Automatic Integration of Cycle-accurate Descriptions with Continuous-time Models for Cyber-Physical Virtual Platforms

Michele Lora, Stefano Centomo, Davide Quaglia, Franco Fummi
Department of Computer Science, University of Verona
name.surname@univr.it

***Abstract*—Development of cyber-physical systems' control algorithms usually relies on architecture-agnostic abstract models, often leading to ineffective implementations. This paper presents a technique to automatically integrate cycle-accurate models of digital HW components with continuous-time physical models. It proposes a solution to the semantic gap between the involved models of computation. Furthermore, model generation and integration for both Simulink-based proprietary environment and FMI-based portable standard are presented.**

The aim of such techniques is to produce cyber-physical virtual-platforms: a powerful tool to refine control algorithms up to their SW implementations on the actual HW platform.

I. Introduction

Virtual platforms are powerful tools to co-design HW/SW devices [1] as they allow to execute SW while considering constraints imposed by the architecture [2]. In the case of Cyber-Physical Systems (CPSs), digital HW/SW is used to control physical processes. Thus, virtual platforms for CPSs must be able to capture evolution of both continuous (*i.e.*, physical processes) and discrete (*i.e.*, HW/SW devices) parts of the system. Unfortunately, virtual platforms for CPSs are not yet available and control SW designers still prefer to rely on multi-domain dynamic system simulators, such as Simulink, that allow them to design control strategies without considering architectural details. Of course, a given architecture may introduce timing artifacts that make control strategy ineffective, thus forcing re-implementation [3]. Cyber-physical virtual platforms would be thus beneficial to reduce this risk if they allow simulating together the timing-accurate behavior of HW/SW devices and the continuous dynamics of physical components of the CPS being simulated. Such a simulation environment may be obtained by integrating models of HW components within dynamic system simulators. It is thus necessary to overcome the semantic gap between cycle-accurate HW models and data-flow continuous-time models [4].

This work assumes that cycle-accurate HW models are originally described as Register Transfer Level (RTL) Intellectual Properties (IPs) using Hardware Description Language (HDL). Therefore it faces the problem of mapping the HDL constructs onto the target simulation environment primitives. Then, it defines two different synchronization techniques to reconcile the concurrent semantics of HDLs with the sequential execution of data-flows.

As depicted in Figure 1, the approach starts from a set of discrete-event HDL models of HW IPs, and a data-flow continuous-time model of the physical processes (*i.e.*, plant and environment). Each discrete-event HW model undergoes

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).

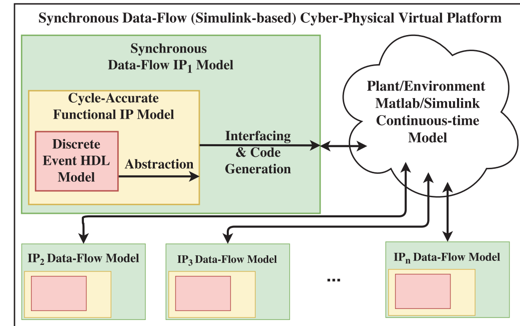


Figure 1: Structure of the target simulation environment.

a state-of-the-art abstraction process [5], [6], [7] to produce its cycle-accurate C++ representation. Then, the novel synchronization and code generation techniques are used to integrate cycle-accurate models to data-flow models. We adopt Simulink as target simulation environment, being the most popular simulator among system engineers. However alternative code-generation steps are presented to provide either Mathworks' C MEX S-function implementations, or portable Functional Mock-up Units (FMUs) compliant with the Functional Mock-up Interface (FMI) standard [8]. The approach is completely automatic: this and the other advantages will be shown in the experimental section of this paper.

