Transaction-level Functional Mockup Units for Cyber-Physical Virtual Platforms

Stefano Centomo, Michele Lora, Franco Fummi
Department of Computer Science – University of Verona (Italy)
name.surname@univr.it

***Abstract*—The modelling process of Cyber-physical Systems aggregates semantics and languages tailored to different specific domains. The simulation of these complex systems involves different tools and their coupling requires computational effort. In the last few years both Academy and Industry worked toward the definition of standard interfaces able to overcome such issues. The Functional Mock-up Interface (FMI) standard emerged as one of the most promising tool to easily export and integrate heterogeneous models. However, the standard still shows some weaknesses, particularly when dealing with Functional Mock-up Units (FMUs) describing discrete-event systems.**

This paper explore the features of the standard to find its shortcomings when dealing with discrete models. Then, it proposes a systematic approach to fully exploit the features of the current standards to overcome such limitations. The solution is based on two concepts: (1) exposing the internal time of the FMUs, and (2) exploits the newly exposed information to implement temporal decoupling. The combination of these two concepts allows to optimize the FMUs coordination algorithms. It reduces the number synchronization points and move the simulation from cycle-accurate to transaction-accurate. The impact of these optimizations is measured on a set of benchmarks having different tread-offs of computation and control.

***Index Terms*—Co-simulation, Functional Mock-up Interface, Transaction-Level, Models of Computation, Automatic Abstraction**

I. Introduction

Cyber-Physical Systems (CPSs) are fundamental in many revolutionary application that are now shaping todays world. They are the technological ground for many different applications, such as smart manufacturing, smart city, autonomous driving, *etc.* Improving design techniques for CPSs is crucial to advance the state of the practice in many different system engineering sub-disciplines [1].

System design requires models, and often models need to be simulated in order to provide designers with the feedback necessary to evaluate the quality of their ideas [2]. However, the intrinsic heterogeneity of CPSs makes modeling and simulation a very difficult task [3]. Technologically, holistic simulation of heterogeneous system requires to use either complex co-simulation environment aggregating specialized simulators for the many design domains involved in the system, or to produce a single holistic model of the system [4]. However, the latter solution requires to access, often unavailable, open specifications for every component of the system. On the other hand, co-simulation requires to interface many different simulation tools, often providing incompatible interfaces.

In this scenario, the FMI standard for co-simulation emerged as one of the most promising technologies to interface heterogeneous simulators and models [5]. It defines an Application Programming Interface (API) that must be implemented by the simulator. As such, FMI can be easily used to build *Cyber-Physical Virtual Platforms* [6] able to emulate the behaviors of both the "cyber" and "physical" parts of a CPS.

Even though the FMI standard proved to be a powerful tool to build such Cyber-Physical Virtual Platform, its focus is still strongly oriented to the simulation of continuous dynamic systems [7]. Thus, simulation of digital components requires to adapt the use of its construct to replicate the semantics of HW components simulators [6]. Some previous work has been done so far to improve the support of Hardware Description Language (HDL) models in FMI [6], [8]. However, the advantages in terms of simulation speed of higher-level models, such as *Transaction-level models* [9], have not been exploited so far due to some limitations of the standard. This paper aim at analying and discussing such limitations. Then, it proposes a set of adjustments in the use of FMI constructs defined in the current standard for co-simulation (*i.e.*, version 2.0). Furthermore, it proposes a simulation coordination scheme that exploits such adjustments. These contributions together allows to produce and use *Transaction-level Functional FMUs*, thus speeding up simulation of Cyber-Physical Virtual Platforms.

In the last few months, the FMI Steering Committee announced a new interface (version 3.0) that aims to introduce the hybrid co-simulation concept [10], but actually is still in pre-alpha version and not yet fully accepted. Furthermore, it will take time to be accepted from all the tools that support the standard. Using the current version 2.0 guarantee the retro-compatibility with the actual version of the tools.

Figure 1 summarizes the contributions of this work. On the left, the CPS to be designed is simulated by using a *Cycle-accurate Cyber-Physical Virtual Platform*. The virtual platform is composed by using the FMI standard, and it is composed by both the models of the "cyber" and the "physical" sub-systems of the model. In this work we focus on the "cyber" part of the system. Here ,it is modeled by aggregating different FMUs representing the different digital components of the system. The simulation is managed by a classic *Master Algorithm* coordinating the FMUs. The time evolution of the virtual platform on the left-side of the figure is accurate with respect to the clock cycle of the system. Thus, each simulation step simulates a single clock cycle,

This work has been partially supported by FIDEL: a joint project between the University of Verona and the French Institute for Research in Computer Science and Automation (INRIA).

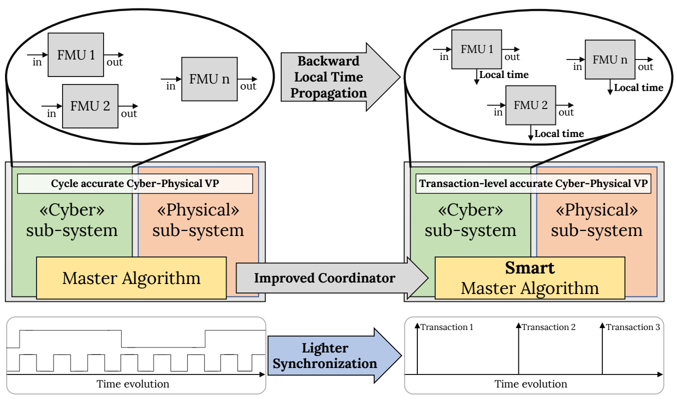


Figure 1: Overview of the contribution.

and synchronization must be performed after each step. This work improves the left side configuration by proposing two modifications to the platform and its components:

- The functionality within the **FMUs** composing the digital part of the system are abstracted to **transaction-level**. Their interfaces are modified to make them communicate their internal local time backward to the master algorithm.
- The **master algorithm** is improved to exploit the information about the local time of the FMUs in the model.

