Moving from Co-Simulation to Simulation for Effective Smart Systems Design

Franco Fummi[1,2], Michele Lora[2], Francesco Stefanni[1], Dimitrios Trachanis[3], Jahn Vanhese[3] and Sara Vinco[2]
[1]EDALab s.r.l. - Verona, Italy {name.surname}@edalab.it
[2]Department of Computer Science - University of Verona, Italy {name.surname}@univr.it
[3]Agilent Technologies - Sint-Denijs-Westrem, Belgium {name.surname}@agilent.com

***Abstract*—Design of smart systems needs to cover a wide variety of domains, ranging from analogue to digital, with power devices, micro-sensors and actuators, up to MEMS. This high level of heterogeneity makes design a very challenging task, as each domain is supported by specific languages, modeling formalisms and simulation frameworks. A major issue is furthermore posed by simulation, that heavily impacts the design and verification loop and that is very hard to be built in such an heterogeneous context. On the other hand, achieving efficient simulation would indeed make smart system design feasible with respect to budget constraints. This work provides a formalization of the typical abstraction levels and design domains of a smart system. This taxonomy allows to identify a precise role in the design flow for co-simulation and simulation scenarios. Moreover, a methodology is proposed to move from the co-simulated heterogeneity to a simulatable homogeneous representation in C++ of the entire smart system. The impact of heterogeneous or homogeneous models of computation is also examined. Experimental results prove the effectiveness of the proposed C++ generation for reaching high-speed simulation.**

I. Introduction

Smart systems represent a broad class of systems defined as intelligent, miniaturized devices incorporating functionalities like sensing, actuation, and control. In order to support these functions, they must include sophisticated and heterogeneous components and subsystems such as: application-specific sensors and actuators, multiple power sources and storage devices, intelligence in the form of power management, baseband computation, digital signal processing, power actuators, and subsystems for various types of wireless connectivity (as shown in Figure 1). Smart components and subsystems are developed and produced with very different technologies and materials specific to the corresponding domain and technology.

In this context, simulation is a very critical task, as each domain of component adopts specific tools and framework, that do not cover the whole smart system heterogeneity. On the other hand, simulation is a key phase in the design and verification process of a system, as it heavily impacts time-to-market and the competitiveness of the final product.

The goal of this paper is to ease simulation and validation of smart systems with three main contributions.

- A *taxonomy of abstraction level/design domains*. This allows to identify a precise role in the design flow for co-simulation and simulation scenarios, and thus to outline the possible strategies for gaining correct simulation of smart systems.
- A comparison between two complementary approaches, simulation and co-simulation, showing their strengths and weaknesses.

This work has been partially supported by the European project SMAC FP7-ICT-2011-7-288827.

978-3-9815370-2-4/DATE14/©2014 EDAA

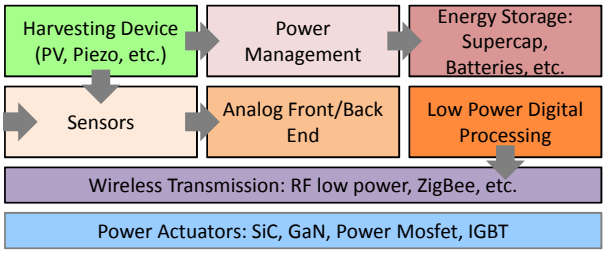

Fig. 1. Typical components of a smart system

- Enhancement of *reuse and integration* by showing how to ease the adoption of homogeneous simulation, with automatic C++ generation from lower abstraction levels and automatic integration of heterogeneous interfaces.

II. MoCs and Frameworks for Smart System Modeling

In literature, the main approaches proposed for handling heterogeneity are (i) top-down flows, relying either on model-based design (MBD) or on a models of computations (MoC), and (ii) co-simulation [1].

In *MBD approaches*, the system model is at the center of the design process and it is continually refined throughout a strictly top-down development flow [2]–[4]. Components following different synchronization mechanisms are put together with data conversion mechanisms that must be implemented manually, and that do not guarantee a correct integration.

Several *MoCs* have been proposed also to describe different aspects of smart systems, such as Extended Finite State Machines (EFSMs) [5] or Hybrid Automata [6]. Unfortunately, every MoC is a stand alone environment that can not cover all the domains comprised in smart system development.

The complementary approach is to integrate existing components in a bottom-up flow. This is realized with *co-simulation* environments where each component is simulated in its native environment and framework [7], [8]. Co-simulation assemblies heterogeneous components without providing a rigorous formal support, and it only moves the problem of integrating heterogeneity components to the problem of integrating different simulators.