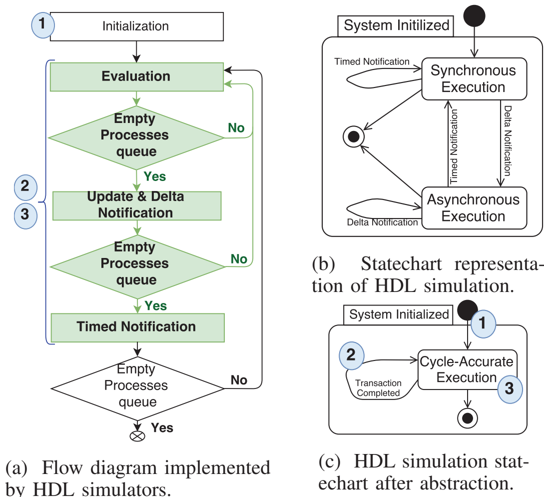
II. Background

A. Related work

As of today Virtual platforms focus on digital HW/SW systems [9] and support of continuous-time models is limited to analog devices [10], [11], and Micro-Electro Mechanical Systems [12]. CPSs SW development is still strongly based on top-down methodologies [13], and Model-based design tools such as Mathworks' Simulink [14], rely on abstract models thus imposing HW-in-the-loop techniques to perform precise timing estimation of the computational unit. Naderlinger [15] proposes *xTask blocks*, *i.e.* Simulink S-Functions representing time-annotated SW tasks: the approach is promising but it is still an approximation based on manual timing estimations. Alternatively, virtual platforms development can rely on Analog-Mixed Signals HDLs: [16] proposes Verilog-AMS and VHDL-AMS as modeling languages for heterogeneous systems. However, system and control engineers are reluctant to rewrite their models.

Co-simulation is widely used to emulate heterogeneous systems [17]. Standardized interfaces, *i.e.* FMI, allow to build complex simulation environments. Integration of SystemC

978-3-9819263-0-9/DATE18/©2018 EDAA



(a) Flow diagram implemented by HDL simulators.

(b) Statechart representation of HDL simulation.

(c) HDL simulation statechart after abstraction.

Figure 2: Execution schemes involved in the RTL to cycle-accurate abstraction and translation methodology.

models using FMI has been explored [18]: however, SystemC is a modeling rather than a design language. [4] formalizes the FMI primitives and proposes a synchronization mechanism to enable co-simulation of discrete-event models and data-flow descriptions. However, it is thought to be used in top-down design flows, without reusing IPs. The automatic generation of cycle-accurate FMI and Simulink blocks from HW models has been proposed by us in [19]. However, we did not address the problem of composing multiple blocks preserving functional equivalence, that is a major target of this paper.

B. *Code Generation for Virtual Platform Integration*

Tools as Carbon Model Studio [6], Verilator [20], HIFSuite [7] and methodologies have been proposed to automatically translate HDL IPs into cycle-accurate C++ models.

The abstraction methodology presented in [5] is used as a starting point for this work: it produces a C++ model that is cycle-accurate with respect to a starting HDL description. Its interface is a function executing a simulation cycle at each call. Inputs and Outputs are managed by a payload structure passed to the function as a parameter. Figure 2 depicts the different computation schemes involved in this abstraction. Figure 2a is the flow chart representing the HDL scheduler. Green boxes highlight the steps executed at each clock cycle. Figure 2b is the state chart implementing the scheduler in Figure 2a. Figure 2c shows the state chart obtained by applying the abstraction methodology to the one in Figure 2b: it has a unique state and a self-transition executing the set of green boxes in Figure 2a at each simulation function call. Circled numbers of Figures 2 and 3 will be used in the following of the paper to highlight the relations between execution schemes.

C. *Interface technologies for system simulation*

This work exploits two standardized interfaces to import and connect heterogeneous models: the FMI standard 2.0 [8], and Mathworks' proprietary C MEX S-Functions [14]. Both provide C-based interfaces to model components functionality.

FMUs [8] are the basic blocks of a simulation environment based on the FMI standard. Each FMU represents a component and it contains all the information necessary to be simulated. It may implement either the *Model Exchange* or the *Co-Simulation* version of the FMI standard: this work focuses only on Co-Simulation version, as it is more suitable for discrete-time models. An FMU consists of an XML file and a C-based implementation. The XML file describes the variables exposed by the FMU. The functional implementation of an FMU is based on the FMI standard C interface defining a set of functions implementing system behavior in terms of input-reading, output-writing and execution operations. The standard does not define how functions are scheduled at each simulation step: it rather specifies some constraints. As such, the same set of FMUs may act differently [21] on different simulators. The simulation evolves according to the state chart in Figure 3. Circled numbers, red thicker lines and bold text in Figure 3 will be used in the following Sections of this paper.

