POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Unifying Flow to Ease Smart Systems Integration

Original
A Unifying Flow to Ease Smart Systems Integration / Lora, Michele; Vinco, Sara; Fummi, Franco. - ELETTRONICO. - (2016), pp. 113-120. (Intervento presentato al convegno International High-Level Design Validation and Test Workshop (HLDVT), 2016 tenutosi a Santa Cruz, California nel 7-8 Ottobre 2016) [10.1109/HLDVT.2016.7748264].

Availability:
This version is available at: 11583/2651280 since: 2020-02-22T21:35:40Z

Publisher:
IEEE

Published
DOI:10.1109/HLDVT.2016.7748264

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

03 May 2024

A Unifying Flow to Ease Smart Systems Integration

Michele Lora
Dept. of Computer Science
University of Verona, Italy
michele.lora@univr.it

Sara Vinco
Dept. of Computer Engineering
Politecnico di Torino, Italy
sara.vinco@polito.it

Franco Fummi
Dept. of Computer Science
University of Verona, Italy
franco.fummi@univr.it

***Abstract*—This paper proposes a meet-in-the-middle approach for the modeling and simulation of heterogeneous systems. The starting point is a set of heterogeneous models, developed by adopting the designer's favorite design language and formalism. The methodology exploits automatic translation, abstraction and integration flows to generate a single homogeneous system-level description, that allows fast system simulation. The approach is supported by two novel design domain/abstraction level taxonomies, that generalize a state-of the art taxonomy [1] to identify what characteristics would allow efficient system-level simulation, together with the transformations that must be applied to the starting model to achieve them. The advantage of the approach on system design and prototyping is particularly evident on any kind of highly heterogeneous systems, such as smart systems. The overall automatic flow has been implemented and applied to an open-source case study to measure the performance of each identified abstraction level, and the impact of the proposed methodology.**

I. Introduction

Microelectronics and systems design have been pushed by the Moore's Law for the last decades, impacting not only general purpose computing, but also embedded devices. This intensive improvement of computational capabilities led designers to move functionality from HW to SW, in order to decrease costs and design time. Thus, a major trend for embedded platforms was to create general-purpose miniaturized architectures, running the SW implementation of a given functionality. At the same time, systems and circuits research in Electronic Design Automation (EDA) was mainly focused on Very Large Scale Integration (VLSI) and single Integrated Circuit (IC), to stay on Moore's Law track, thus repeatedly doubling the density of transistors into a single IC, and the number of chips in a single board [2].

The recent years saw a shift in research, that moved its focus from devices to systems. Thus, we entered in the so-called "More than Moore" era, where communication, sensing and actuation have been integrated alongside computation within the same system. Emerging technologies embed more and more intelligence into everyday life, thus pushing toward a "chimera" defined as *Smart Planet*. The idea was first proposed by IBM in 2008 [3], and it consists in leveraging the use of new technologies for a smarter management of the physical environment, ranging from energy management to traffic.

Figure 1 depicts the ideal structure of the smart planet. To realize such a system, it must be possible to:

- perform distributed sensing of the environment, to gather data about the involved physical processes;

978-1-5090-4270-8/16/$31.00©2016 IEEE

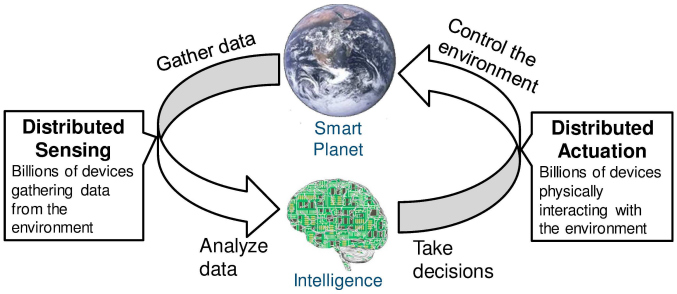

Figure 1: Main idea behind the concept of "Smart Planet".

- analyzing sensed data to correctly understand and predict the evolution of physical processes;
- take decisions on how the system should react to the current state of the physical environment;
- control the environment through actuation mechanisms;
- perform communication between different parts of the system, given the distributed nature of the problem.

As such, the design of a smarter planet can be seen as the solution of an enormous distributed control problem, where the "plant" to control is the physical environment we are living in. Thus, the environment itself must be instrumented with miniaturized components capable of performing *sensing*, *actuation*, *computation and communication*, while meeting strong extra-functional requirements (*e.g.*, thermal properties, power consumption, and reliability). Such components are the *Smart Devices* or *Smart Systems*. Thus, we might say that **Smart Systems are the building blocks of the Smart Planet**.