These modifications allow to produce the ***Transaction-level accurate Cyber-Physical Virtual Platform*** on the right-side of the figure. The platform synchronizes at each transaction defined by the communication protocol. Thus, it benefits the lighter synchronization for improving the simulation speed.

The paper is organized as follows: Section II gives the necessary background about FMI, and summarize the state of the art. Section III discuss the advantages and the limitations in the current version of the FMI standard and discuss a set of possible improvements. Section IV presents the methodology proposed by this paper. The presented approach is implemented by building an automatic tool-chain and then experimentally evaluated in Section V. Finally, in Section VI we draw some conclusions.

II. Background and Related Work

FMI is a tool-independent standard that aims at enhancing the interoperability between tools of different vendors in the field of systems design [5], [11]. It supports both model exchange and co-simulation of dynamic models produced by using different tools and languages. The standard has been originally developed by Daimler AG, and maintained by the MODELISAR Consortium. After the termination of the MODELISAR European Project, the standard is maintained by the Modelica association. The last version of the standard is the 2.0, accepted in 2014, while the version 3.0 is currently under review. The basic blocks of a FMI-based simulation environment are called FMUs. Multiple FMUs can be imported within a simulation tool to be executed. Each FMU may implement only one of the two variations of the standard: *Model Exchange* or *Co-Simulation*. A model exchange FMU describes functionalities by using differential, algebraic and discrete equations with time-, state- and step-events [11]. The equations must be solved by an external solver, that is thus

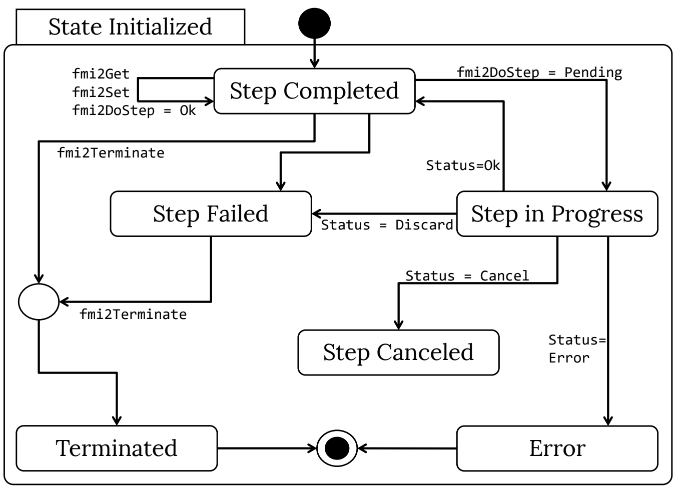


Figure 2: Statechart representation of the coordinator algorithm for a FMU.

required to simulate model exchange FMUs. A co-simulation FMU must model the functionality and implement the solver. As such, the described model can be simulated without using any external solver.

In our opinion, the standard for model exchange does not suit well for describing discrete-event models. Thus, in this work we focus on co-simulation, and we hereby describe the main features and structure of a FMU for co-simulation.

A. FMI Standard 2.0 for co-simulation

An FMU is composed by an XML file and the implementation of the functionality as a dynamic library written in C. Any FMU for co-simulation must implement also the solver necessary to execute the functionality. The XML file must specify all the variables of the FMU visible to the simulation environment [5]. Each variable has a *name*, *causality* (*e.g.*, input, output, parameter, *etc.*), a *type* and a *value reference*. The value reference of a variable must be unique among the variables of each type, since each variable will be uniquely identified by the pair made of its type and value reference.

The dynamic library must implement the functionality through a set of functions defined by the standard. The most important, among the many defined in the standard, are:

- `fmi2SetupExperiment`: initializes the internal variables of the FMU.
- `fmi2Set`: sets the value of an internal variable of the FMU *i.e.*, it assigns a value to an input.
- `fmi2Get`: gets the value of an internal variable of the FMU *i.e.*, it returns the value of an output.
- `fmi2DoStep`: advances the simulation time of the component executing the behavior defined by the model.

The standard defines the signature of all the functions to implement, but it imposes how they should be used, as it rather defines only some limitations on the possible combinations.

B. Simulation coordination in the FMI standard

Any model having one or more FMUs requires a coordination mechanism compliant with the FMI standard. Version 2.0

of the standard [5], [11] defines the concept of *master algorithm* as the actor in charge of exchanging data between FMUs and synchronizing the simulation of the involved solvers. The former task is implemented by the master algorithm by invoking the `fmi2Get` and `fmi2Set` functions of the co-simulation API. The latter task is implemented by carrying on the components execution by invoking the `fmi2DoStep` functions of the FMUs composing the model being simulated. While the standard defines some rules about the master algorithm, the exact definition of the algorithm is not part of the standard. The rules defined are mostly imposing some limitations on the structure.