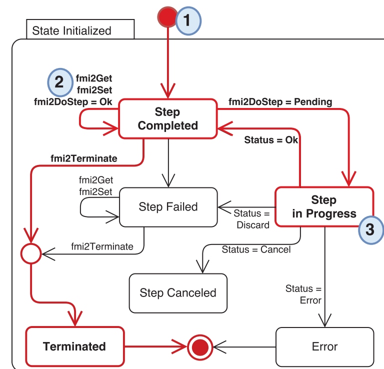


Figure 3: Statechart representation of the calling sequence of Co-simulation C functions to simulate an initialized FMU.

S-Functions rely on similar concepts, although they provides a simpler interface. A configuration file specifies the functions in charge of managing the simulation. The simulator executes the *initialization function* when the model is instantiated. Then, at each simulation step, it executes its *output function* in which the designer specifies the order of the input-reading, execution, and output-writing operations.

III. INTEGRATION METHODOLOGY

To integrate HDL IPs into dynamic system simulators we first introduce a mapping between the starting HDLs and the target interfacing technologies. This is necessary to create the modules to integrate. Then, we define two synchronization approaches providing alternative features.

A. *Mapping HDL primitives to FMI and S-Functions*

Figure 2 summarizes the two different execution schemes involved in the RTL to cycle-accurate abstraction for digital IPs. Figure 3 shows the execution scheme imposed by the FMI standard. S-functions rely on a similar schema and differs only on primitives naming. Circled numbers in figures label relations among schemes: ① labels the initialization phases, ② input and output phases and ③ the execution phase. Labels in Figure 3 show that the *Step Completed* and *Step In Progress* states are necessary to reproduce HDL semantic.

Table I reports the FMI and S-Function primitives used to map HDL events. That is how correspondences highlighted by circled numbers in Figures 2 and 3 are implemented.

Table I: Mapping of HDLs simulation events onto the primitives offered by the FMI standard and S-Functions.

HDL simulation Events	**FMI Standard** Primitives	**S-Functions** Primitives
Initialization	Sequence of assignments in initialization mode	Initialization function defined by the `mdlStart` callback method
Input Signals Reading	`fmi2SetInteger`	Parameters passed to Compute Output function by value
Output Signals Writing	`fmi2GetInteger`	parameters passed to Compute Output function by reference
Simulation Cycle Execution	`fmi2DoStep`	Output function defined by the `mdlOutputs` callback method

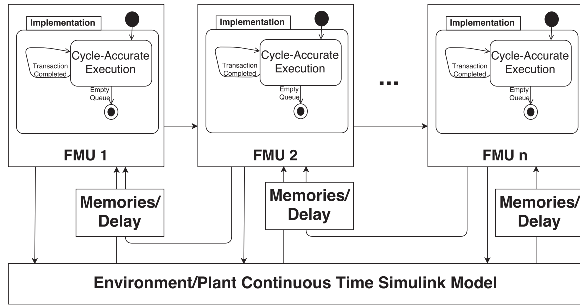


Figure 4: Naïve structure of Simulink model with multiple computational components.

HDL model initialization (*i.e.*, ①) is reproduced by two FMI primitives. The `fmi2Instantiate` function allocates the memory necessary to load the model, and creates the indexes identifying the external variables of the FMU. The `fmi2SetupExperiment` function leads internal values to their initial configuration. To reproduce the same behavior in an S-Function it implements a *setup function* specified by the `mdlStart` method of the S-Function configuration file. The setup function leads internal values to their initial configuration.

HDL input and output events (*i.e.*, ②) are reproduced in FMI by the `fmi2SetInteger` and `fmi2GetInteger` functions, respectively. They are called by the simulator at each simulation cycle for all the variables exported by the FMU as input or output variable. After the data-type abstraction is performed to generate the C cycle-accurate models, all the variables are integer. S-Functions input and output variables are parameters of the output function. Inputs are passed to the function by value, while outputs by reference.

