POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Layered Methodology for the Simulation of Extra-Functional Properties in Smart Systems

Original
A Layered Methodology for the Simulation of Extra-Functional Properties in Smart Systems / Vinco, Sara; Chen, Yukai; Fummi, Franco; Macii, Enrico; Poncino, Massimo. - In: IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. - ISSN 0278-0070. - ELETTRONICO. - 36:10(2017), pp. 1702-1715. [10.1109/TCAD.2017.2650980]

Availability:
This version is available at: 11583/2669873 since: 2020-02-22T11:33:44Z

Publisher:
IEEE

Published
DOI:10.1109/TCAD.2017.2650980

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)

24 April 2024

A Layered Methodology for the Simulation of Extra-Functional Properties in Smart Systems

Sara Vinco *Member, IEEE*, Yukai Chen *Member, IEEE*, Franco Fummi *Member, IEEE*, Enrico Macii *Fellow, IEEE*, Massimo Poncino *Senior Member, IEEE*

***Abstract*—Smart Systems represent a broad class of intelligent, miniaturized devices incorporating functionality like sensing, actuation, and control. In order to support these functions, they must include sophisticated and heterogeneous components, such as sensors and actuators, multiple power sources and storage devices, digital signal processing, and wireless connectivity.**

The high degree of heterogeneity typical of smart systems has a heavy impact on their design: the challenges are not in fact restricted to their functionality, but are also related to a number of extra-functional properties, including power consumption, temperature and aging. Current simulation- or model-based design approaches do not target a smart system as a whole, but rather single domains (digital, analog, power devices, etc.) or properties. This paper tries to overcome this limitation by proposing a framework for the concurrent simulation of both functionality and such extra-functional properties. The latter are modeled as different information flows, managed by dedicated "virtual buses" and formalized through the adoption of IP-XACT. SystemC, through the support of physical and continuous time modeling provided by its Analog and Mixed Signal (AMS) extension, is used to implement both functional and extra-functional models.

Experimental results show the efficiency, accuracy and modularity of the proposed approach on an example case study, in which substantial speedups with respect to standard model-based design tools go along with a very high degree of accuracy ($< 10^{-5}$%). Furthermore, the case study highlights that the proposed framework allows to easily capture at run time the mutual impact of properties, *e.g.*, in case of power and temperature.

***Index Terms*—SystemC, System on chip, Simulation, Modeling, Power modeling and estimation, Extra-functional simulation.**

I. Introduction

Smart electronic systems include heterogeneous components such as digital, analog/RF devices, sensors/actuators, and energy generation/storage devices; even the very simulation of their *functionality* represents already a significant challenge, since these diverse components imply different time scales and accuracies, and interactions among very different domains [2], [4]. However, the assessment of functionality is not the only dimension to be considered at validation time. Since these systems combine heterogeneous domains, other metrics must be considered to ensure their correct operations, such as power consumption, thermal behavior, or reliability. The challenge is therefore that of monitoring the evolution over time of other *extra-functional* properties, of which power, temperature, and reliability are three relevant examples [33].

Modeling and monitoring of these properties in the various domains in isolation is not a new problem. Power and thermal analysis, for instance, has been studied since a couple of decades in digital and analog systems. How to manage the complex interactions of these properties in heterogeneous smart systems, however, is still an open problem. Moreover, these quantities are inter-dependent in complex ways. Power consumption affects thermal and aging patterns, while temperature affects power consumption (in particular, static power in digital logic) and it is an essential parameter in any reliability or aging model. On top of everything, the functional operation of the system (*e.g.*, the duty cycle) affects all other properties. Accurately tracking the mutual influence among extra-functional properties must be done *at runtime*; unfortunately, no current design methodology can simultaneously master all extra-functional aspects of a smart system. State of the art approaches either simulate specific properties independently with ad hoc simulators [9], adopt time-consuming co-simulation approaches [6], [33], or support only subsets of the typical smart systems domains [13], [16]. These approaches are generally not very friendly for "functional" designers used to specify systems using HDLs like SystemC or SystemVerilog and that have little familiarity with the various extra-functional properties. Furthermore, the setup of co-simulation environments requires expertise and deep knowledge of all the involved tools.