Figure 2 reports a statechart simplified version of the master algorithm. It shows the functions that the algorithm must invoke for each FMU in the model according to standard 2.0. The figure reports only the execution of a initialized FMU, that is a system that already successfully went through the FMU setup state. Once the `fmi2SetupExperiment` function returned successfully, the execution reaches the *Step Completed* atomic state within the *State Initialized* sub-machine. The master algorithm may invoke the `fmi2Get` or the `fmi2Set` functions to respectively read or write values of the FMU external variables. Otherwise, the algorithm may invoke the `fmi2DoStep` function: the algorithm must call the function by passing as a parameter the amount of time that must be simulated. If so, the machine moves to the *Step in Progress* state. The FMU simulates by executing its functionality: if the step is not canceled or discarded, and no errors are catched during the FMU execution, the `fmi2DoStep` returns and the machine goes back to the *Step Completed* state, and the FMU advances its own local time according to the one previously passed as a parameter. These steps iterate until no `fmi2Terminate` function is invoked. A simulation tool may implement a coordinator for FMI by simply iterating this process for each FMU, or by implementing some more complex mechanism. Still, it must adhere to the state-chart in Figure 2. Finally, the standard explicitly states that is not legal to call a `fmi2Get` function after `fmi2Set` functions without calling the `fmi2DoStep` function in between.

Listing 1 shows a C implementation of a trivial master algorithm using the functions defined by the FMI standard for co-simulation. The procedure loads the FMUs instantiating two variable of type `fmi2Component` that will points to the FMU implementations (Lines 3–6). The status variable is declared (Line 8): every function defined in the standard returns a status. The master algorithm initializes the FMUs, the timing variables and defines four integer variables (Lines 10–15). Then, a thousand simulation cycles are executed: the algorithm reads the output from the FMUs and assigns it to the input variables (Lines 18–20). Then, it sets the input variables of the FMUs (Lines 21–22). Finally, the algorithm executes the functionalities, advancing the global time of the FMUs and update the global time (Lines 23–25).

C. Related Work

The weaknesses of the FMI standard have been first identified in [12].They are related to managing hybrid systems, having both discrete and continuous dynamics. The analysis

Listing 1: Sketch of the C implementation of a basic master algorithm compliant with the FMI standard. The algorithm executes a thousands iterations, each of those advances the local and global time of 10 time units.

```c
 1 ...
 2 int main(int ac, char * av[]){
 3   fmi2Component component_1 =
 4     load_fmu("./component_1.fmu");
 5   fmi2Component component_2 =
 6     load_fmu("./component_2.fmu");
 7   ...
 8   fmi2Status st;
 9   ...
10   st = fmi2SetupExperiment(component_1);
11   st = fmi2SetupExperiment(component_2);
12   ...
13   time = 0; step = 10;
14   ...
15   fmi2Integer in_1, in_2, out_1, out_2;
16   // Simulation starts here.
17   for(int i = 0; i < 1000; ++i) {
18     st = fmi2GetInteger(component_1, 0, &out_1);
19     st = fmi2GetInteger(component_2, 0, &out_2);
20     in_1 = out_2; in_2 = out_1;
21     st = fmi2SetInteger(component_1, 1, in_1);
22     st = fmi2SetInteger(component_2, 1, in_2);
23     st = fmi2DoStep(component_1, time, step);
24     st = fmi2DoStep(component_2, time, step);
25     time = time + step;
26   }
27 }
28 ...
```

highlights that FMI is more suited for physical, continuous-time (or discretized) systems, rather than discrete-event systems. Thus, it is tricky to use FMI when models require discrete events.

Some work has been done to close the semantic gap between continuous-time models, and discrete-event (or even clock accurate) models in FMI. The solution proposed in [7] relies on tokens synchronizing the FMUs in the model when discrete-events happen. However, such a mechanism may introduce many synchronization points in the execution thus slowing down the simulation. The work in [6] showed how the token-based simulation does not suit well to simulate models coming from HDL descriptions. Then, it proposed an ad-hoc synchronization methodology to reproduce the cycle-accurate behavior of HDL descriptions. It manages the synchronization locally to each FMU, while the data are exchanged by an additional FMU acting as a communication hub for the data in the system. It relies on automatic code generation to generate the FMUs implementing such mechanism. Automatic code generation of FMUs for co-simulation from HDL descriptions has been presented in [8]: it relies on a state-of-the-art abstraction technique [13] to translate HDL models into C descriptions. Then, the C implementation of the functionality is wrapped by an interface using the FMI API for co-simulation.

While none of the approaches described above is proposing modifications to the standard, a number of papers do it. [10] proposes an additional mechanism to add to the FMI standard, aside to the model exchange and the co-simulation mechanisms. This is called *Hybrid Co-simulation*, and it is specifically tailored to manage mixed continuous/discrete models, having discrete-events impacting on continuous dynamics. [14] proposes some modifications to the API specified by the FMI standard for co-simulation. In particular, it proposes to add some variants of the `fmi2DoStep` function able to manage discrete-event and to interrupt (and preempt) the execution of an FMU when events must be managed.

To the best of our knowledge, **none of the previous work proposed *transaction-level FMU***. This is due to the fact that the master algorithm must always know in advance the next step size for each FMU [12], [15]. Our solution overcomes this limitation, allowing to produce more abstract FMUs.

III. FMI Standard Advantages and Limitations

As a first contribution of this paper, we discuss the standard's features useful to create cyber-physical virtual platform. Then we will discuss some limitations that make integration of virtual platforms difficult. Our discussion will be from a "cyber" point of view, as we aim at highlighting the weaknesses of the standard when dealing with discrete-event and cycle-accurate components.

