simulation was paused due to the occurrence of the *switch* event, then the receiving simulation unit is at the time when the event occurred, so the *stopWorking* variable is changed. The temporal connector between the fan controller and the CPU, as defined in Figure 5, requires to simulate both models until a specific point in time. In lines 21 to 36, the simulation units must reach an *expectedTime*. If there is one (or several) intermediate temporal steps in between now and the expected time (i.e., $now\%5 = 0$ in our case), then the simulation units are simulated until this point in time and data are exchanged as expected.

Listing 1.1: Coordination Algorithm dedicated to the example on Figure 5 using the proposed interface

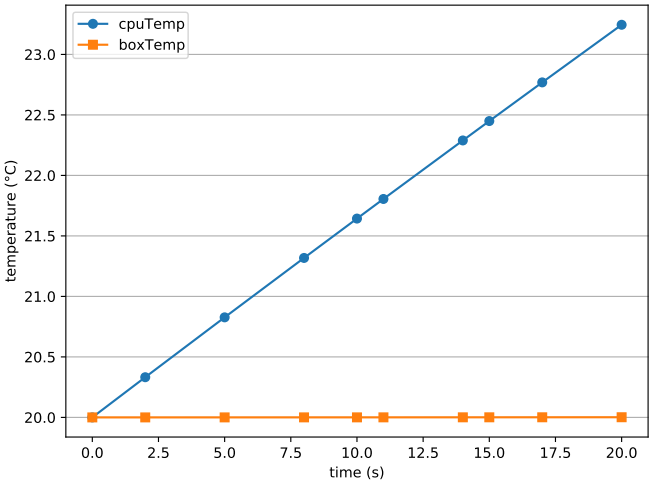
```
1  public void coSimulate(double endtime) {
       //now = 0; localIsStopped = false;
3    while (now < endTime){
```

```
        ReadyToReadPredicate r2rp("cpuTemperature");
        EventPredicate ep ("switch");
        BinaryPredicate bp (r2rp, ep);
        StopCondition sc = controlerSU.doStep(bp);
        if (sc.stopReason == READYTOREAD) {
            simulateBoxAndFanControl(sc.stopTime);
            double cpuTemperature = c.boxSU.read("cpuTemperature");
            controlerSU.setVariable("cpuTemperature", cpuTemperature);
        } else { //event occured
            simulateBoxAndFanControl(sc.stopTime);
            localIsStopped = !localIsStopped;
            boxSU.write("stopWorking").with(localIsStopped);
        }
    }

    public void simulateBoxAndFanControl(double expectedTime) {
        double delta = expectedTime - now;
        while (delta + (now % 5) >= 5) { //\Delta t == 5 for each connector
            ↪ from boxSU and fanControllerSU
            double stepToDo = (5-(now % 5));
            boxSU.doStep(stepToDo);
            fanControllerSU.doStep(5);
            double cpuTemperature = boxSU.read("cpuTemperature");
            fanControllerSU.write("cpuTemperature").with(cpuTemperature);
            int fanCommand = fanControllerSU.read("fanSpeedCommand");
            boxSU.write("fanSpeedCommand").with(fanCommand);
            double boxTemperature = boxSU.read("BoxTemperature");
            now += stepToDo;
            delta = expectedTime - now;
        }
        if (delta > 0) {
            boxSU.doStep(delta);
            now += delta;
        }
    }
```

The results from the beginning of the co-simulation obtained with this setup are provided in Figure 7. The reader should notice that the points are only retrieved as specified in Figure 5, i.e., at the exact time it is needed to have a correct co-simulation. For instance on Figure 7, we can see that a first paused was realized by the overheat controller at time 2, i.e., which is the non deterministic time spent for the state machine to enter in the *normalTemp* state, where the guard of output transition is evaluated and consequently the CPU temperature is read. Then, pauses are realized every 5 seconds and every multiple of 3 (the reading period in the first state of *overHeatController*). This way, we reduce the number of communication points to their strict minimum to have a correct co-simulation and we avoid the delays introduced by the classical sampling strategy.