In this context system integrators, such as SystemVue [9], have been proposed. In SystemVue a system is described as a schematic of components connected with wires and busses. The simulation technology is based on a Data-Flow MoC and is related to the Berkeley Ptolemy multidomain, heterogeneous simulation platform [3]. SystemVue provides numerous libraries with parametrized components and interfaces to diverse modeling formats, ranging from MATLAB to the main HDLs. Moreover, it provides a C++ API to create libraries of custom components.

	MEMS, sensors and actuators	*Power sources*	*Discrete and power devices*	*Analog and RF*	*Digital HW*	*Embedded SW*
Transactional	SystemVue	SystemVue	SystemVue	SystemVue	SystemVue SystemC TLM	SystemVue QEMU
Functional	C++	C++	C++	C++, SystemVue,	C++, SystemC	**QEMU**
Structural	ADS, AMS HDL, Matlab, MEMS+	Matlab, Simulink AMS HDL	ADS	AMS HDL, ADS, Matlab, ADS	**RTL HDL**	Cycle accurate QEMU
Device	MEMS+, Matlab, AMS HDL	FEM, Spice	EMPro, Spectre, Momentum	EMPro, Spectre, Momentum	AMS HDL	-
Physical	**FEM, Matlab, MEMS+**	**FEM, Spice**	**EMPro, Spectre, Momentum**	**EMPro, Spectre, Momentum**	AMS HDL	-

SIMULATION
SIMULATION (UNIVERCM SUPPORT)
CO-SIMULATION

Fig. 2. Simulation–levels and Design domains taxonomy.

Bottom-up flows could be enhanced by using a unique MoC, capable of cover a wide range of domains and allowing reuse with automatic conversion tools. This would guarantee a clear semantics and, thus, mathematical and rigorous rules for component integration, adaptation and reuse [10]. The couple HIFSuite [11] and UNIVERCM [12] aims at providing such an environment. HIFSuite furnishes a set of back-end and front-end automatic tools for a variety of languages, while UNIVERCM is an automaton-based MoC that unifies the modeling of both analog and digital domains.

III. Simulation Platform for Smart System Design

A. Simulation levels and Design domains taxonomy

Simulation and design are heavily influenced by the abstraction level of a component or a system. It is thus necessary to clearly identify the abstraction level involved in smart system design and to associate each domain and simulator to the correct level. The main factors determining the level of abstraction are: time granularity, interconnection model, state space granularity and data aggregation. *Time granularity* may be continuous or discrete time, or follow an event based semantics where time ticks only when the system state changes. The *interconnection model* describes communication and synchronization between components as potential or flow quantities, flow charts or transactions. The *granularity of state space* details data aggregation for simulation purposes, *i.e.*, variables managed by differential equations, symbolic variables or objective constructs. Finally, *data aggregation* states whether the component is modeled by considering the minimum (black box) or maximum (clear box) number of state space variables necessary for a correct representation of the observable behavior.

Given these factors, it is possible to identify five main abstraction levels typical of smart systems.

- At *transactional* level, simulation is strictly event based and inter component communication happens via transactions. Variables are used to model system state.
- At *functional* level, simulation is event based but communication relies on the flow chart interconnection style.
- The *structural* level may support both continuous time evolution (modeled with differential equations and conservative laws) and discrete time evolution (with event based or a flow chart synchronization and finite set variables). A black box approach is adopted.
- At *device* level, simulation can be both continuous or discrete time. A clear box approach is adopted.
- The *physical* level adopts continuous time synchronization and the conservative interconnection style. State space granularity is described as differential equations and with a clear box approach.

Given the taxonomy and the heterogeneous domains typically present in any smart system, it is possible to build a design-domains/simulation-level matrix, shown in Figure 2. Such a chart allows to identify the abstraction level (rows) and the domain (column) of the most widespread tool and languages in the context of smart systems. This allows to correctly differentiate the use of co-simulation and simulation. Text in bold shows the typical entrance level and tools for each domain.

Models of the *lowest abstraction levels* are represented by different design languages, thus they must be simulated by using their own simulator (e.g., Matlab, Modelsim, EMPro, etc.).

Moving to the *functional level*, there is a convergence in the modeling language, as all domains are represented in C++. Even if this would in principle allow simulation, the MoC implemented into each C++ model can be different from domain to domain. Thus, simulation can be built if models are coherent with respect to the same MoC. As a result, the chosen MoC must be able to cover all domains. As outlined in Section III, UNIVERCM proved to cover most of the heterogeneous domains. If UNIVERCM allows to generate C++ code (Section III-B), then all C++ components can be simply linked together to simulate a design covering more than one domain.