The tight interaction with the physical environment makes Smart Systems extremely heterogeneous objects [4], relying on countless different technologies and consequently requiring heterogeneous expertise and tools for their design. Previous work [1] showed how system simulation may benefit from reducing heterogeneity to homogeneous models, while raising abstraction in order to work at System-level. This paper goes further, by proposing a concrete automatic design flow for smart systems, that unifies heterogeneous descriptions into homogeneous models, that can be the starting point for well-established System-Level Design (SLD) flows. The proposed flow allows the designer to model each component with the most suitable language or tool, while producing highly efficient executable virtual platforms for smart system simulation.

Table I: State-of-the-art taxonomy of the main tools used for smart system design [1].

Abstraction Levels	MEMS Sensors & Actuators	Power Sources	Discrete and power devices	Analog and RF	Digital Hardware	Embedded Software	
Transactional	SystemVue	SystemVue	SystemVue	SystemVue	SystemVue SystemC-TLM	SystemVue QEMU	*Simulation*
Functional	C++	C++	C++	C++	C++	**QEMU**	
Structural	ADS, Matlab, AMS HDLs, MEMS+	Matlab, Simulink AMS HDLs	ADS	AMS HDLs, Matlab, ADS, SPICE	**RTL HDLs**	Cycle Accurate QEMU	*Co-Simulation*
Device	AMS HDL, Matlab, MEMS+	FEM models, SPICE	EMPro, Momentum Spectre	AMS HDLs, Matlab, ADS, SPICE	AMS HDLs, Matlab, ADS, SPICE	-	
Physical	**FEM models, Matlab, MEMS+**	**FEM models, SPICE**	**EMPro, Momentum Spectre**	**AMS HDLs Matlab, ADS, SPICE**	AMS HDLs, SPICE	-	

II. State of the art on modeling and simulation of heterogeneous systems and contribution

The approaches proposed in the literature to handle heterogeneity in modern devices can be divided into two main categories: *top-down* and *bottom-up* design flows [5].

Top-down flows are mostly based on the concept of Model-based Design (MBD). MBD relies on high-level models specified by the designer, that are then step-by-step refined to achieve a final implementation through a set of correct-by-construction transformations [6]. MBD frameworks, including Simulink [7] and Ptolemy [8], are extensively used in the practice by system engineers since they enable to model systems employing different Models of Computation (MoCs), and to put them in communication to integrate complex models. However, they do not easily permit component reuse: integrating already designed components requires a manual re-modeling, that may lead to errors and inconsistencies [5].

Reuse of components is well supported by bottom-up flows. In particular, *Component-based* approaches reduce complexity by assembling strongly encapsulated design entities (*i.e.*, *components*) equipped with concise and rigorous interface specifications. As such, the first challenge is to provide interface specifications that are rich enough to cover all phases of the design cycle. The IP-XACT standard [9] has been defined to allow specification of interfaces for digital IPs. However, an IP-XACT-based automatic tool-chain did not emerge at the state of the practice. Consequently, component-based approaches must still rely on complex co-simulation environments to simulate an entire heterogeneous system [10]. Such environments are usually based on a multitude of simulation tools, each of them specialized on a specific design domain. Thus, they are computationally burdened by synchronization and interprocess communication overheads. Furthermore, due to the lack of specification languages support, the integration of the components is still performed by hand in a time-consuming and error-prone process.

Recently, some solutions are emerging to mitigate the limitations of the two aforementioned approaches. Virtual Platforms, such as Imperas' OVP [11] or Cadence's Virtual System Platform [12], allow to define the computational architecture of a system by employing a high-level instruction set simulator of the underlying CPU to develop SW running on the platform. Then, system peripherals may be easily plugged into the simulation by using their Hardware Description Language (HDL) specifications, thus enhancing integration of components for component-based approaches. However, these tools are mostly focused on HW/SW co-simulation and they do not support continuous-time models. As a result, they do not permit to simulate analog-mixed signals peripherals and physical processes.

The recently proposed Functional Mock-up Interface (FMI) standard [13] facilitates to plug third-party simulators to MBD frameworks. This allows to co-simulate already designed components of the system, thus improving reuse. However, some limitations are still unsolved [14] and systematic reuse is still not possible to realize in MBD approaches.

Platform-Based Design (PBD) emerged as the prominent approach to the design of complex heterogeneous systems [15]. PBD is a meet-in-the-middle approach that merges the advantages of both top-down and bottom-up approaches. It provides a rigorous framework to reason about both vertical and horizontal integration between components that are heterogeneous in terms of design-domain or abstraction-level. However, automation for PBD is not available at the state-of-the-art. This paper aims at introducing an automatic design flow based on PBD for heterogeneous smart devices.

Concerning simulation tools and frameworks for smart systems, a first effort to categorize them has been performed in the context of the SMAC European Project [16]. A preliminary outcome of this project [1] was the taxonomy reported in Table I. The taxonomy identifies the abstraction levels (rows) and the design domains (column) of the most widespread tools and languages. Starting from this taxonomy, [1] proposes a code generation methodology that generates high-level models of the components by relying on a single MoC (UNIVERCM [17]). Experimental results showed that resulting simulation outperforms co-simulation due to synchronization overhead, and that simulation can be performed only at the higher abstraction levels, while the lower co-simulation framework are unavoidable.