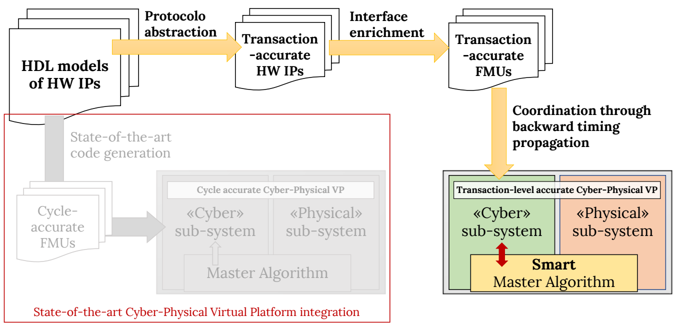
Indeed the standard allows to ease the integration of different tools. It simplifies the interfacing of heterogeneous description. It allows the designer to care only marginally about communication and synchronization between simulators. Furthermore, it is reasonable to assume that complex CPSs are designed by multiple teams of designers. For instance, a team might be in charge of the physical part while the other designs the computational infrastructure. The FMI standard allows to easily integrate the models produced by different teams, to build a holistic simulation of the system.

However, as hinted in Section II-C, the standard has been strongly oriented to continuous systems and dynamics. We can identify different drawbacks when modeling discrete components, and in particular when simulating digital components.

The set of *data types* provided by the standard is limited. When modeling digital HW it happens to use multi-valued logic values, or signals that uses an arbitrary number of bits. Meanwhile, FMI allows only integer, real, string and boolean. Thus, HDL data-types must be mapped on the provided types. Different mappings have been already proposed in the past. Multi-valued logics as well as arbitrary long bit vectors have been mapped onto strings [16], and (more efficiently) abstracted to unsigned integer [8]. Still, none of the previous mapping is ideal even though they partially solve the problem.

The data-types provided by FMI are even more insufficient when modeling digital HW models at higher levels of abstraction, or when modeling SW. In such case, models may require aggregate data types, *e.g.*, to represent sockets' payloads in transaction-level description, or classes of SW models. In this case, FMI does not provide any other solution than breaking down any aggregated type into its basic components.

The standard does not provide any mechanism to *specify the Model of Computation* employed by the FMU to implement the functionality. In the case of a digital HW description assignments are concurrent. However, simulators usually rely on sequential models of computation (*e.g.*, data-flows). When aggregating digital HW components using FMI, complex synchronization structures must be built [6], [7] to guarantee the functional equivalence of the aggregated model of the system.



HDL models of HW IPs
Protocolo abstraction
Transaction -accurate HW IPs
Interface enrichment
Transaction -accurate FMUs
Coordination through backward timing propagation
State-of-the-art code generation
Cycle-accurate FMUs
Cycle accurate Cyber-Physical VP
«Cyber» sub-system
«Physical» sub-system
Master Algorithm
State-of-the-art Cyber-Physical Virtual Platform integration
Transaction-level accurate Cyber-Physical VP
«Cyber» sub-system
«Physical» sub-system
Smart Master Algorithm

Figure 3: Overview of the proposed approach, and comparison with the state-of-the-art methodology presented in [8].

It is not possible to retrieve the *internal time of an FMU*. The master algorithm "imposes" to each FMU its internal timing. The main issue is related to the `fmi2DoStep` function behavior: it is called by the master algorithm and it carries on the simulation time while executing an FMU functionality. The execution of an FMU cannot be preempted by external events. Neither the FMU is allowed to simulate an amount of time different with respect to the one imposed by the master algorithm, since the FMU cannot communicate back to the master algorithm its effective internal timing. For this reason, the master algorithm must alway be able to know exactly the length of the next time step of each FMU. This forces the master algorithm to call the `fmi2DoStep` function of an FMI using the shortest time step available, or to perform multiple step revisions. Thus, this limitation leads to an higher number of synchronization points in the simulation and makes impossible to use advanced synchronization techniques, such as *temporal decoupling*. Thus, it is not well suited to manage discrete events that might be generated by system's components. In the case of HW description, this usually forces to simulate each FMU with a time granularity equal to the clock cycle [6].

A. Novelty

To summarize, the paper aims at improving the previously published solutions found in the literature in two directions:

- While previous solutions propose modifications to the standard, this paper proposes a solution that modifies the characteristics of FMUs while still being **compliant with the current version of the Standard** (*i.e.*, version 2.0).
- Previous solutions rely on a very thin grained synchronization mechanism. This solution is able to produce **modules requiring coarser synchronization**. That is, in the case of a HW Intellectual Property (IP) component, the FMU implementing its functionality can perform synchronization at each transaction of its communication protocol. In other words, this paper aims at moving the simulation granularity from RTL to TLM.

Finally, an automatic tool-chain has been built to evaluate our methodology on a set of HDL IPs.