The goal of this work is to fill such a gap by reducing both functionality and extra-functional properties **to a single modeling and simulation framework**. The proposed approach envisions a **multi-layer, bus-centric framework**. *Multi-layer* because it is structured hierarchically, with each property corresponding to a simulation layer, and each component implemented as a set of property-specific models. *Bus-centric* in that each property is simulated by adopting a specific "virtual bus", which conveys and elaborates property-specific information, used to derive property-specific status of the overall system. This allows reducing all layers to a common structure, easing synchronization and information exchange. These two features, coupled with the adoption of a conventional HDL as specification language and simulation backbone, ease the adoption of the proposed approach by functional designers, by providing them with an enhanced support to the traditional functional design flow. The resulting

Copyright © 2015 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

S. Vinco, Y. Chen, E. Macii, M. Poncino are with the Department of Control and Computer Engineering, Politecnico di Torino, Italy, e-mail: sara.vinco@polito.it, yukai.chen@polito.it, enrico.macii@polito.it, massimo.poncino@polito.it.

F. Fummi is with the Department of Computer Science, University of Verona, Italy, e-mail: franco.fummi@univr.it.

framework is composed of a number of property-specific models per component, managed by property-specific buses and bus managers, in charge of aggregating information and of applying control policies.
The following are the specific contributions of this work:

- Construction of a **novel multi-layer simulation infrastructure** that allows one to model a range of functional and extra-functional properties simultaneously and *in a single simulator instance*;
- Adoption of **buses as central components** for reproducing property-specific information flows;
- **Adoption of standard functional description languages**, namely, SystemC [18], [19] and IP-XACT [20], to ease formalization and automation of the proposed framework;
- Formalization of **property-specific features**, in terms of signals, semantics and information flows;
- Presentation of existing models and methodologies to **incorporate each extra-functional property**;
- **Automation of code generation**, to enhance and ease the adoption of the proposed approach.

The application of our approach to an industry-strength smart system case-study demonstrates its effectiveness with respect to model-based design tools in modeling the various properties, both in terms of accuracy (average errors on the order of 10^{-6}%) and of simulation time (speedups of at least one order of magnitude).
The paper is organized as follows. Section II reviews state-of-the-art solutions targeting the design of smart systems. Section III outlines the proposed approach. Sections IV–VI detail the languages adopted for framework implementation, and the code generation flow. Finally, Section VII applies the overall approach to the industrial smart system case-study.

II. Related Work

Various solutions have been proposed to address the simulation challenges posed by smart systems heterogeneity.
Co-simulation adopts dedicated simulators and custom models for each heterogeneous aspect of the system [2], [4]. This results in a high complexity and in a significant overhead due to the management of different simulators, *e.g.*, to synchronize simulation time and event queues. To overcome these limitations, many frameworks have been proposed to standardize the integration of tools [6], [33], through either unified APIs [6] or libraries [33]. Even if this simplifies the construction of the co-simulation environment, the designer is required to have a detailed knowledge not only of the modeled domain and tools, but also of the underlying integration framework. Furthermore, the adoption of a centralized solver imposes a unique time management approach, thus constraining both the synchronization points and the semantics of the involved tools.
Equation-based approaches, such as Modelica [13], are effective for the physical part, but they are not suitable for modeling the "cyber" part (*e.g.*, the software executed by a core) and the mutual influence of cyber and physical aspects.
Platform-based design approaches, *e.g.*, Metronomy [16], focus on the interaction of controllers with the environment to prove temporal properties of the implemented system. To achieve this result, functionality modeling relies on MetroII [11], that provides a range of Models of Computation (MoCs), differing in how time and data are managed. Communication between different MoCs relies on adapters, which introduce a significant communication overhead, and on synchronous events, thus not allowing asynchronous interactions. Moreover, MoCs are designed for describing HW-SW systems; thus, they fail at supporting extra-functional properties, *e.g.*, in case of electrical network models or of power components (e.g., batteries). Finally, the adoption of custom languages and formalisms restricts the reuse of existing IPs, and it requires a deep knowledge of the semantics of the various MoCs.
The *heterogeneous rich component* (HRC) approach proposes to separately simulate different system properties by adopting native tools [9], [22]. Mutual influence of properties is then modeled by using traces produced by one simulation as input for the next one, and the resulting information is used to validate the overall system with extra-functional contracts. Such an approach does not envision a common simulation framework or a standard interaction between properties. Furthermore, the result is a sequential adoption of tools, that is extremely limited in its ability to capture the runtime dynamics of the system and the mutual influence of properties [33].
This work builds upon the methodology presented in [43], where the power bus is used both as a voltage reference and to reproduce energy flows. However, [43] restricts modeling to the power domain. In this work we extend the basic bus-centric idea by formalizing the guidelines for modeling and concurrently simulating generic extra-functional properties.