A. Contribution

Some limitations are identifiable in [1]. The taxonomy (Table I) does not properly organize the design domains (*e.g.*, it does not cover the network and communication domain).

Table II: Taxonomy of the main modeling and simulation tools used for smart system design.

Abstraction Levels	MEMS Sensors & Actuators	Power Sources	Analog and Physical	Network and Communication	Digital Hardware	Embedded Software	
Transactional	SystemVue	SystemVue	SystemVue	SystemVue	SystemVue	SystemVue	
Functional	C++	C++	C++	C++	C++	C++	*Simulation*
Homogeneous Structural	SystemC (AMS)	SystemC (AMS)	SystemC (AMS)	SystemC (SCNSL)	SystemC (RTL, TLM)	SystemC (TLM)	
Structural	ADS, Matlab, AMS HDLs, MEMS+	Simulink AMS HDLs	AMS HDLs, Matlab, ADS, SPICE	NS3, OPNET, SystemC (SCNSL)	RTL HDLs	**Cycle Accurate QEMU**	
Device	AMS HDL, Matlab, MEMS+	FEM models, SPICE	EMPro, Momentum Spectre	**AMS HDLs, Matlab, ADS, SPICE**	**AMS HDLs, Matlab, ADS, SPICE**	-	*Co-Simulation*
Physical	**FEM models, Matlab, MEMS+**	**FEM models, SPICE**	**EMPro, Momentum Spectre**	AMS HDLs, Matlab, ADS, SPICE	AMS HDLs, SPICE	-	

At the same time, the organization of the abstraction levels does not allow to strictly define simulation and co-simulation, *i.e.* due to the use of QEMU at the two topmost levels. Furthermore, [1] does not provide full automation of the high-level models code generation process. Indeed, not all the design domains are supported for automatic translation into UNIVERCM [18], thus enforcing a manual modeling effort.

This paper aims at overcoming such limitations. It proposes a rationalization of the taxonomy in Table I to better define at which levels to employ co-simulation and simulation frameworks. Then, the a MoC-based generalization of the taxonomy is proposed, to identify the ideal abstraction level for system-level simulation. Finally, a fully automatic translation and abstraction flow is defined. This allows to automatically move models up in the taxonomy, in order to produce system-level models for smart design simulation at the identified level of abstraction.

III. TAXONOMY OF THE SMART SYSTEM DESIGN SPACE

This section reasons about the abstraction-levels and domains involved in smart system design, to introduce the target of the proposed design flow.

A. Taxonomy of the main tools used for smart system design

The analysis of the limitations of [1] lead to a reconsideration of Table I, and to the definition of a new taxonomy, depicted in Table II. With respect to Table I, the Analog HW domain is now generalized by the introduction of a new domain: *Analog and Physical*, that includes models expressing analog electronics or physical processes. This domain does not include MEMS designs, that usually rely on domain-specific modeling tools and techniques (and thus have a dedicated domain, *i.e.*, MEMS Sensors & Actuators). The novel definition of the Analog and Physical domain allows to absorb also the Power and Discrete Devices domain of Table I, as these devices usually rely on the same modeling and simulation technologies used for analog devices.

Secondly, Table I grouped RF models together with analog models, in a single design domain (Analog and RF). RF models can be considered within the novel Analog and Physical domain whenever they are employed in sensing and actuation tasks. Whenever RF models are instead considered as communication media, it does not seem correct to consider them as part of this domain. A new design domain, *Network and Communication*, has thus been inserted to consider such components and their network models.

The last modification involves one of the rows of Table I. The Structural abstraction level has been split into two variations: *Structural* and *Homogeneous Structural*. The former allows to connect simultaneously different tools, specific of the involved design domains. On the contrary, the enforces the adoption of a single simulation framework to represent within a single environment all the involved domains. This is the case,*e.g.*, of SystemC and its extension, that allows to cover multiple domains with a single framework, even if each component may adopt different synchronization mechanisms.

Based on this taxonomy it is possibly to correctly differentiate the use of co-simulation and simulation according to the two dimensions. In particular, in this work we define:

- *simulation* as the imitation of system behavior over time performed entirely within a single environment.
- *co-simulation* as the imitation of system behavior over time performed by the collaboration of different simulations.

What suddenly emerges analyzing the proposed taxonomy is that models at the lowest abstraction levels are implemented with different design languages, that require their own simulator. This forces to build co-simulation environments. At the state of the practice, simulation can be performed only at the highest levels of abstraction, where there is a convergence in terms of languages and tools.

When the goal is achieving fast execution for smart virtual platforms, raising the abstraction level leads thus to two main advantages. On one hand, a more abstract model needs to simulate a lower level of detail, thus providing a first speed-up. On the other hand, it makes possible to move from co-simulation to simulation, thus removing the overhead introduced by the synchronization of multiple tools. Note that automating model abstraction would give a further advantage. Executable virtual platforms can indeed be created in a manner that is seamless to designers specialized in different design domains and technologies: designers can model components

Table III: Taxonomy of the main MoCs used for smart system design.