Simulation of cycle-accurate models is a sequence of periodic executions (*i.e.*, ③). Once a cycle-accurate model is transformed into an FMU, the `fmi2DoStep` primitive is called at each clock cycle. The same behavior is reproduced using S-Functions by defining an *output function*, specified by the `mdlOutputs` callback method. The output function takes the input values and the references of the output variables as parameters.

B. Monolithic model approach

Blocks (*i.e.*, FMU or S-Function) generated applying the mapping discussed above can be inserted in a data-flow model, such as the ones used in Simulink. Designers may be led to model a system using different HW IPs trivially connecting the generated blocks according to the structure in Figure 4, without any particular precaution. This may lead to errors: synchronization problems arise while trying to represent HW

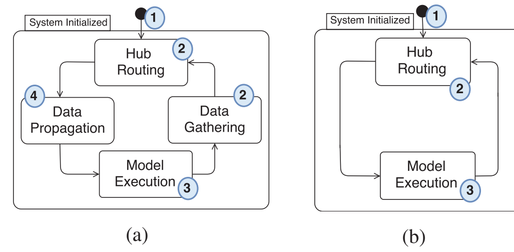


Figure 5: State charts representing the FSMs used to manage synchronization locally to (a) FMUs and (b) S-functions.

models using data-flow-based tools because of the different execution semantics involved. Data-flow models are intrinsically sequential, while HDL semantics is concurrent.

A first approach delegates integration to the abstraction procedure applied to create cycle-accurate models. It schedules, and thus integrates, any set of synchronous and asynchronous processes of an HDL hierarchical model. Thus, different HW IPs can be hierarchically composed into a unique HDL description and abstracted into its equivalent cycle-accurate model. Finally, it undergoes the mapping proposed above.

This solution manages the synchronization and communication between digital HW components internally to the monolithic model of the HW platform. Unfortunately, it requires defining the entire HW architecture of the system before its integration in the cyber-physical virtual platform. Thus, the designer must sacrifice the possibility to replace components freely.

C. Hub-based approach

The second solution reconciles the models of computation involved, by enabling concurrent execution of components within data-flows. The main idea is to decouple IPs execution from their synchronization and communication. IPs must read their inputs and perform one execution step to produce their outputs before to exchange data with each other.

All the clusters of communicating IPs are identified. **Communication, synchronization, and data exchange are managed globally** for each cluster by using a **dedicated module called Hub**. Then, the **synchronization is managed locally to each module** by using a **Finite State Machine (FSM)**.

Two situations may occur while integrating modules into a CPS simulator using integration technologies when considering the algorithm regulating the input-reading, execution and output-writing phases (*e.g.*, the FMI master algorithm for co-simulation): the algorithm can be fixed or it can be configurable. If the algorithm is fixed, all the possible sequences of primitive calls must be managed. Otherwise, some optimization may be possible. Figure 5 shows the FSMs managing the two cases. The machine in Figure 5a is used when the scheduler is fixed, the machine in Figure 5b is used otherwise. The reference FMI standard toolbox available for Simulink [14] fixes the sequence of actions for each simulation cycle to *execution, output, input*: thus requiring the first solution. S-Functions grant more freedom: the solely “output function” is executed at each simulation cycle. It takes care internally of reading input, executing and writing output: it thus allows to use the second solution.