III. A Layered Vision for Smart System Simulation

Two are the distinctive features of the proposed methodology.

Layered approach: The reconciliation of both functional and extra-functional aspects to *a single simulation infrastructure and language* is achieved by structuring the simulated system according to different views, called *layers*, each one relative to one specific property (Figure 1). This approach allows handling information related to each property independently. At the same time, the proposed approach allows to simultaneously simulate multiple layers *in a single simulator instance*, *i.e.*, a single run. This allows reproducing the mutual interactions among layers while keeping the overhead low thanks to this unified approach.
Such a layered paradigm is coupled to the selection of a unique functional language for all layers; this easies the adoption of the proposed framework by functional designers because (i) it allows using a familiar language, and (ii) it does not require a deep knowledge about the physics behind the properties, or about the property-specific tools and languages.

Bus-centric modular architecture: Each layer is constrained to a single underlying "architecture", whose central element is the *layer-specific bus*. Similarly to what happens with functional simulation, each bus carries information between components (*intra-layer communication*).
An important reason for adopting a bus-centric architecture for all layers is to enforce the legacy with the architecture used

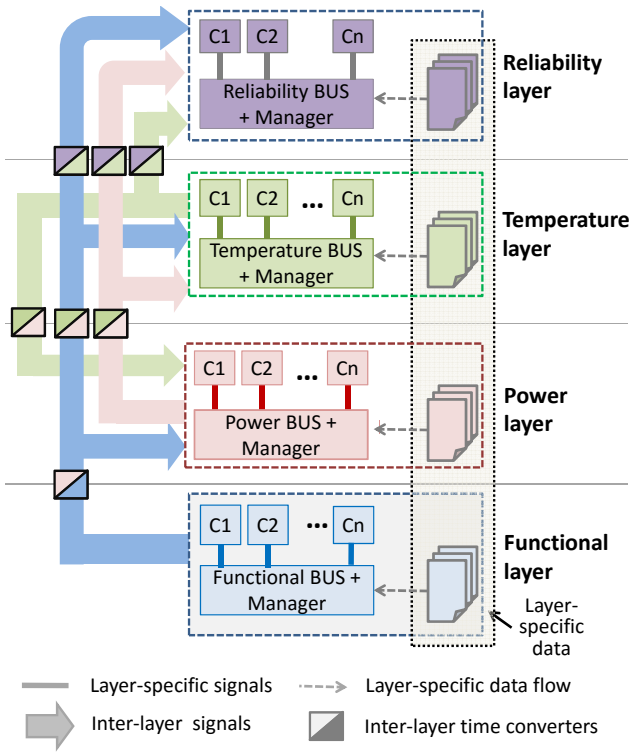


Figure 1. Layered Framework for Extra-Functional Property Simulation.

by functional simulation (the *Functional Layer* in Figure 1), for which (i) the bus-based structure reflects the logical organization among the blocks by mimicking the actual physical interconnection, and (ii) whose simulation results are used by all other layers to evaluate other properties, as shown in the figure. As any bus-based structure, the proposed bus-centric architecture is highly scalable, as adding a component at any layer simply requires sticking to the required interface.

It is worth emphasizing that the layer-specific buses do not necessary reflect the actual physical buses; they actually carry the information flow relative to the given property, that is not per se determined by the topology and the physical connections.

Figure 1 abstractly represents the system as the interconnection of *components* ($C_1, \ldots, C_n$) through layer-specific buses. Each component is associated with a model in each layer, representing the property-specific evolution of the component.