Abstraction Levels	MEMS Sensors & Actuators	Power Sources	Analog and Physical	Network and Communication	Digital Hardware	Embedded Software	Synchronization & Concurrency
Behavioral	Discrete Events						
Transactional	Continuous Time	Continuous Time or Discrete Events	Continuous Time	Discrete Events	Discrete Events	Discrete Events	Synchronous Data Flow
Functional	Continuous Time	Continuous Time or Discrete Events	Continuous Time	Discrete Events	Discrete Events	Discrete Events	Discrete Events
Homogeneous Structural	Continuous Time			Discrete Events	Discrete Events	Discrete Events	
Structural	Continuous Time			Discrete Events	Discrete Events	Discrete Events	
Device	Continuous Time					-	Continuous Time
Physical	Continuous Time					-	

with languages and formalisms they are used to, and get at the same time the benefits of a higher level of abstraction through the adoption of automatic flows.

By looking at the taxonomy in Table II, it is quite straightforward to see that the tools and frameworks supported are only a small subset of the ones available for smart system design. The scope of the taxonomy is indeed restricted to tools and frameworks used in the context of the SMAC project. This makes the taxonomy non-exhaustive when dealing with a generic smart system.

B. Taxonomy of the main MoCs used for smart system design

The second taxonomy proposed in this paper aims at being more complete, by moving the focus from tools to the underlying execution mechanism they implement, *i.e.*, the MoC. This allows to generalize the taxonomy, as a given MoC is adopted by a number of tools. At the same time, the implemented MoC is the element that, despite of the syntactic constructs, better characterizes a tool or a framework, and that facilitate to understand whether the tool matches a certain domain or not.

The resulting taxonomy is depicted in Table III, that identifies the abstraction levels (rows) and the domains (columns) of the most widespread MoCs. The lowest abstraction levels feature the continuous time MoC, that adheres more to the natural evolution of physical processes (levels *Physical* and *Device*). As the level of abstraction raises, other MoCs come into play, including the discrete event MoC, that matches well domains like digital HW and embedded SW. Note that, once again, the taxonomy in Table III reports only a subset of the possible MoCs, as it focuses on the ones derived for the tools in Table II.

So far, the taxonomies focused on single components. However, when dealing with MoCs, it is necessary to specify not only the *definition of a component* (*i.e.*, what is an actor, and how actors evolve), but also the *concurrency* and *communication mechanisms* (*i.e.*, how actors act together and exchange information) [8]. For this reason, Table III adds a further dimension to the analysis: the *Synchronization & Concurrency* column reports the MoC governing the interaction between components at any abstraction level. The adopted MoC varies across the different abstraction levels: continuous time, adopted by the lowest levels of abstraction, is replaced by discrete event at *Structural*, *Homogeneous structural* and *Functional*. Finally, the *Transactional* level bases communication on the synchronous dataflow MoC.

It is now possible to define the target abstraction level for System-level simulation of heterogeneous smart systems. The target abstracted model of the system must be *the simplest model that preserves the details to consider* (Occam's razor), *but not simpler* [8]. In the case of System-level simulation, this level must be able to preserve all the input/output events of the system that are considered "of interest" by the designer. At the same time, the abstraction must be able to remove all other details, not to burden simulation with unnecessary details. At such level, the entire system can be seen as a sequence of discrete events in the logic time. For this reason, the taxonomy in Table III adds the *Behavioral abstraction-level*, defined as the level where every component and as well the synchronization between components is modeled according to the *discrete-event* MoC.

The conclusion that can be reached analyzing the two taxonomies is that the target environment:

- must rely on *a single tool or language*, to remove synchronization and communication overhead;
- must be based on the *discrete-event MoC*, in order to preserve only details "of interest".

IV. Integration flow

Figure 2 gives an overview of the proposed methodology for enhancing smart system design. The flow consciously recalls the classical schema of PBD [15]. In fact, the proposed flow considers the *High-Level Specification* of the functionality as the *Application Space*, while the *Architectural Space* is given by the *Library of Reusable Components*. Thus, the pair composed by the specification and the library is the starting point of the proposed design flow, that exploits both top-down and bottom-up steps to reduce all the heterogeneous descriptions into homogeneous models, and to obtain a final implementation of the overall system.