IV. Methodology

Figure 3 gives an overview of the proposed methodology. It starts from a set of HDL IPs models described by using

Listing 2: `modelDescription.xml` file of the `component_1` with time port.

```
 1 ...
 2 <ModelVariables>
 3
 4 <!-- Input Ports -->
 5   <ScalarVariable name="in_1"
 6                   causality="input"
 7                   valueReference="0">
 8                   <Boolean start="false"/>
 9   </ScalarVariable>
10
11   <ScalarVariable name="in_2"
12                   causality="input"
13                   valueReference="1">
14                   <Boolean start="false"/>
15   </ScalarVariable>
16
17 <!-- Output Ports -->
18   <ScalarVariable name="fmi2TLifaceTime"
19                   causality="output"
20                   valueReference="-1">
21                   <Integer start="0"/>
22   </ScalarVariable>
23
24   <ScalarVariable name="out_1"
25                   causality="output"
26                   valueReference="0">
27                   <Integer start="0"/>
28   </ScalarVariable>
29
30   <ScalarVariable name="out_2"
31                   causality="output"
32                   vr="1">
33                   <Integer start="0"/>
34   </ScalarVariable>
35
36 </ModelVariables>
37 ...
```

either VHDL or Verilog. While in the previous approach [8] (*i.e.*, red box in Figure 3) such models were simply translated into C models then wrapped into FMUs, in this paper HDL IPs undergoes an abstraction and manipulation process (colored arrows in Figure 3). The produced models rely on a transaction-level synchronization mechanism. Finally, these FMUs are inserted within the Cyber-Physical Virtual Platform. They will be coordinated by a master algorithm that is aware of the shifting in synchronization and communication granularity achieved by applying the transformations.

A. FMUs generation and timing backward propagation

The most constraining issue identified in Section III is the impossibility for the FMUs to propagate their local time back to the coordinator. In our analysis, it is the main problem to to tackle for achieving an efficient discrete-event simulation. In fact it allows the master algorithm to decide efficiently the length of the next simulation step.

This problem can be solved by allowing each FMU to simulate in a decoupled way, without defining the simulation step size. When the Master Algorithm calls the `fmi2DoStep`, the FMU simulates until it does not need to synchronize or communicate with other components in the system.

The proposed methodology starts by generating Transaction-Level models starting from HW descriptions. We apply the methodology defined in [17] that starts from HW models described at Register Transfer Level (RTL) and abstract them to Transactional Level Modeling (TLM): it takes in input an HDL model and its communication protocol. The HW descriptions can be provided by using the most common HDLs (*i.e.*, VHDL or Verilog). The protocol of a component can be specified in different ways. The state-of-the-art implementations of the RTL-to-TLM abstraction methodology relies on ad-hoc protocol specification languages [13]. The abstraction result is a C++ class representing a Transaction-Level model. Each transaction of the system is executed by invoking its `simulate` function, and it emulates one transaction of the specified protocol. The internal time of the model is annotated as `Integer` datatype, which represents the number of clock-cycles executed in the last transaction. The abstraction procedure computes the number of clock cycles for each transaction, and annotate it within the generated model.

The interface of the model is isolated in a structure embedded inside the C++ class. The structure contains a set of fields representing the original ports of the HW models. The data-types of these fields are abstracted into C native data-types. For instance, a 32-bit `logic_vector` datatype is abstracted into `uint32_t` C datatype. The methodology relies on automatic abstraction of HDL data-type [13] to perform this transformation. Furthermore, the interface structure also contains the time annotation of the model.

Then, the methodology goes on by wrapping the C++ class within the FMI functions. Thus, it generates the set of `fmi2Set` and `fmi2Get` necessary to write and read, respectively, input and output variables from and to the components. It also generates the `fmi2DoStep` function that calls the generated `simulate` function emulating a component transaction. The `fmi2DoStep` function still accepts the step length, in order to stay compliant with the standard. However, it ignores it as the actual internal time of the FMU at the end of the execution is computed by the `simulate` function. Then, the methodology continues by generating the XML file of the FMU. The original ports of the HW model are mapped in the FMI data-types. In particular, the `Boolean` and `Integer` FMI types are used to represent respectively single bit (or logic) and bit (or logic) vectors. Then, the *value reference*, is assigned to each port. The assignment of the value reference starts from 0 for each data-type. Listing 2 depicts the definitions of the ports in the XML file for a component originally having two input and two output ports.

Then, the methodology enriches the interface of the FMU with the internal time annotation of the Transaction-Level Model. The internal time of the Transaction-level model is exposed as a new `Integer` port of the FMUs (see Listing 2, line 18-22).The value reference `-1` is reserved for the timing port, since ports are uniquely identified by their type and value reference pairs. This assures that it can be uniquely identified once the FMU is loaded by a simulator. Furthermore, the timing port is called `fmi2TLifaceTime` in order to decrease the chances of name clashing with the other ports of the FMU. This last solution is helpful to increase also the readability of the produced FMUs.

B. A better coordinator for discrete systems

Listing 1 describes an example of a trivial Master Algorithm. It is able to executes cycle-accurate FMUs and it must synchronize the components composing the system at each

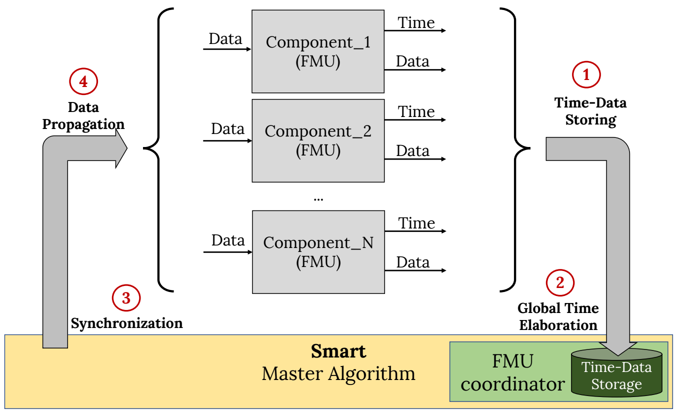


Figure 4: Scheme of the *Smart Master Algorithm* with the *FMU Coordinator* of Transaction-Level FMUs.

clock cycle. For this reason, in the trivial master algorithm the time step of each `fmi2doStep` is set to be equal to the clock period of the system being modeled. Such a solution is indeed precise; however, it uses an unnecessarily high number of synchronization points. The backward propagation of the FMUs internal time can now be exploited to to reduce the number of synchronization points.