In the Figure 8, the first point in time is the one when the state machine switch from the *normalTemp* state to the *tooHot* state. It occurred at time 14679. Consequently, as long as the state machine remains in this state, data are retrieved every 5 seconds as specified in the temporal connectors and in the reading period from the state machine. However, since the state machine entered in the *tooHot* state at time 14679, then the simulation unit was paused after 5 seconds, i.e., at 14684, while the temporal connectors induce a pause every 5 seconds. We can see here that the internal semantics of the simulation is consistently exposed and took into consideration.



	Temperature (°C)	
Time (s)	CPU	Box
0	20	20
2	20.33212	20.00001
5	20.82698	20.00009
8	21.31794	20.00023
10	21.64328	20.00035
11	21.80536	20.00042
14	22.28902	20.00067
15	22.44940	20.00076
17	22.76890	20.00097
20	23.24500	20.00133

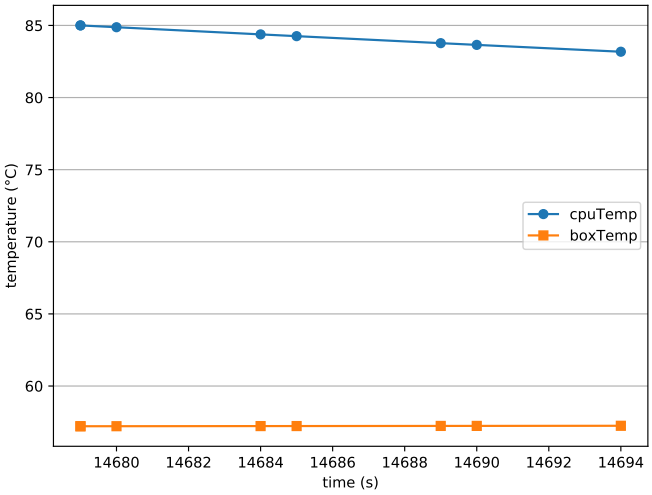
Fig. 7: Results obtained at the begin of the co-simulation.

Finally, in Figure 9, the simulation is run for 8 hours and 20 minutes (30000 seconds). For this simulation, we obtained 15023 communication steps without sacrificing accuracy over performance. If we were using a time-triggered interface and allowed an error up to 100ms, then we would have 300'000 communication step and a loss of accuracy. Additionally, we believe that the proposed interface is intuitive to use and may be extended for different purposes. In the next section, before to conclude, a small discussion about implementation is made.

5 Discussion

We argued that the proposed interface is extendable, efficient, and intuitive to use. In this section, we discuss some of these points according to our experiment in implementing the API in the GEMOC studio.

Concerning the implementation of the predicates, two main points can be addressed. First, its efficiency strongly relies on how the API is internally implemented. In our case we modified the code generation to generate a pause when needed. For instance, for the *Updated* predicate, all assignments are instrumented to create a pause. This has only a minor impact on performance. However, if the implementation is done in a wrapper where all micro steps are checked to see if a variable has been updated, then the execution may suffer from a slowdown. The same phenomenon happens for the Threshold predicate. If one sample the variable to check the crossing, the execution will be slow down and the exact point in time when the crossing occurs may be missed. It is better to inject the actual zero crossing in the model (typically in the equation set) to ensure better performance and accuracy. This is what is expected to be done in collaboration with Safran. Also, the implementation of the predicates must actually follow the



	Temperature (°C)	
Time (s)	CPU	Box
14679	85.00110	57.21210
14680	84.87561	57.21432
14684	84.37902	57.22311
14685	84.25629	57.22527
14689	83.77094	57.23384
14690	83.65101	57.23595
14694	83.17671	57.24429

Fig. 8: Results obtained when the controller enters in the *tooHot* state.