The final objective is to create a virtual prototype of the system under design, expressed at the behavioral level of abstraction. The overall process is fully automated by

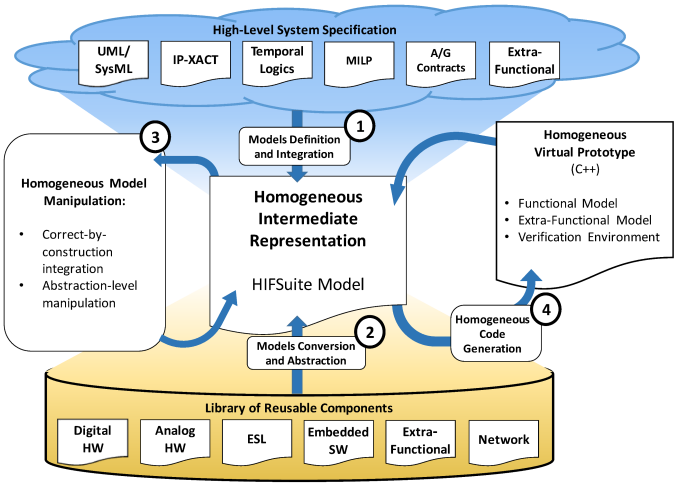

Figure 2: Overview of the proposed design flow.

using the intermediate representation and model manipulation functionalities provided by HIFSuite [19].

A. Models Definition and Integration

Requirements and high-level specifications go through the *top-down flow* (*i.e.*, Step ① in Figure 2): they are concretized with a precise semantics provided by the HIFSuite representation, and then integrated with any other model composing the intermediate representation of the system functionality. The framework supports different specification mechanisms:

- *Modeling languages*, such as UML and SysML, can be used to specify the desired functionality and tasks [20].
- *IP-XACT* permits to specify how the subcomponents of the system must interact in order to implement the functionality. A recent extension to extra-functional information permits to analyze and develop the functional behavior of the system keeping an eye on extra-functional properties of the system, that may be crucial when dealing with smart systems design [21].
- *Formal specifications and extra-functional parameters* are currently supported in terms of mathematical based-formalisms, *e.g.* Temporal Logic properties, Mixed-Integer Linear problems, and Assume-Guarantees Contracts. This allows designers to specify in mathematical terms system parameters, *e.g.*, properties orthogonal to functionality, like timing, reliability, and costs.

To support this flow, the HIFSuite front-end mechanisms and semantics are extended to manage UML/SysML models, Assume/Guarantees Contracts and IP-XACT specification for both functional and extra-functional specifications.

B. Models Conversion and Abstraction

Models within the library of reusable components are already specified in a number of languages and frameworks. It is thus necessary to remove their language-specific details, and to convert them into the intermediate representation of HIFSuite through the *bottom-up* path of the methodology (*i.e.*, Step ② in Figure 2). These automatic translations correctly map the semantics of the different formalisms and languages involved into the intermediate format syntax and semantics [17], [19]–[23].

In order to provide automation, it was necessary to extend the front-end functionalities provided by HIFSuite, originally intended only for the Digital HW domain. In particular:

- *Analog HW and MEMS* design domains have been supported by proposing a novel translation methodology [23], that reproduces the entire set of equations expressed by the original analog model of the system in the homogeneous intermediate representation.
- *Embedded SW* has been supported by implementing a front-end tool for UML/SysML state diagrams annotated with C++ instructions [20].
- The possibility to import the *IP-XACT* model of the components has been implemented to improve the reuse of System-level models and IPs.

After applying the top-down specification and bottom-up reuse, all the components of the system are represented in a homogeneous representation. Thus, they can be manipulated within a single framework.

C. Homogeneous Model Manipulation

After Steps ① and ②, components are still disconnected from each other and fully detailed. Step ③ of Figure 2 performs different manipulations on top of the homogeneous model, in order to correctly integrate components and to manipulate their levels of abstraction.

HIFSuite already provided abstraction and optimization functionalities for digital HW models [19], [22]. However some additions have been implemented to support heterogeneous smart devices. In particular, the methodology presented in [24] to automatically abstract Analog HW and MEMS models has been implemented on top of HIFSuite in a tool called *OCCAM*. The abstraction procedure starts from an analog model expressed in the HIFSuite format and a set of physical quantities (*i.e.*, "values of interest") of the system specified by the designer. Usually, values of interest are those quantities that carry behavioral information to the other components of the system (*e.g.*, input or output nodes of an electrical analog device). The abstraction procedure ensures the preservation of the values of interest, and it exploits static algebraic symbolic resolution to simplify the system. After the abstraction, the model of the component preserves only the events on the values of interest.

Integration of components is then performed by matching components interfaces (automatically extracted from the HIFSuite models) with the information given by the designer about interfaces and communication between components (*e.g.*, high-level system specifications provided as IP-XACT descriptions). Once components are integrated into the homogeneous model, the processes involved in the system are synchronized. The synchronization is delegated to a scheduling generation procedure [22], extended to support events generated by continuous-time components.