Each bus is integrated by a layer-specific *bus manager*, which aggregates information specific to each component to determine information flow and overall property-specific information. Thus, the bus supports not only the information flow but also its "arbitration" and management, *e.g.* in the form of property-specific control policies.

Depending on the actual extra-functional properties modeled and simulated, layers exchange information representing their mutual influence. This work focuses on three major extra-functional properties: **power, temperature, and reliability**, which are traditionally considered as the most relevant ones for electronic systems [33]. It is obviously possible to extend the framework to support any desired property.

A. Application to Extra-Functional Properties

Each layer is characterized by four main characteristics, corresponding to columns of Table I:

Layer-Specific Signals: These are the main property-specific signals that are tracked by the simulation engine in each layer (Figure 1). Power behavior is determined by simulating voltage and current signals; temperature is the defining characteristic of the temperature layer, while the reliability layer tracks the failure rate of system components.

Inter-Layer Signals: These signals are used to exchange information between layers. Since all layers are simulated in a single simulation run, each layer will react to changes in those inter-layer signals instantaneously. This solution provides a clear advantage with respect to the HRC approach, where properties are simulated sequentially and the mutual influence between properties is modeled through explicit traces [9]. Figure 1 exposes these inter-relations among layers: the power layer requires "workload" information from the functional layer (*e.g.*, duty cycle, switching activity); the temperature layer requires both workload and power information, while at the same time temperature information is fed back to the power layer to estimate static power. Finally, the reliability layer is affected by information produced by all other layers.

Role of Bus and Manager: The role of the bus and of the manager determines the *simulation semantics* of each layer. For instance, in the temperature layer, the semantics corresponds to a circuit-level simulation equivalent of the thermal network [36], while at reliability layer the semantics is an aggregation function to determine overall failure rate from local values [23]. The role of the manager is related to (1) possible "protocols" governing the interconnection among the components (as in functional simulation, where the bus is a physical component implementing specified rules, *e.g.*, arbitration), and (2) possible policies that govern the system evolution (*e.g.*, disabling a component based on some condition).

Layer-Specific Data/Information: These data refer to additional information essential for the simulation but not involved with the simulation semantics. As an example, when simulating temperature we need to know both the floorplan of the components (to determine heat exchange) and the three-dimensional geometrical structure of the system.

B. Handling Different Time Scales

As a matter of fact, each layer evolves according to its own "time constant", representing a realistic rate of update of the corresponding property. We call this quantity the *property-specific time scale*. For instance, while functional simulation could be accurate to the clock cycle or to a transaction (for Transaction-Level Modeling), temperature updates according to time scales in the order the tens of milliseconds. Simulating each property with the most appropriate time scale is not done just to track as much realistically as possible the true nature of that property, but also to avoid excessively fine-grain computations and to speed up simulation. Generally speaking, the scale of simulation tends to increase from the functional level upwards.

Table I
LAYER-SPECIFIC MAPPING OF CONCEPTS ONTO EXTRA-FUNCTIONAL PROPERTIES.

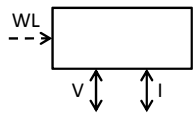
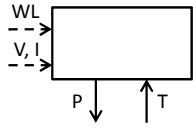
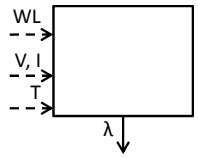
Property	Layer-Specific Signals	Inter-Layer Signals	Role of Bus & Manager	Layer-Specific Data/Information
POWER	**Voltage (V)** **Current (I)**	**Workload** (functional layer) **Temperature** (temperature layer)	The bus models the energy paths among the components (guaranteeing energy conservation) and it provides a reference voltage. The manager monitors the energy flow and it is augmented with policies (*e.g.*, use the battery or a power sources to provide energy to the loads).	Environmental data that determine the behavior of power components, *e.g.*, irradiance or vibration. These data are provided either as parameters or as traces.
TEMPERATURE	**Temperature (T)**	**Power** (power layer) **Workload** (functional layer)	The bus is in charge of conveying all technological and power information to the bus manager. The manager determines the temperature of each component by solving the electrical circuit equivalent of the overall system thermal network.	(1) *Geometrical data*: *i.e.*, floorplan; width, height and thickness of each component [32]. (2) *Technology data*: physical properties like thermal conductivity and thermal capacitance per volume [31].
RELIABILITY	**Failure rate (λ)**	**Temperature** (temperature layer) **Power** (power layer) **Workload** (functional layer)	The bus conveys the failure rates relative to individual components to the reliability bus manager. The manager determines overall system failure rate as a function of the local components failure rates.	(1) *Technology data*: material-related quantities [21]. (2) *Topology information*: *parallel connection* $\Rightarrow$ all components must fail before the system fails; *series connection* $\Rightarrow$ first failing component will cause the entire system to fail [23].