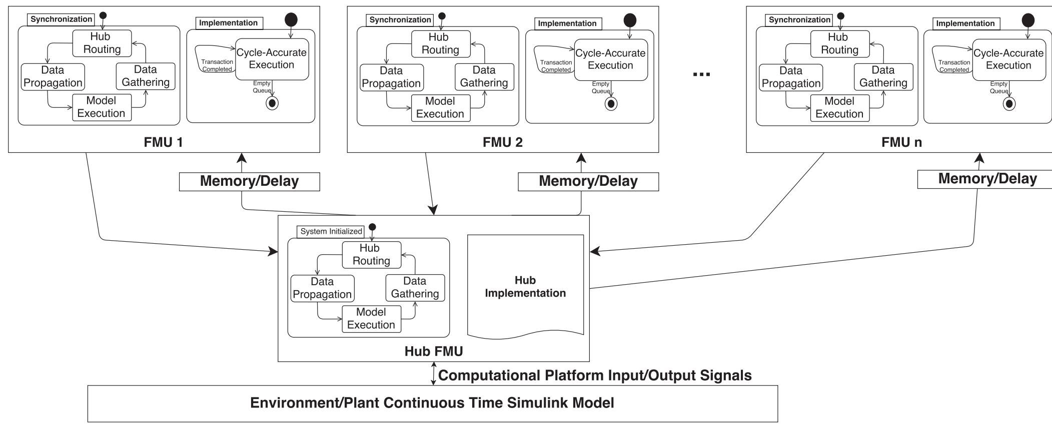


Figure 6: Structure of the simulation environment combining the continuous-time model of the physical plant and a computational infrastructure composed by multiple FMUs.

The states in Figure 5 are hereby described referring to the mapping defined above for the FMI standard:

- *Hub Routing*: the Hub updates its output variables using the values of its input variables read at the previous execution: it mimics the signals propagation defined when binding signals in RTL models. FMUs implementing components of the platform do not perform any computation.
- *Data Propagation*: the values written by the hub are stored into Simulink memory units, allowing to break algebraic loops. FMUs implementing components and the hub are not performing any computation.
- *Model Execution*: the FMUs modeling components execute one simulation cycle, write their output variables and update their input values reading memories output values.
- *Data Gathering*: the hub updates its inputs reading values from the components FMUs and the Simulink model.

Circled numbers in Figure 5 highlight relations between FSM states and the execution schemes to reproduce (Figure 2). *Data Propagation* (④) has no correspondence in previous schemes being an artifact stalling the execution to assure correct values propagation. The 4-state FSM can manage the most critical family of scheduling sequences allowed by the FMI standard, *i.e.*, simulation cycles with input reading operations scheduled after the model execution.

On the other hand, if the scheduler is not using a critical sequence it is possible to use the 2-state FSM. The latter is the case when executing S-Functions where we are free to impose input-reading as the initial action of each simulation step.

Each module (*i.e.*, FMU or S-Function) contains both its cycle-accurate implementation and the chosen FSM. It is now possible to define a schematic structure overcoming problems emerged using the structure in Figure 4. Multiple modules representing different IPs are connected to each other as depicted in Figure 6. Such a structure is required by the Hub-based approach to faithfully reproduce the communication scheme between multiple HDL models. All the input or output ports of modules produced from IPs are connected to the module implementing the hub. The latter takes care of propagating values from each module to the others, and from the modules implementing the HW infrastructure of the system to the remainder of the model. Finally, the simulation step of each module is set to be equal to the device's *clock period divided by the number of states* in the FSM. In this way, the hub module "parallelizes" components' executions, while interfacing them to the sequential semantics of the model.

Table II: Comparison between feature of the state-of-the-art [4] methodology and the proposed approaches.

Interface Technology	Synch. Approach	Components Reuse	Automatic generation	Seamless integration	Portability
FMI	Token-based [4]	×	×	×	✓
	Monolithic Model	✓	✓	×	✓
	Hub-based	✓	✓	✓	✓
S-Functions	Monolithic Model	✓	✓	×	×
	Hub-based	✓	✓	✓	×

D. Alternatives Taxonomy

Table II summarizes the desired features supported by the different alternatives proposed by this paper. It compares them to the state-of-the-art, (*i.e.* token-based synchronization methodology [4]). The approach found in the literature is thought to be used in a top-down scenario, where components reuse is not considered. Despite its portability, granted by the FMI Standard, it does not support automatic generation and seamless integration since it requires to enrich interfaces of discrete-event models adding event ports.