After abstraction and integration, the system is completely specified by a single homogeneous model, entirely based on

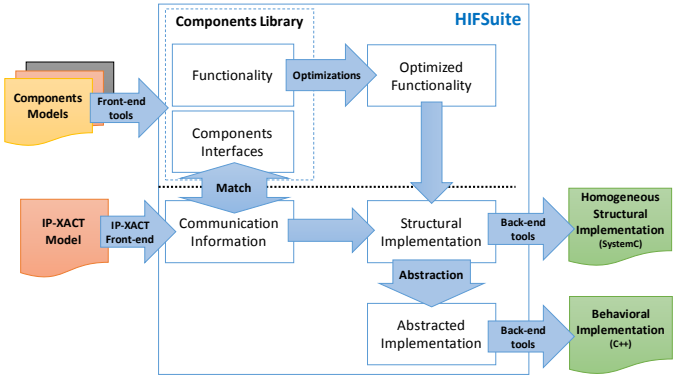

Figure 3: HIFSuite-based flow for integration and abstraction of smart device models into SystemC and C++ representations.

the discrete-event model of computation and thus specified at the *behavioral abstraction-level.*

D. Homogeneous Code Generation

One of the main characteristics of the desired simulation environment is that it relies on a single language, to remove any synchronization overhead. This is achieved by automatically generating code from the homogeneous intermediate model, expressed in the HIFSuite format.

Figure 3 recaps how components are integrated and abstracted to generate the virtual prototype of a given system. At this point, the starting descriptions have been converted to the HIFSuite intermediate format through front-ends (leftmost arrows). The homogeneous format is then handled by considering functionality and interfaces separately. Interfaces are matched with the high-level architecture specification, originally modeled through IP-XACT descriptions. This permits to determine the structure of the system, in terms of SystemC modules and connections. The functionality is then used to populate the modules, after the application of optimization steps, like [25]. This information enables HIFSuite back-end tools to generate a *homogeneous structural* implementation of the system in SystemC.

In order to improve performance, it is possible to raise the level of abstraction to *behavioral level*. To this extent, the model undergoes the abstraction procedure implemented by the framework. The final model of the system is automatically implemented in C++ by the HIFSuite back-end tools, thus constituting an efficient virtual prototype of the system.

V. Case Study: the Open-Source Smart System Test-Case

The Open-Source Smart-System Test Case (OS^3TC) represents an example Smart System, developed in the context of the SMAC European Project [16]. The OS^3TC supports all the typical domains of smart systems, and thus clearly highlights the degree of heterogeneity and the typical issues that arise at design time. The OS^3TC is available as open-source demonstrator for HIFSuite[1].

Figure 4 depicts the structure of the OS^3TC, where different colors expose the involved domains: *Digital HW* (blue),

[1]http://download.hifsuite.com/

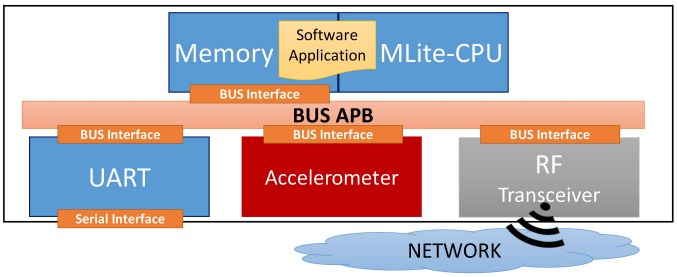

Figure 4: Overview of the OS^3TC structure. Colors highlight the different design domains involved.

Analog HW (red), *Network* peripherals (grey), *embedded SW* (yellow), and finally, *system-level interconnections* (orange, implemented as digital HW components). The OS^3TC components are:

- *MLite-CPU*: digital HW microprocessor compatible with the MIPS instruction set. The IP is originally provided as VHDL description, within the Plasma project of OpenCores;
- *Memory*: digital HW component implementing a 256KB SRAM memory, used to store the SW application as well as the data sensed and computed by the device and to handle Memory Mapped Input-Output communication with the peripherals. It is originally implemented in Verilog.
- *UART*: digital HW peripheral performing Parallel to Serial conversion. It provides a serial interface to the device, that can be used to program the entire system, or read computed data. Its open-source VHDL implementation is available in the 16550 UART component project of OpenCores.
- *Accelerometer*: analog HW peripheral, implementing a two-axis accelerometer. It is used to sense the environment, in terms of the accelerations that the platform is subjected to. The original Verilog-A description of the IP has been provided by industrial partners of the SMAC European Project.
- *RF-Transceiver and Network*: network peripheral used to transmit data over a packet-based network. The original model is developed at a high-level of abstraction, using *SCNSL*.
- *APB Bus*: main bus used to connect the CPU-memory subsystem to the peripherals. The adopted implementation complies with the ARM APB and it is modeled both in Verilog and as IP-XACT specification.
- *Communication, Interfaces and Interconnections*: internal connections and communication among the components of the platform are described at ESL using IP-XACT.
- a *Software Application* running on the CPU. The SW main loop samples and gathers data sensed by the accelerometer. The data are then transmitted as follows: the least significant byte is written on the UART, while the entire value is sent through a packet switching network by using the RF-transceiver. The SW is given as Assembly code and as the list of opcodes generated for the MLite-CPU.