Figure 4 shows the execution scheme of the *Smart Master Algorithm*. Its core is the *FMU Coordinator*: it is in charge of storing the internal time values of the FMUs in the system, and it decides at each simulation step which components must be executed. Initially, the *Smart Master Algorithm* simulates all the FMUs, without defining a step size. All the FMUs return to the coordinator their internal time after their first execution. Then, the *Smart Master Algorithm* iterates the following steps (as in Figure 4).

- ① *Time-Data Storing*: the internal time and the new data of each FMUs are retrieved from the master algorithm and passed to the *FMU Coordinator* that stores them.
- ② *Global Time Elaboration*: the *FMU Coordinator* elaborates the new Global Time of the simulation as the minimum value among all the internal times of the FMUs.
- ③ *Synchronization*: any FMU having the internal time equal to the Global Time is inserted into the list of *runnable FMUs*. Data read after the last execution of each runnable FMU, and previously stored by the coordinator, are shared with the system (*i.e.*, the values become valid for the entire system).
- ④ *Data Propagation*: the *Smart Master Algorithm* propagates the data and simulates the FMUs present in the list of runnable FMUs.

Listing 3 shows a sketch of the novel Smart Master Algorithm proposed. It reports only the most important parts of a possible C++ implementation of the coordination mechanism. Initially (Lines 2–9) there is the declaration of a status variable, an integer variable tracking the global time, and an array of components. The FMUs composing the system are stored in the array after being loaded. It is also declared an array to store the local times of the FMUs. Notice that the same position in the two arrays refers always to the same FMU.

Listing 3: Sketch of the C++ implementation of the *Smart Master Algorithm* exploiting backward timing propagation.

```
 1 ...
 2 fmi2Status st;
 3 unsigned int global_time=0;
 4
 5 fmi2Component components[num];
 6 components[0]=load_fmu("./component_1.fmu");
 7 components[1]=load_fmu("./component_2.fmu");
 8
 9 unsigned int local_time_vector[num];
10
11 for(int i=0; i < num; i++) {
12  st=fmi2DoStep(components[i], global_time, 0);
13  st=fmi2GetInteger(component[i], -1, &local_time);
14  local_time_vector[i]=local_time;
15  retrieve_and_store_output(component[i]);
16 }
17
18 while(global_time < 1000) {
19  set< fmi2Component > runnable_FMUs;
20  global_time=find_minimum(local_time_vector[0]);
21
22  for(int i=0; i < num; i++) {
23   if(local_time_vector[i] == global_time)
24    runnable_FMUs.insert(components[i]);
25  }
26
27  propagate_data(runnable_FMUs);
28  set< fmi2Component >::iterator it;
29  for(it=runnable_FMUs.begin();
30    it != runnable_FMUs.end; it++)
31  {
32   fmi2Component * component=*it;
33   st=writeInputs(component);
34   st=fmi2DoStep(component, global_time, 0);
35   st=fmi2GetInteger(component[i], -1, &local_time);
36   local_time_vector[i]=local_time;
37   retrieve_and_store_output(component[i]);
38  }
39 }
40 ...
```

Then, the coordinator initializes the simulation (Lines 11–16) by executing all the components once without advancing the global time. For each execution, the local time is retrieved (Lines 13) and stored (Line 14). Then, all the output values written by the FMU are retrieved and stored (Line 15). Then, the system is simulated (Lines 18–39). At each simulation cycle a set containing the runnable FMUs is created empty, and populated after the the global time has been update (Lines 19–25). Then, data previously produced by the runnable FMUs are propagated (Line 27). Finally, each runnable FMU is executed (Lines 28–37).

V. Methodology Application

The methodology has been implemented by assembling a tool-chain able to perform the different abstraction, manipulation and translation steps defined. We used the API provided by the HIFSuite framework [18] to extended the tool-chain presented in [8]. The automatic abstraction of HDL descriptions is performed by specifying the components' protocols to generate the corresponding transaction-level C++ descriptions as defined in [17]. The models produced by the abstraction are enriched with the timing backward propagation mechanism. Finally, a tool wraps the model within the FMI

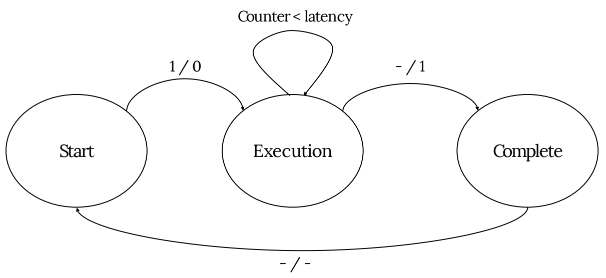


Figure 5: FSM implemented in the HW model with generic latency.

Table I: Execution time of FMUs simulation using trivial Master Algorithm, with different number of iterations.