The other entries in the table point out the features of solutions combining the interfacing technologies and synchronization approaches proposed above. Contrary to S-Functions, FMI-based solutions are always portable. Both interfacing techniques, as well as both the synchronization approaches, allow component reuse and automatic model generation. The Hub-based approach provides seamless integration: it allows to easily modify the structure of the digital devices minimizing the number of model regenerations. The monolithic models require to re-generate the module at each change of the HW architecture. As such, the former is more suitable for design-space exploration. However, Section IV will show that monolithic models provide faster simulation.

IV. Methodology Application

The methodology has been implemented in a tool on top of HIFSuite [7] to exploit its already available abstraction

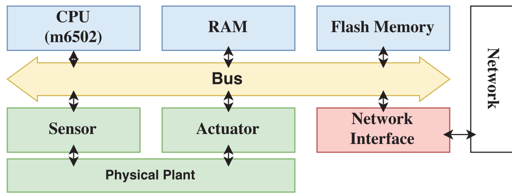


Figure 7: Architecture of the CPS used as case study. Colors represents the original modeling languages of the components: blue for Verilog, yellow for VHDL, green for Simulink, while red closed source HDL IPs.

features [5]. The tool has been applied to a case study composed of a physical plant controlled by a SW running on a CPU. The case study has been composed to be representative of many CPS families, and Figure 7 shows its structure.

Our approaches have been used to integrate accurate models of the digital components into Simulink, building a cycle-accurate cyber-physical virtual platform. Then, we use this case study to evaluate the efficiency of the proposed methodology regarding simulation speed, while the effectiveness of the methodology is shown by exploiting a cycle-accurate virtual platform to perform some design-space exploration steps.

A. Simulation performance

Last column of Table III reports the time (seconds) required to simulate 1 second of the real system by using different alternatives of the virtual platform. The first column enumerates scenarios. All the simulations have been executed on an Intel i7-3770 CPU at 3.40GHz and 16 GB of DDR3 Ram Machine running Linux Ubuntu 16.04 and Mathworks' Simulink 8.8.

Simulation time is obviously minimized when modeling the entire system within Simulink (Scenario 0). However, Section IV-B will point out how this model lacks in accuracy.

Scenarios 5 to 7 relies on the FMI standard and the state-of-the-art synchronization methodology in [4]. They can be compared to scenarios implementing the hub-based synchronization (Scenarios 1 to 3 using S-Functions and 8 to 10 using FMI). S-Functions always outperform both alternatives. The token-based approach is slightly faster than the FMI-implemented hub-based technique. However, [4] is purely top-down, and it does not easily support reuse of components. Moreover, generation of FMUs cannot be automatized, depending on FMUs instantiation. All the approaches we proposed are instead thought to maximize component reuse, automatic generation, and integration.

Scalability is analyzed by comparing Scenarios 1, 5 and 8, respectively to Scenarios 3, 7 and 10. The analysis is shown in Figure 8 reporting the simulation time of each approach (y-axis), given the number of instantiated units (x-axis). The constant overhead required by each approach must be evaluated when two units are instantiated. The *Hub-based* synchronization has a heavier constant overhead *w.r.t.* to [4] when using the same interface technology, *i.e.* FMUs. Nonetheless, the *Token-based* approach overhead grows faster adding units: from 5.16 to 10.74 seconds (2.08x) for [4], from 11.90 to 15.94 seconds (1.33x) for the Hub-based approach using FMI: the Hub-based approach seems to be more suitable when virtual platforms are composed of many digital modules. The Hub-based synchronization implemented

Table III: Simulation overhead necessary to simulate one minute of the system by using different modeling alternatives.

#	Interface Technology	Synchronization Methodology	Configuration	Execution Time (s)
0	–	–	Stateflow controller in Simulink model	0.32
1	S-Functions	Hub-based	4 C MEX Compiled S-Functions	4.29
2			3 C MEX Compiled S-Functions	3.78
3			2 C MEX Compiled S-Functions	3.63
4		Monolithic Model	1 C MEX Compiled S-Function	1.98
5	FMI	Token-based [4]	4 FMUs	10.74
6			3 FMUs	8.88
7			2 FMUs	5.16
8		Hub-based	4 FMUs	15.94
9			3 FMUs	13.14
10			2 FMUs	11.90
11		Monolithic Model	1 FMU	2.61

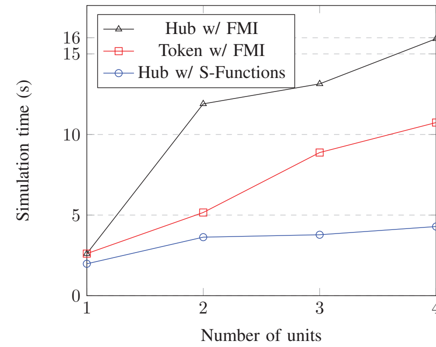


Figure 8: Scalability of the proposed approaches.