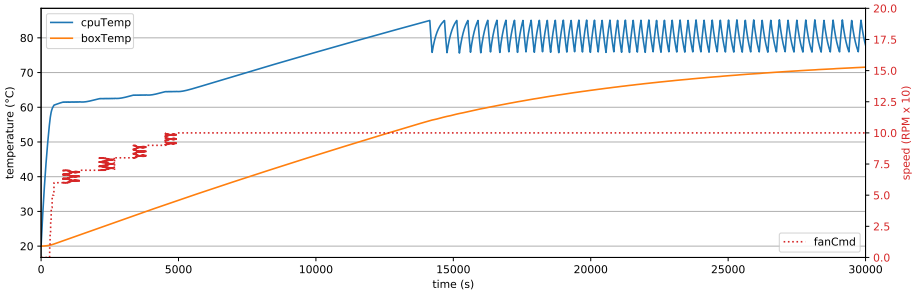


Fig. 9: Results obtained when running the coordination algorithm.

semantics of the simulation unit. For instance, if a simulation unit is executing a model developed in a synchronous language [5], then all the assignments should NOT be caught since according to the synchronous semantics, data are latched at specific points in time. In our implementation, we relied on annotations to provide flexibility on the exposed semantics. Consequently, the tool developer is in charge of providing the expected semantics.

Concerning the extension of the predicate, there are two minors points to take care of. First, it is important to rely on a mechanism to clearly specify which predicate is supported for a specific simulation unit. This may for instance be done in an artefact equivalent of the FMI model description. Second, there is a risk of an uncontrolled evolution of predicates, leading to a predicate tower of Babel. This is a long term issue and we believe there are few risks it happens. If the road to this situation is taken, it may be interesting to provide an official set

of predicate extension repository, where people can look for existing predicate before to create their own and where all predicates are put together.

6 Conclusion

We presented in this paper a new API, initially thought as a co-simulation interface. However, it can be used for different purposes. It uses the proposed notion of predicate to represent the condition under which a simulation unit must be paused. These predicates can be of different nature depending on the use of the API. In each case, it relies on the information provided about the simulation unit. In the co-simulation case, it relies on the data nature (continuous, piece-wise continuous, etc) exposed by the simulation unit. This information is an abstraction of the internal behavioral semantics of the simulation unit. We developed a case study where we showed how the API can be used in a semantic-aware way. The use of the API adapts the number of co-simulation steps to the internal behavior of the simulation units, keeping only the communication points required for a correct co-simulation. We believe such communication between the coordination algorithm and the simulation unit provides the basis for an analysis of a co-simulation.

In future works, we first want to focus on the automated generation of the coordination algorithm. As shown in Listing 1.1, the coordination is dedicated to a specific simulation and it may be tricky to write it by hand for a more complex system. Additionally, it becomes important to allow for the distribution of co-simulation. Our approach, by limiting the number of co-simulation steps to the minimum, is well appropriate to distribution. This is why we are actually finishing the development of the generator of distributed coordination algorithm based on our interface. Another future work concern the integration of such approach into a system engineering approach but this is a longer term work.

References

1. André, C.: Syntax and semantics of the clock constraint specification language. Tech. Rep. 6925, INRIA (2009)
2. Association, I.S., et al.: Ieee standard for modeling and simulation (m&s) high level architecture (hla)—framework and rules. Institute of Electrical and Electronics Engineers, New York. IEEE Standard pp. 10–1109 (2010)
3. Awais, M.U., Palensky, P., Elsheikh, A., Widl, E., Matthias, S.: 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. pp. 315–320. IEEE (2013)
4. Bastian, J., Clauß, C., Wolf, S., Schneider, P.: Master for co-simulation using fmi. In: Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany. pp. 115–120. Linköping University Electronic Press (2011)
5. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)

6. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the gemoc studio (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. pp. 84–89. ACM (2016)
7. Broman, D., Brooks, C., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of fmus for co-simulation. In: Proceedings of the Eleventh ACM International Conference on Embedded Software. p. 2. IEEE Press (2013)
8. Broman, D., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Requirements for hybrid cosimulation standards. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. p. 179–188. HSCC '15, Association for Computing Machinery, New York, NY, USA (2015)
9. Centomo, S., Deantoni, J., De Simone, R.: Using SystemC Cyber Models in an FMI Co-Simulation Environment. In: 19th Euromicro Conference on Digital System Design 31 August - 2 September 2016. 19th Euromicro Conference on Digital System Design, vol. 19. Limassol, Cyprus (Aug 2016). https://doi.org/10.1109/DSD.2016.86, https://hal.inria.fr/hal-01358702
10. Combemale, B., Deantoni, J., Baudry, B., France, R.B., Jézéquel, J., Gray, J.: Globalizing modeling languages. Computer **47**(6), 68–71 (June 2014). https://doi.org/10.1109/MC.2014.147
11. Cremona, F., Lohstroh, M., Broman, D., Di Natale, M., Lee, E.A., Tripakis, S.: Step Revision in Hybrid Co-simulation with FMI. In: 14th ACM-IEEE International Conference on Formal Methods and Models for System Design. IEEE, Kanpur, India (Nov 2016)
12. Deantoni, J., André, C., Gascon, R.: CCSL denotational semantics. Research Report RR-8628, Inria (Nov 2014), https://hal.inria.fr/hal-01082274
13. Garlan, D., Shaw, M.: An introduction to software architecture. Advances in software engineering and knowledge engineering **1**(3.4) (1993)
14. Lee, E.A.: Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). pp. 363–369 (2008)
15. Liboni, G., Deantoni, J.: Wip on a coordination language to automate the generation of co-simulations. In: 2019 Forum for Specification and Design Languages (FDL). pp. 1–4. IEEE (2019)
16. Liboni, G., Deantoni, J., Portaluri, A., Quaglia, D., De Simone, R.: 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), https://hal.inria.fr/hal-01675396
17. Medvidovic, N., Taylor, R.N.: A framework for classifying and comparing architecture description languages. ACM SIGSOFT Software Engineering Notes **22**(6), 60–76 (1997)
18. Modelisar: FMI for Model Exchange and Co-Simulation (July 2014), https://fmi-standard.org/downloads#version2
19. Mustafiz, S., Gomes, C., Vangheluwe, H., Barroca, B.: Modular design of hybrid languages by explicit modeling of semantic adaptation. In: 2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS). pp. 1–8 (April 2016). https://doi.org/10.23919/TMS.2016.7918835

20. Neema, H., Gohl, J., Lattmann, Z., Sztipanovits, J., Karsai, G., Neema, S., Bapty, T., Batteh, J., Tummescheit, H., Sureshkumar, C.: 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. pp. 235–245. Linköping University Electronic Press (2014)
21. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Advances in computers **46**, 329–400 (1998)
22. Savicks, V., Butler, M., Colley, J.: Co-simulating event-b and continuous models via fmi. In: Proceedings of the 2014 Summer Simulation Multiconference. p. 37. Society for Computer Simulation International (2014)
23. Schierz, T., Arnold, M., Clauß, C.: Co-simulation with communication step size control in an fmi compatible master algorithm. In: Proceedings of the 9th International MODELICA Conference; Munich; Germany. pp. 205–214. Linköping University Electronic Press (2012)
24. Tavella, J.P., Caujolle, M., Tan, C., Plessis, G., Schumann, M., Vialle, S., Dad, C., Cuccuru, A., Revol, S.: Toward an Hybrid Co-simulation with the FMI-CS Standard (Apr 2016), https://hal-centralesupelec.archives-ouvertes.fr/hal-01301183, research Report
25. Tavella, J.P., Caujolle, M., Vialle, S., Dad, C., Tan, C., Plessis, G., Schumann, M., Cuccuru, A., Revol, S.: 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). pp. 1–5. IEEE, Berlin, Germany (Sep 2016). https://doi.org/10.1109/ETFA.2016.7733616
26. Thule, C., Gomes, C., Deantoni, J., Larsen, P.G., Brauer, J., Vangheluwe, H.: Towards the Verification of Hybrid Co-simulation Algorithms. In: Workshop on Formal Co-Simulation of Cyber-Physical Systems (SEFM satellite). Toulouse, France (Jun 2018), https://hal.inria.fr/hal-01871531
27. Tripakis, S.: Bridging the semantic gap between heterogeneous modeling formalisms and fmi. In: 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). pp. 60–69. IEEE (2015)
28. Van Acker, B., Denil, J., Vangheluwe, H., De Meulenaere, P.: 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. pp. 205–212. DEVS '15, Society for Computer Simulation International, San Diego, CA, USA (2015)
29. Wang, B., Baras, J.S.: 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. pp. 33–40. DS-RT '13, IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/DS-RT.2013.12, http://dx.doi.org/10.1109/DS-RT.2013.12