Figure 1 shows that each layer receives information from other layers after a conversion into the appropriate time scale, applied through *time converters*. When converting signals towards finer-grain time scale (*e.g.*, in case of the link between the temperature and the power layer), converters are simply time "extenders"; the finer-grain time scale is obtained by repeating a given value multiple times. Conversely, when converting signal upwards towards coarser-grain time scales, the converters must aggregate the fine-grain information, to make it available at the coarser-grain time scale. In case of *quantitative information* (*e.g.*, values of power consumption or of temperature), this is performed by implement a moving average of the downstream time scale. For instance, for simulating the power layer in a system including a battery (which is not sensitive to ns-scale variations), cycle-accurate traces from functional simulation can be averaged over 1,000 cycles. On the other hand, the functional layer typically generates *qualitative information*, such as device state or fetched instruction, that can not be averaged. In this case, the converters aggregate the qualitative information, e.g., by deriving statistics like the percentage of occurrence of each device state or fetched instruction. This information is then passed to the coarser-grain layer, to influence the corresponding evolution.

IV. MODELING OF LAYER-SPECIFIC INTERFACES

Table II summarizes the component interfaces in the various layers. Column *Component Interface* depicts the generic interface of the components at each layer, in terms of inter-layer signals (dashed arrows) and layer-specific ports (solid arrows). While the former are always inputs to a component (which "uses" that information), the direction of layer-specific port depends on the specific property. For the power layer, Table II shows the most generic interface in which voltage and current are bidirectional ports; this would be the case, e.g., of an energy storage devices in which power can be both stored and extracted. In general, components that consume power use the signals to "declare" their power demand. Thus, the (V,I) signals are outputs to the component. On the other hand, components generating power receive the power demand of the remainder of the system, and thus their (V,I) signals have the opposite direction. Temperature, as will be detailed in Section V-B, is computed in a centralized fashion by solving an equivalent electrical network inside the temperature bus; therefore components send their power consumption to the bus, and receive back the computed temperature value. Reliability, conversely, is estimated by each component autonomously, and is then aggregated in a cumulative system status by the reliability bus. Finally, the last column of Table II describes system connections and necessary auxiliary data.

Table II
INTERFACE DEFINITION FOR EXTRA-FUNCTIONAL LAYERS.

Property	Component Interface	Design Connections and Description
POWER		Components are connected to the power bus, and special components (called *converters*) may be needed to maintain compatibility of voltage levels. The IP-XACT design description reflects this topology.
TEMPERATURE		All components are connected to the temperature bus, as described by the IP-XACT design description. This does not reflect the physical contiguity between components (*i.e.*, the floorplan), that is provided through a specific configuration file.
RELIABILITY		The IP-XACT design description connects each component to the reliability bus. A configuration file describes the relationship between components: whether they are in parallel (all must fail for the system to fail) or else in series (failure of anyone causes overall system failure) [23].

```
1.  <component>
2.    <vendor>user</vendor>
3.    <library>multilayer</library>
4.    <name>core</name>
5.    <version>1.0</version>
6.    <layer>power</layer>
7.    <ports>
8.      <port>
9.        <name>V</name>
10.       <direction> out </direction>
11.         <value unit="volt" prefix="">1.0</value>
12.       </port>
13.       <port>
14.         <name>I</name>
15.         <direction> out </direction>
16.         <value unit="ampere" prefix= "milli">0.1</value>
17.       </port>
18.   </ports>
19.   <powerExtension>
20.     <componentRole>load</componentRole>
21.   </powerExtension>
22. </component>
```

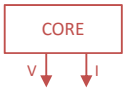


Figure 2. Example of IP-XACT *component description* for the power layer of the example system.