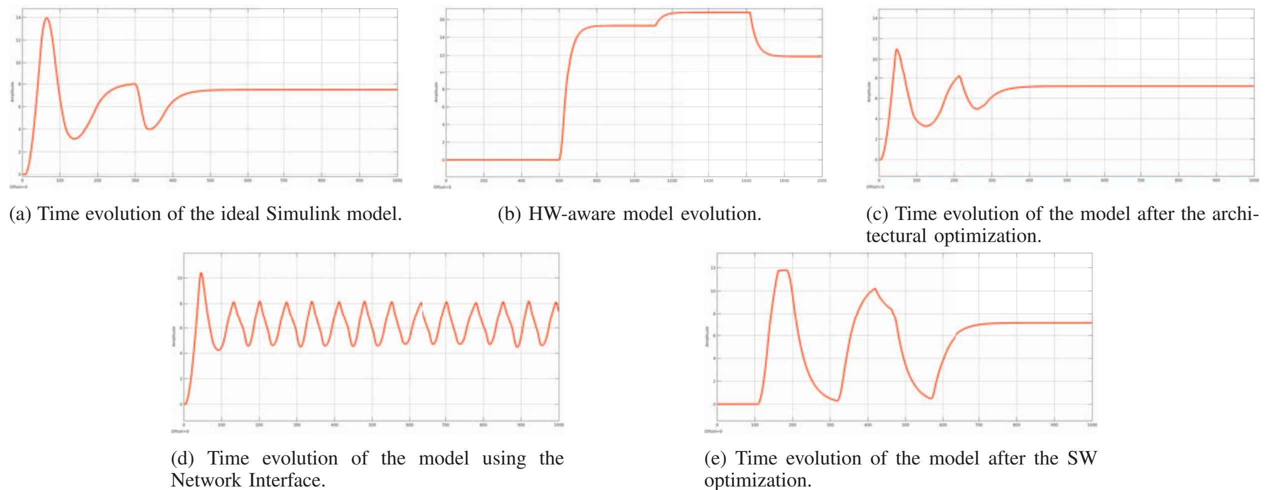
using S-Functions outperforms both the others. S-Functions introduce lighter overhead than FMUs. Furthermore, doubling the number of units the simulation time grows slower: from 3.63 (Scenario 3) to 4.29 seconds (Scenario 1) (1.18x).

Finally, Scenarios 4 and 11 synchronize IPs by exploiting automatic abstraction and integration. They provide the fastest virtual platforms and they are the best choice to effectively execute SW and not to explore alternative platforms.

B. Design Space Exploration

Figure 9 summarizes the development and refinement steps carried on to obtain the controller HW/SW implementation. It aims at showing the benefits of embedding cycle-accurate models within cyber-physical virtual platforms:

- The system is originally modeled by using Simulink (Scenario 0 of Table III). The HW/SW controller is modeled using Stateflow. Figure 9a shows the model's time evolution. The Controller easily manages the system leading to stability after around 450 seconds.
- The model is refined by introducing the FMUs implementing the cycle-accurate models of HW components. A SW running on the CPU implements the controller. HW resources are very limited: fixed-point arithmetic is not available, and the bus causes a communication bottleneck. Figure 9b shows the new model evolution: the SW cannot produce the actuation signals in time to control the system due to architectural constraints.



(a) Time evolution of the ideal Simulink model.

(b) HW-aware model evolution.

(c) Time evolution of the model after the architectural optimization.

(d) Time evolution of the model using the Network Interface.

(e) Time evolution of the model after the SW optimization.