At *transaction level*, the communication protocol is common to all domains as it relies on transactions. SystemVue covers all domains and thus it can be used as a general integration framework (Section III-C). This allows to simulate in a coherent way functional models based on different computation models.

B. C++ Code Generation from UNIVERCM

UNIVERCM allows to reconcile heterogeneous domains to a unique MoC, thanks to automatic generation of the homogeneous format and thus enhancing bottom-up flows. The reconciled version of each automata can be then mapped to a target language, such as C++, by defining formal transformations, with the goal of generating an implementation of the whole system in a single language [13].

Each UNIVERCM automaton is mapped to a C++ function, representing the whole automaton evolution, as depicted in Figure 3. The function body is built as a `switch` statement, where each case represents one of the automaton states. Each *edge* is implemented as an `if` or `else if` statement, whose guard is a logic and of the enabling condition of the edge and of the activation condition on synchronization events. *Continuous evolution* is implemented as a discretized implementation of the flow predicate, by adopting one of the many numerical algorithms to solve differential equations.

Code generated from the UNIVERCM automata is ruled by a *management function*, in charge of activating automata and of managing the status of the overall system and parallel composition of automata. Whenever the system is made of more subcomponents, automata have to be composed by checking the correspondence between variables and synchronization events

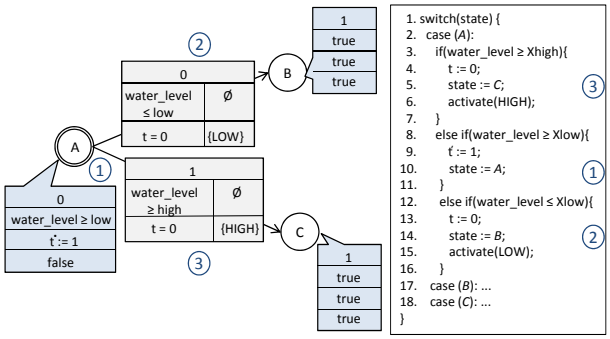

Fig. 3. UNIVERCM automaton to be converted to C++ (left) and corresponding generated code (right)

of the two. This mapping must be identified by the designer, to allow the management function to propagate the correct values.

C. SystemVue C++ integration

The first step to integrate a C++ external component in SystemVue is to specify its interface as names and data types of all the inputs, outputs and parameters. The behavior of the C++ code is then described by respecting the SystemVue API through four C++ methods. The Setup() method is used to specify the rate of each port. The Initialize() method contains all initialization code and it is automatically invoked after the simulator has calculated the schedule for the simulation run. The Run() method performs any actions that the model needs to perform during simulation. Finally, the Finalize() method performs any post-simulation coding.

In this way, SystemVue allows to define and embed custom C++ code. By means of HIFSuite, this SystemVue interface can exploited to integrate digital components from various HDL descriptions.

D. Impact of MoCs on simulation and co-simulation performance

Adopting a single MoC allows to avoid co-simulation and overheads due to data sharing and time synchronization. The taxonomy in Figure 2 helps in further understanding the impact of MoCs and of heterogeneity on simulation and co-simulation at different abstraction levels.

As mentioned in Section III-A, lowest abstraction levels are represented by different design languages and MoCs. As a result, each domain must be simulated by using their own simulator (e.g., Matlab, Modelsim, EMPro, etc.). Co-simulation frameworks are thus built by connecting different simulators, such as shown in [7], [8]. However, explicitly modeling synchronization between simulators heavily impacts simulation performance and effectiveness [14]. Other approaches have a lighter impact, by compiling separately the different formats and linking them together as done by ModelSim to co-simulate SystemC and VHDL. This lighter approach is still affected by the presence of heterogeneous MoCs, as the data sharing mechanism and time synchronization introduce a heavy overhead.

Functional level brings to a convergence in terms of modeling language. Here MoCs shows their impact to the full. If all C++ components follow the same MoC, then they can easily integrated with no further overhead. Else, if the adopted MoCs are heterogeneous, it becomes necessary to introduce a communication layer applying data and synchronization conversion.

This can be easily achieved at *transaction level*, as transactions and standard interfaces force a communication protocol. This mitigates the effect of having multiple MoCs.