The list of the constituting components highlights that the

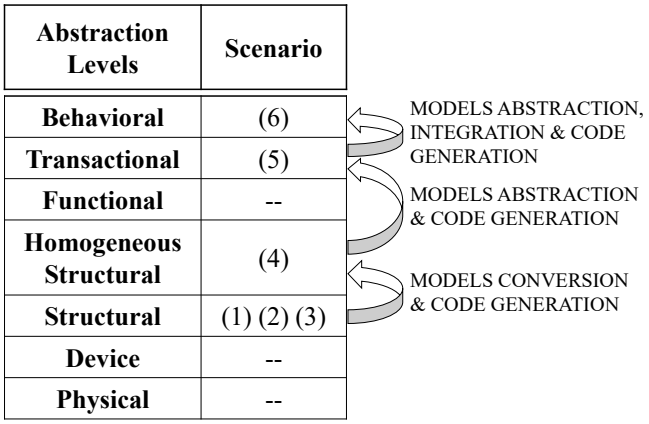

Figure 5: Mapping of the six scenarios used in the experimental section with respect to the abstraction levels of the taxonomy in Table III.

OS^3TC is subjected to a high degree of heterogeneity, in terms of both domains and abstraction levels. As such it represents a good case study for smart systems design methodologies.

VI. Experimental Results

The OS^3TC has been used to evaluate the impact of translation and abstraction on system simulation. For the experimental analysis, we generated six scenarios at different abstraction levels of the proposed taxonomy, and we compared them in terms of simulation speed.

We considered three co-simulation scenarios:

(1) The original models of the components are co-simulated by different simulators and synchronized through Keysight's SystemVue, a Ptolemy-based simulation environment [26];
(2) The original models of the components are co-simulated by different simulators and synchronized within the heterogeneous environment provided by Mentor's Questa;
(3) All the components are translated into C++, despite of the UART, that is co-simulated by using an ad-hoc simulator and synchronized with the others by using SystemVue.

The flow in this paper has then been applied to the OS^3TC to create three simulation scenarios:

(4) Components are translated into SystemC-RTL and SystemC-AMS, to be integrated and simulated within the SystemC simulation kernel;
(5) Components are abstracted into C++ descriptions, imported and integrated within SystemVue;
(6) Components are abstracted and integrated to create a unique homogeneous C++ model providing a "monolithic" simulation of the entire smart system.

Figure 5 places the scenarios in the abstraction levels of the taxonomies described in Section III. Arrows on the right of the Figure depict the transformations applied to move from one level to another (as described in Section IV). The three co-simulation scenarios are Structural models of the OS^3TC, while the three simulation scenarios express the system at three different abstraction levels. Scenario (4) is obtained by applying model conversion for each component of the system to automatically generate its SystemC model, and is thus at *Homogeneous structural level*. Components are aggregated together by interfacing the SystemC models at the Register-Transfer Level (RTL). The system model in Scenario (5) is obtained by abstracting the Homogeneous Structural model of each component to the *Transactional* level. SystemVue-compliant C++-implementations of the system components are generated and integrated by the designer within SystemVue. Finally, Scenario (6) performs further abstraction and integration to create a "monolithic" *Behavioral* discrete-event model of the OS^3TC. Code generation is exploited to create the final C++ model for simulating the entire system.

Table IV summarizes the execution time needed by the different scenarios to simulate 100 ms of system execution, with a time step of 100 ns. Column *Relative Speed-up* reports the speed-up between consecutive scenarios, *e.g.*, as a result of abstraction or transformations. Column *Total Speed-up* reports the speed-up of each scenario w.r.t. scenario (1), that is the most heterogeneous one.

A first fact that appears clear from these results is that the *number of simulators instantiated*, hence the number of co-simulation interfaces employed, heavily impacts performance. In particular, it is worth noticing that every co-simulation interface (three in the case of the first entry of the table, two in the second and only one in the third), seems to introduce around 60 seconds overhead *w.r.t.* the simulation without any co-simulation interface (fourth scenario). This corresponds to an overhead of more than 60%. As a result, the impact of interfaces and conversion layers between different tools seems highly relevant and dependent on the number of used interfaces and external tools. Thus, translating to a unique language positively impacts on the simulation time required.

Introducing abstraction together with translation (fifth entry of the table) provides the *maximum relative speed-up*. Finally, the sixth scenario merges synchronization and behaviors within a unique monolithic executable model, preserving only the events of functional interest for the designer. This provides the most optimized simulation environment available for the system and thus the best simulation performance, corresponding to the *maximum total speed-up*. These results show that the configuration identified in Section III, combining homogeneous simulation and the adoption of the discrete-event MoC, is the most effective for the design of heterogeneous smart systems.

Note that, given the same case-study, this paper leads to a there is a slight reduction of speed-up with respect to the results obtained in [1] (24.11x against 13.10x). The better performance of [1] can be easily justified by the fact that the proposed flow was not entirely automatic. The approach in [1] required indeed manual definition of the UNIVERCM models of some components, thus resulting in a time-consuming and error-prone re-modeling activity by the designer, that on the other hand allowed the application of manual optimization. In this work, instead, the process is *fully automatic* and allowed to produce the final Scenario (6) in few minutes rather than few man-days, still providing a speed-up in the same order of magnitude.