A. Basic Technology: IP-XACT

In order to ease formalization and automation of the proposed framework and to make its applicability as much general as possible, interfaces are described using IP-XACT [20], an XML format for describing interfaces of digital IPs and systems. IP-XACT was chosen because it is the de-facto standard for functional interface specification and it also provides mechanisms for its extension, thus allowing to add the support for extra-functional domains. IP-XACT relies on an explicit bus-based architectural template, thus reflecting the bus-centric structure of the layers in our approach.

IP-XACT supports three main description schemas. A *component description* essentially contains the interface of an IP, provided as a list of ports. A *design description* represents the instances of components in a system and the interconnection between them. A *design configuration description* stores additional configuration information that can be used at later design stages, *e.g.*, by tool-chains.

B. Modeling Extra-Functional Interfaces with IP-XACT

IP-XACT descriptions are used to model property-specific component interfaces, intra-layer connection of components, and layer-specific data. The rest of this section describes the technical details and the required extensions to the standard.

1) IP-XACT Component Descriptions: Each system component is provided with one IP-XACT component description per layer, describing the layer-specific interface. Figure 2 shows an excerpt of IP-XACT component description at the power layer for the core used as our example of a "system".

Component descriptions use the same identification tags (*i.e.*, component name, library, version and vendor, lines 2–5) [20]. An additional custom tag (`<layer>`, Line 6) specifies the specific property. After this sort of header, the component description lists the layer-specific ports of the component (the solid arrows in the blocks of Table II).

These *extra-functional ports* cannot be cast to integers or bit vectors, as they represent quantities that model a continuous physical evolution. Thus, they integrate standard IP-XACT tags (e.g., for port name and direction) with the extension defined for analog and mixed-signal modeling in [1]. Ports are associated with a default value, that is annotated with the relative unit of measure (*e.g.*, Volt or Ampere) and a prefix (*e.g.*, kilo and milli). Lines 7–18 exemplify this for the `V` and `I` ports of the core.

Power layer descriptions require additional tags to specify the role of each component. This is modeled through the `<powerExtension>` tag (lines 17–19) which embeds in turn the `<componentRole>` tag. *E.g.*, the core exemplified in Figure 2 is a component that *consumes* power, which we can generically call a *load*.

Furthermore, some components might need to have control ports (e.g., a "standby" port for power management) and/or status ports (*e.g.*, to track the state of charge of an energy storage devices). These ports are listed together with the layer-specific signals, but they are not listed in Table II, since they are not featured by all types of power components.

Component descriptions relative to other layers have the identical structure of the example of Figure 2, only with port names and the relative units differing. Two tags are however specific to the power layer: the `<powerExtension>` one (only power components have multiple "roles"), and the status/control signals.

```
1.  <design>
2.    <vendor>user</vendor>
3.    <library>multilayer</library>
4.    <name>temperature_connections</name>
5.    <version>1.0</version>
6.    <layer>temperature</layer>
7.    <componentInstance>
8.      <instanceName> core </instanceName>
9.      <componentRef vendor ="multilayer" library="modules"
                name="core" versione="1.0"/>
10.   </componentInstance>
11.   <componentInstance>
12.     <instanceName>temperature_bus</instanceName>
13.     <componentRef vendor ="multilayer" library="buses"
                name="temperature_bus" version="1.0"/>
14.   </componentInstance>
15.   <adHocConnection>
16.     <name>Pcore</name>
17.     <portReference component="core" port="P"/>
18.     <portReference component ="temperature_bus" port ="P"/>
19.   </adHocConnection>
20.   <adHocConnection>
21.     <name>Tcore</name>
22.     <portReference component="core" port="T"/>
23.     <portReference component ="temperature_bus" port ="T"/>
24.   </adHocConnection>
25. </design>
```

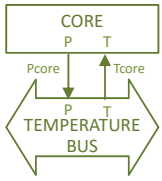


Figure 3. Example of IP-XACT *design description* for the temperature layer of the example system and connection to the temperature bus.

2) IP-XACT Design Descriptions: The overall system consist of one IP-XACT design description per layer, defining the connections between the layer-specific ports of components. Figure 3 shows an example for the temperature layer. Similar considerations apply to the other layers.