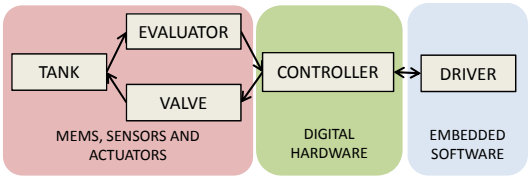

Fig. 4. Overview of the water tank system compositions.

IV. Experimental Results

A. Water tank system

An overview of the water tank system is given in Fig. 4. The *tank* component, characterized by an uncontrolled outbound water flow, belongs to the Analog domain. Two components may belong to the MEMS sensors and actuators domain: a *valve*, whose aperture affects the incoming flow of water and an *evaluator*, that checks the level of water in the tank. The system includes also a *controller*, that acts on the aperture of the valve (digital HW domain) and a *software driver* that sets the legal upper and lower bounds accepted for the water level (embedded SW domain). The heterogeneity in terms of domains and abstraction levels makes simulation with a single environment or language impossible.

Six different descriptions of the system have been chosen (see Table I):

1. *Heterogeneous simulation*: the components are described using different HDLs and integrated within a unique tool (*i.e.*, ModelSim).
2. *Mixed top-down/bottom-up approach*: the tank is modeled by using the graphical editor provided with SystemVue. The controller is a VHDL component imported in the SystemVue environment while the remaining components have been synthesized in C++ via UNIVERCM. In this case, ModelSim needs to be co-simulated with SstemVue.
3. *Homogeneous co-simulation*: the tank is modeled by exploiting the SystemVue modeling tool, while all the other components are imported into SystemVue as C++ modules.
4. *SystemVue simulation*: all C++ descriptions are automatically generated and integrated by using SystemVue.
5. *UNIVERCM-SystemC simulation*: SystemC code is generated from the UNIVERCM representation of all components (as explained in [14]).
6. *UNIVERCM simulation*: a purely C++ description obtained via UNIVERCM by exploiting the methodology proposed in Section III-B.

To compare all the approaches, 1000 seconds of execution of the given plant have been simulated. Results are reported in Table I. Heterogeneity has a bad impact on simulation performance. Indeed, mixing different tools, as for description 2, leads to a significant performance degradation due to synchronization mechanisms among tools. Descriptions 1 and 3 highlight that also mixing languages and formalisms reduces the performance. A good performance is achievable by connecting components described in a homogeneous way, by either using an aggregation tool such as SystemVue (description 4) or the SystemC scheduler (description 5). However, the best result is obtained by the UNIVERCM simulation, thanks to its homogeneity both in terms of used tools, computational model and aggregation.

B. SMAC open–source platform

The different available approaches have been applied also to the smart system depicted in Figure 5. The platform is

TABLE I. SIMULATION TIME OF THE WATER TANK SYSTEM BY USING THE DIFFERENT PROPOSED DESCRIPTIONS.

#	Simulation Framework	Tank	Valve	Evaluator	Controller	Driver	Execution Time (ms.)
1	Heterogeneous simulation	ModelSim (SystemC)	ModelSim (SystemC)	ModelSim (SystemC)	ModelSim (VHDL)	ModelSim (SystemC)	1,427
2	SystemVue-based co-sim.	SystemVue	SystemVue (C++)	SystemVue (C++)	ModelSim (VHDL)	SystemVue (C++)	49,001
3	UNIVERCM – SystemVue Heterogeneous Simulation	SystemVue	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	1,516
4	SystemVue simulation	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	505
5	UNIVERCM SystemC Simulation	UNIVERCM SystemC	UNIVERCM SystemC	UNIVERCM SystemC	UNIVERCM SystemC	UNIVERCM SystemC	474
6	UNIVERCM simulation	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	154

TABLE II. SIMULATION TIME OF THE SMAC OPEN-SOURCE PLATFORM BY USING THE DIFFERENT PROPOSED DESCRIPTIONS.

#	Simulation Framework	XYAxis	Mlite-CPU	SW Application	UART	Memory	RF Transceiver	APB Bus	Execution Time (ms.)
1	SystemVue-based co-simulation	ADS (Verilog-A)	ModelSim (VHDL)	SystemVue (C++)	ModelSim (VHDL)	ModelSim (VHDL)	ModelSim (SCNSL)	ModelSim (VHDL)	509,002
2	SystemVue simulation	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	SystemVue (C++)	21,984
3	UNIVERCM Simulation	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	UNIVERCM C++	21,118