Table IV: Execution time needed by the different simulation and co-simulation scenarios considered.

Scenario			Simulation Time (s)	Relative Speed-up	Total Speed-up
Co-Simulation	(1)	Structural (SystemVue-based coordination)	278.59	-	-
	(2)	Structural (Modelsim & ELDO SPICE)	215.47	1.29x	1.29x
	(3)	Structural (SystemVue & Modelsim)	153.23	1.37x	1.82x
Simulation	(4)	Homogeneous Structural (SystemC-RTL)	97.59	1.61x	2.85x
	(5)	Transactional (C++/SystemVue)	36.32	2.69x	7.67x
	(6)	Behavioral (C++)	21.26	1.71x	13.10x

VII. Conclusions

This work presented a meet-in-the-middle approach to model and simulate heterogeneous smart devices. From a set of components belonging to different design domains and expressed in a heterogeneous set of abstraction levels, the methodology exploits automatic translation, abstraction and integration to reconcile the heterogeneous set of component models into a single homogeneous system-level model. The high-level model generation is guided by the analysis of two taxonomies of modeling and simulation technologies for heterogeneous devices. A virtual prototype of smart system device has been presented and released open-source as a demonstrator. The application of the proposed methodology showed the positive impact of the proposed solution.

References

[1] 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.

[2] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, no. 7589, pp. 144–147, feb 2016.

[3] S. J. Palmisano, "A smarter planet: the next leadership agenda," *IBM. November*, vol. 6, 2008.

[4] G. Akhras, "Smart materials and smart systems for the future," *Canadian Military Journal*, vol. 1, no. 3, pp. 25–31, 2000.

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

[6] J. A. Estefan *et al.*, "Survey of model-based systems engineering (MBSE) methodologies," *Incose MBSE Focus Group*, vol. 25, 2007.

[7] Mathworks, Matlab, "Simulink/Stateflow," mathworks.com/products/stateflow.

[8] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org Berkeley, CA, USA, 2014.

[9] V. Berman, "Standards: The P1685 IP-XACT IP Metadata Standard," *IEEE Design & Test of Computers*, vol. 23, no. 4, pp. 316–317, apr 2006, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1683721.

[10] 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.

[11] Imperas Software, "OVP - Open Virtual Platforms," www.ovpworld.org.

[12] Cadence Design Systems, "Virtual System Platform," www.cadence.com.

[13] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel *et al.*, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proc. of MODELICA*, no. 076. Linköping University Electronic Press, 2012, pp. 173–184.

[14] C. Bertsch, E. Ahle, and U. Schulmeister, "The functional mockup interface-seen from an industrial perspective," in *Proc. of MODELICA*, no. 096. Linköping University Electronic Press, 2014, pp. 27–33.

[15] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EETimes*, 2002.

[16] R. Gillon, G. Gangemi, M. Grosso, F. Fummi, and M. Poncino, "Multi-domain simulation as a foundation for the engineering of smart systems: Challenges and the SMAC vision," in *IEEE ICECS*. IEEE, dec 2014, pp. 858–861.

[17] L. Di 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.

[18] L. Di Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, and S. Vinco, "A formal support for homogeneous simulation of heterogeneous embedded systems," in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES) 2012*, 2012, pp. 211–219.

[19] N. Bombieri, G. D. 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, p. 4, 2010.

[20] M. Lora, F. Martinelli, and F. Fummi, "Hardware Synthesis from Software-oriented UML Descriptions," in *Proc. of IEEE MTV*, 2014, pp. 33–38.

[21] S. Vinco, M. Lora, E. Macii, and M. Poncino, "IP-XACT for smart systems design: extensions for the integration of functional and extra-functional models," in *Proc. of ECSI/IEEE FDL 16*, 2016, pp. 1–8.

[22] S. Vinco, V. Guarnieri, and F. Fummi, "Code Manipulation for Virtual Platform Integration," *IEEE Transactions on Computers*, doi: 10.1109/TC.2015.2500573, 2015.

[23] S. Vinco, M. Lora, and M. Zwolinski, "Conservative Behavioural Modelling in SystemC-AMS," in *Proc. of ECSI/IEEE FDL*, 2015, pp. 1–8.

[24] E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi, "Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems," in *Proc. of IEEE/ACM DATE*, 2016, pp. 1–6).

[25] N. Bombieri, F. Fummi, V. Guarnieri, F. Stefanni, and S. Vinco, "HDTLib: an efficient implementation of SystemC data types for fast simulation at different abstraction levels," *Design Automation for Embedded Systems*, vol. 16, no. 2, pp. 115–135, jul 2012.

[26] Keysight Technologies, "SystemVue," http://www.keysight.com/.