# iterations (clock cycles)	Execution of FMUs (seconds)				
	2	5	10	20	40
100 K	4.76	10.75	21.89	43.34	82.46
1 M	41.87	104.13	198.74	405.25	834.34
10 M	421.93	1021.64	2015.55	4129.17	8322.22
20 M	886.78	2062.32	4267.29	8219.65	16466.54

APIs for co-simulation. We applied the tool-chain to a set of benchmarks characterized by two different dimensions: the protocol latency and the number of FMUs. In this way, we aim at estimating the scalability of the proposed approach with respect to these two dimensions. We implemented the same functionality within each HW component of the system, since the paper focuses on the interfaces of the components, rather than on their internal functionalities. The internal functionality is kept extremely simple: as such, the communication and synchronization overhead is predominant in the simulation. Each component implements a simple counter, that counts until its pre-defined latency is reached. Figure 5 shows the Finite State Machine (FSM) implemented within the HW models used in our experimental setup. Whenever a HW model reaches the *Execution* state it remains in the state for a certain amount of clock cycles. The number of clock cycles represents the latency of the model. For each experiment, we have considered components with different latencies. In the experiments we refer to the *base latency* of an experiment as the minimum latency of the component in that experiment.

We generate two FMUs of different types for the same HW model: the cycle-accurate FMU and the transaction-level FMU with backward timing propagation. All the experiments have been performed on a 64-bit machine running Ubuntu Linux 16.04, equipped with 16 GB of memory and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

Table I reports the execution time by using the Trivial Master Algorithm, with different cycle-accurate FMUs and different numbers of iterations. The protocol latency dimension is not considered in this Table because the Trivial Master Algorithm simulates only cycle-accurate FMUs. Using the Trivial Master Algorithm the protocol latency does not affect the coordination overhead in the simulation. The results show that moving in both the dimensions (number of FMUs or iterations) the execution time increases almost linearly.

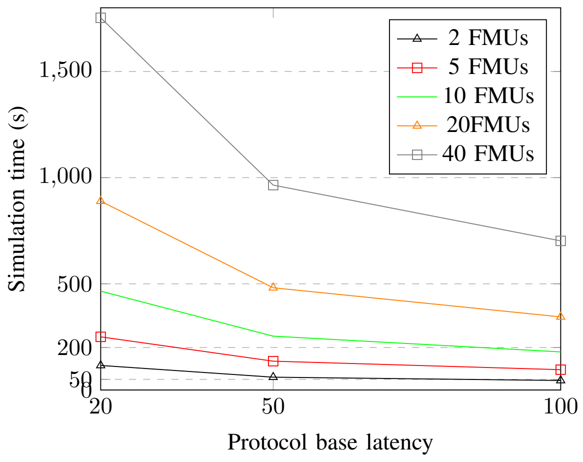


Figure 6: Trend of the simulation overhead using the Smart Master Algorithm with respect to the protocol latency.

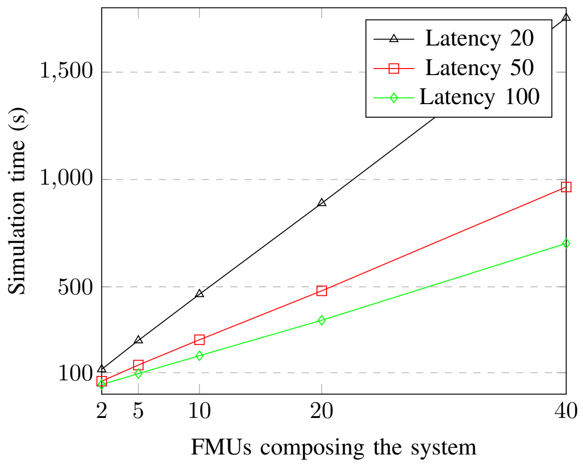


Figure 7: Scalability of the Smart Master Algorithm with respect to the number of FMUs.

Table II compares the simulation speed achievable by using the Trivial Master Algorithm and the Smart Master Algorithm. The results regarding the experiments with the Smart Master Algorithm depend on the protocol latency. The Trivial Master Algorithm results are not depending on this dimension. The comparison shows how the simulation time required when using the Smart Master Algorithm is in relationship with the protocol latency of the FMUs. The Smart Master algorithm with the transaction-level FMUs achieves up to 11x speed-up when using the largest protocol latencies considered. Reducing the protocol latencies of the transaction-level FMUs, the Smart Master Algorithm reduces the simulation performace, because of the increasing number of synchronization points. Of course, when the protocol latency is equal to one clock cycle (*e.g.*, when modeling combinatorial circuits) we have a degenerate case: the transaction-level and the cycle-accurate implementations will have the same amount of synchronization points. As such, only in that case, the Smarter Master Algorithm is slightly outperformed by the trivial one, due to the higher amount of computation required by the coordinator.

Figures 6 and 7 give a graphical representation of how the

Table II: Execution Time Comparison of Normal Master Algorithm and Smart Master Algorithm with different protocol latencies. In all the scenarios, 10 million clock cycles of the system have been simulated.