Figure 9: Time evolution of the CPS model in the different phases of SW design exploration and optimization.

- Analyzing the previous model execution may help designers: SW-implemented multiplications was burdening the SW. As such, the designer inserts a further IP providing HW fixed-point arithmetic. Figure 9c shows the system reaching stability after 400 seconds.
- Previous models were not considering the Network Interface peripheral that requires a SW-implemented polling mechanism. The peripheral is imported as an FMU using the *Hub-based* mechanism. The SW is modified to send data through the network. Figure 9d shows that the polling mechanism introduces overhead causing the controller inability to control the system.
- The SW has been modified to run while the Network Interface is sending data. The previous model allows identifying the part of computation that can be executed while waiting for the peripheral to complete. In the evolution depicted in Figure 9e the system is now stable after 700 seconds.

The steps just presented show the importance of relying on accurate models when developing control SW for CPS. They highlight the positive impact of automatic integration of cycle-accurate models into cyber-physical virtual platforms.

V. Conclusions

This paper enables integration of cycle-accurate models within dynamic system simulators, to create cyber-physical virtual platforms. It proposes two integration solution to efficiently synchronize the discrete-event and data-flow models of computation. We evaluated the different techniques by using the same case study, highlighting for each alternative its advantages and drawbacks. This analysis may guide designers to choose the solution better suited to any design phase.

Experiments showed the benefits of using cyber-physical virtual platforms. In particular, a case study clearly shows the importance of using cycle-accurate models when developing timed-constrained control SW for CPSs.

References

[1] B. Bailey and G. Martin, "Codesign experiences based on a virtual platform," in *ESL Models and their Application*. Springer, 2010, pp. 273–308.

[2] F. Rosa, L. Ost, R. Reis, and G. Sassatelli, "Instruction-driven timing CPU model for efficient embedded software development using OVP," in *Proc. of IEEE ICECS 2013*, pp. 855–858.

[3] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty, "Multi-Objective Co-Optimization of FlexRay-Based Distributed Control Systems," in *Proc. of IEEE RTAS 2016*, pp. 1–12.

[4] S. Tripakis, "Bridging the semantic gap between heterogeneous modeling formalisms and FMI," in *Proc. of IEEE SAMOS 2015*, pp. 60–69.

[5] 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.

[6] ARM. Carbon Model Studio. http://carbondesignsystems.com/.

[7] 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.

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

[9] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale, "Virtual Prototyping of Cyber-Physical Systems," in *Proc. of ASPDAC 2012*, pp. 219–226.

[10] 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 ACM/IEEE DATE 2014*, pp. 1–4.

[11] M. Lora, S. Vinco, E. Fraccaroli, D. Quaglia, and F. Fummi, "Analog Models Manipulation for Effective Integration in Smart System Virtual Platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[12] E. Fraccaroli, M. Lora, and F. Fummi, "Automatic abstraction of multi-discipline analog models for efficient functional simulation," in *Proc. of IEEE/ACM DATE 2017*, pp. 662–665.

[13] W. Chang, D. Roy, L. Zhang, and S. Chakraborty, "Model-based design of resource-efficient automotive control software," in *Proc. of IEEE/ACM ICCAD 2016*, pp. 1–8.

[14] Mathworks, Matlab, "Simulink/Stateflow."

[15] A. Naderlinger, "Simulating Preemptive Scheduling with Timing-aware Blocks in Simulink," in *Proc. of ACM/IEEE DATE 2017*.

[16] F. Pecheux, C. Lallement, and A. Vachoux, "VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 2, pp. 204–225, feb 2005.

[17] W. Li, X. Zhang, and H. Li, "Co-simulation platforms for co-design of networked control systems: An overview," *Control Engineering Practice*, vol. 23, pp. 44–56, 2014.

[18] 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 DSD 2016*, pp. 1–8.

[19] 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 FDL 2017*, pp. 1–8.

[20] W. Snyder, P. Wasson, and D. Galbi, "Verilator-Convert Verilog code to C++/SystemC," 2012.

[21] 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 ACM EMSOFT 2013*.