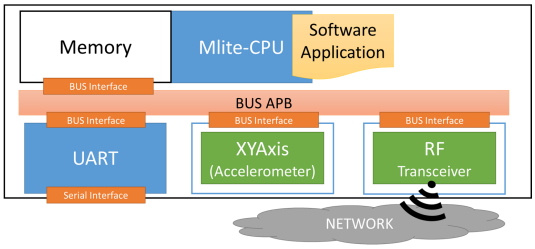

Fig. 5. Overview of the SMAC open–source platform compositions.

widely heterogeneous, and it is composed by: (1) a *XYAxis accelerometer*, i.e., a MEMS sensing data related to movements of the system and (2) a MIPS processor (the *Mlite-CPU*), belonging at the digital HW domain, executing (3) a *Software application* elaborating data sensed by the accelerometer and stored in (4) a *Memory*. Communication is managed by another digital component: a (5) *Universal Asynchronous Receiver/Transmitter (UART)*, providing a serial interface to the sensor. (6) A *RF Transceiver*, used to send and receive data from other smart sensors, and (7) an *APB Bus* used to connect all the components composing the platform.

The effectiveness of the approach has been proved comparing three different descriptions of a simplified version of the system: heterogeneous co-simulation, SystemVue-based simulation and UNIVERCM simulation. All simulations are run to reproduce 1000 seconds of the real platform execution. Results, shown in Table II, show that the use of a unique MoC (i.e., UNIVERCM simulation) gives excellent performance and it allows to efficiently simulate the system. Employing an aggregation tool, such as SystemVue, gives good performance further enhanced by the easiness of integration, that does not require to uniform the adopted MoC. Employing state of the art techniques, such as heterogeneous co-simulation, leads to a huge communication and synchronization overhead.

V. CONCLUSIONS

The paper tackled the heterogeneity of smart systems. It built a taxonomy of abstraction level/design domains to identify a precise role in the design flow for co-simulation and simulation scenarios. It highlighted the importance of MoCs in the performance and integration difficulties of smart system integration. Finally, it proposed a methodology to convert a heterogeneous description into a homogeneous representation based on C++. The experimental results show the importance of removing heterogeneity. Benefits are obtained using a unique language and optimality employing a single computational model. The paper also shows a comprehensive methodology that easies and speeds up the design and simulation, with a positive effect on time-to-market and competitiveness. Future extensions of this work will address the accuracy concerning analog behavior and its trade-off with respect simulation performance.

REFERENCES

[1] G. De Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers, 2001.

[2] T. MathWorks, "Stateflow: Design and simulate state machines and control logic," URL: http://www.mathworks.com/products/stateflow/.

[3] E. A. Lee, "Overview of the Ptolemy project," 2001, http://ptolemy.eecs.berkeley.edu.

[4] F. Balarin, Y.Watanabe, H. Hsieh, L. Lavagno, C. Passarone, and A. Sangiovanni-Vincentelli, "Metropolis: an Integrated Electronic System Design Environment," in *Computer*, vol. 36, 2003, pp. 45 – 52.

[5] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[6] T. Henzinger, "The Theory of Hybrid Automata," in *IEEE Symposium on Logic in Computer Science (LICS)*, 1996, pp. 278 – 292.

[7] F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E. Aboulhamid, "A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation," in *Proc. of IEEE BMAS*, 2007, pp. 1 – 6.

[8] F. Fummi, M. Loghi, M. Poncino, and G. Pravadelli, "A cosimulation methodology for hw/sw validation and performance estimation," *ACM Trans. DAES*, vol. 14, pp. 23:1–23:32, April 2009.

[9] Agilent Technologies, "SystemVue Electronic System-Level (ESL) Design Software," URL:http://www.home.agilent.com/en/pc-1297131/systemvue.

[10] E. Lee and A. Sangiovanni-Vincentelli, "Component-based design for the future," in *Proc. of ACM/IEEE DATE*, vol. 1, 2011, pp. 1–5.

[11] EDALab s.r.l., *HIFSuite*, URL: http://www.hifsuite.com.

[12] L. D. Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, and S. Vinco, "UNIVERCM: the universal versatile computational model for heterogeneous system integration," in *IEEE TCOMP*, vol. 62, Feb. 2013, pp. 225 –241.

[13] F. Fummi, M. Lora, F. Stefanni, and S. Vinco, "Code generation alternatives to reduce heterogeneous embedded systems to homogeneity," in *Proceedings of Forum on Design Languages and specification (FDL '13)*, Sep. 2013.

[14] L. D. Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, and S. Vinco, "UNIVERCM: the UNIversal VERsatile Computational Model for heterogeneous system integration," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 225–241, 2013.