Base Latency (clock cycles)	Execution of FMUs (seconds)														
	2			5			10			20			40		
	Trivial	Smart	Speed-up	Trivial	Smart	Speed-up	Trivial	Smart	Speed-up	Trivial	Smart	Speed-up	Trivial	Smart	Speed-up
20	421.93	115.54	3.65x	1021.64	250.23	4.08x	2015.55	465.48	4.33x	4129.17	889.39	4.64x	8322.22	1752.71	4.75x
50	421.93	60.28	7x	1021.64	135.65	7.53x	2015.55	253.43	7.95x	4129.17	481.36	8.58x	8322.22	964.92	8.62x
100	421.93	44.71	9.44x	1021.64	95.57	10.69x	2015.55	179.34	11.24x	4129.17	344.25	11.99x	8322.22	702.17	11.85x

simulation overhead changes when changing the protocol base latencies and the number of FMUs respectively. The vertical axes of both table reports the simulation time, while the horizontal axes reports the two considered dimensions. The trends in Figure 6 show how performance improve by incrementing the latency. This is due to the fact that a longer latency allows for more temporal decoupling, thus less synchronization and communication overhead. Figure 7 shows that the simulation time increases linearly with the number of involved FMUs. Thus, it shows the minimal impact of the more sophisticated master algorithm proposed in this paper.

VI. Concluding remarks

In this work we showed a way to exploit the current version of the FMI standard to simulate FMUs representing digital components at transaction-level. We added some information to the FMUs and we adopted an ad-hoc master algorithm but still conform to the standard.

The experimental results showed that the proposed solution is beneficial for the simulation. However, our approach suffers the effort necessary to explicitly force the standard to accept the transaction-level FMUs we defined. After this work, we are convinced that improvements of the standard will ease simulation of discrete-event models.

Most importantly, this work highlights the benefits provided by the possibility to propagate timing information in the backward path going from FMUs to the master algorithm. Such a feature allows to label each output value got from an FMU with the internal time of the FMU. The time in the label will indicate the moment in the simulated time when the variable obtain that particular value. Of course, multiple values may exist for a single output variables. However, each of these values will have a different label indicating different simulation time-step since FMUs can only move forward in time. Furthermore, at each invocation of the `fmi2DoStep` function an FMU advances of at least an amount of time equal to the time granularity of the simulation that is strictly greater than zero. As a consequence, for each variable there will be only one value for each simulation step. Thus, the solution preserves determinism.

Such a solution recalls the *discrete-event* semantics typical of digital system. Thus, it will allow to implement a simulation mechanism more similar to the one used to simulate HDL models, as shown in the paper. Consequently, it might drastically improve the capabilities of the standard of simulating digital devices along with continuous-time models. Thus, it will drastically ease the integration of efficient Cyber-Physical Virtual Platforms.

References

[1] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-Physical Systems: The Next Computing Revolution," in *Proc. of the 47th Design Automation Conference*. ACM, 2010, pp. 731–736.

[2] S. W. Golomb, "Mathematical models: Uses and limitations," *IEEE Transactions on Reliability*, vol. 20, no. 3, pp. 130–131, 1971.

[3] E. A. Lee, "Fundamental Limits of Cyber-Physical Systems Modeling," *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 1, p. 3, 2017.

[4] F. Fummi, M. Lora, F. Stefanni, D. Trachanis, J. Vanhese, and S. Vinco, "Moving from Co-Simulation to Simulation for Effective Smart Systems Design," in *Proc. of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 286.

[5] T. Blochwitz *et al.*, "Functional Mockup Interface 2.0: The standard for tool independent exchange of simulation models," in *Proc. of MODELICA Conference 2012*, 2012, pp. 173–184.

[6] M. Lora, S. Centomo, D. Quaglia, and F. Fummi, "Automatic Integration of Cycle-accurate Descriptions with Continuous-time Models for Cyber-Physical Virtual Platforms," in *Proc. of ACM/IEEE Design Automation & Testing in Europe 2018*, pp. 1–6.

[7] S. Tripakis, "Bridging the semantic gap between heterogeneous modeling formalisms and FMI," in *Proc. of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2015, pp. 60–69.

[8] S. Centomo, M. Lora, A. Portaluri, F. Stefanni, and F. Fummi, "Automatic Generation of Cycle-Accurate Simulink Blocks from HDL IPs," in *Proc. of ECSI/IEEE Forum on Specification & Design Languages 2017 (FDL 17)*, 2017, pp. 1–8.

[9] L. Cai and D. Gajski, "Transaction Level Modeling: an Overview," in *Proc. of the 1st IEEE/ACM/IFIP CODES-ISSS*. ACM, 2003, pp. 19–24.

[10] F. Cremona, M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, "Hybrid co-simulation: it's about time," *Software & Systems Modeling*, nov 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0633-6

[11] MODELISAR Consortium, Modelica Association *et al.*, "Functional Mock-up Interface for Model Exchange and Co-Simulation – Version 2.0," *Available from https://www.fmi-standard.org*.

[12] D. Broman, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Requirements for Hybrid Cosimulation Standards," in *Proc. of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 2015, pp. 179–188.

[13] S. Vinco, V. Guarnieri, and F. Fummi, "Code Manipulation for Virtual Platform Integration," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2694–2708, 2016.

[14] 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 *Proc. of Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2018.

[15] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate Composition of FMUs for Co-Simulation," in *Proc. of the Eleventh ACM International Conference on Embedded Software*, 2013, p. 2.

[16] S. Centomo, J. Deantoni, and R. De Simone, "Using SystemC Cyber Models in an FMI Co-Simulation Environment: Results and Proposed FMI Enhancements," in *Proc. of Euromicro Conference on Digital System Design (DSD)*. IEEE, 2016, pp. 318–325.

[17] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.

[18] N. Bombieri, G. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, "HIFSuite: tools for HDL code conversion and manipulation," *EURASIP Journal on Embedded Systems*, vol. 2010, no. 1, pp. 1–20, 2010.