The connections modeled in the IP-XACT design descriptions reflect the content of the last column of Table II. In the power layer, components are not directly connected to the power bus, as converters may be necessary for voltage compatibility. The

design description is thus adherent to the physical connections. Conversely, temperature and reliability layers are just conceptual and not linked to the physical structure of the system. In such IP-XACT design descriptions, components are directly connected to the property-specific bus as in Figure 3.

Design descriptions are very similar for any layer; the latter is identified by `<layer>` tag (lines 2–6). Each description is nothing but an instantiation of the various components through a series of `<componentInstance>` tags, each pointing to the corresponding IP-XACT component description. Lines 7–14 of Figure 3 show the tags necessary to instantiate the core and the temperature bus in our working example.

The various instances are then bound to ports with the `<adHocConnection>` tag, by referencing ports through their name and the name of the parent component instance. In the example lines 15–24 represent the binding between the ports of the core and of the temperature bus: the power ports (lines 15–19) and the temperature ports (lines 20–24).

3) IP-XACT Design Configuration Descriptions: IP-XACT design configuration descriptions are used to model layer-specific data. The IP-XACT configuration description explicitly references the corresponding design description with the `<designRef>` tag (similar to the `<componentRef>` tag in Figure 3). The layer-specific data, *e.g.*, material characteristics, are then encapsulated by the `<configurableElementValue>` tags and organized as XML sub-trees.

V. Modeling of Extra-Functional Behavior

Different extra-functional properties determine different models for the components and different policies to be implemented by the property-specific buses. This section provides an overview of state-of-the-art models (see Column *Model Types* of Table III), since understanding their typical characteristics is essential to determine the actual implementation of the overall framework. However, note that the focus of this work is on the feasibility of the proposed framework. Components may be implemented with the designer's favorite models; the only restriction imposed are the property-specific interfaces, detailed in section IV.

It is worth emphasizing the we do not propose new power, temperature, or reliability models, but rather how to reconcile existing state-of-the-art models in the layered, bus-centric framework of Figure 1. Therefore, the effort in building such models and collecting necessary technology information is the same that is required for their use in any simulation framework alternative to the proposed one.

A. Power Layer

Power models in this context can be classified into *functional* and *circuit-level* models. Functional models implement the behavior of the component by means of a function (*e.g.*, an equation, an algorithm, or even a simple waveform over time). Examples of functional models are, for instance, power state machines to model power consumption of power-manageable loads [10], or analytical equations to model the discharge time of a battery [28]. Conversely, circuit-equivalent models emulate the behavior of a component through an equivalent electrical circuit. In literature, a variety of circuit models exists for the various elements, *e.g.*, batteries [27], converters [8] and power sources [3].

The bus manager in this layer implements control policies that monitor the power production/demand of components, enable power source and/or energy storage devices, and control the power state of the loads [43].

Table III
Model Types and Semantics.

Property	Model Types	Semantics
Power	*Functional models*: analytical compact models (*e.g.*, a function [28], or a power-state machine [10]).*Circuit-level models*: equivalent electrical circuits [27].	TDF (for functional models) / ELN (for circuit models).
Temperature	Models use a *RC-circuit* equivalent [36]. Temperature is estimated through a centralized model implemented in the bus manager. Components simply forward power information to the bus.	ELN for the RC circuit models (bus), TDF for component models.
Reliability	An expression of *failure rate* in terms of stress paramters and of material-specific parameters [34].	TDF.

B. Temperature Layer

Thermal simulators are typically based on *RC circuits* solvers [7], [36]. An equivalent RC circuit represents the heat flows across the system (*i.e.*, both heat propagation across all the multiple package layers and heat flow across adjacent components). The temperature of each component is affected not only by its working conditions (through its power consumption) and by the technology (*i.e.*, material coefficients [31]), but also by the spatial arrangement of the components (that can be derived with high-level floor-planning tools [32]). For this reason, temperature does not fit as nicely as power and reliability into the bus-based approach: we can not build separate temperature models for all components and simply combine their output at the bus level, as done for other properties. However, it is possible to accommodate temperature into the conceptual bus-based template by moving the RC solver computation inside the bus manager. This explains the direction of ports in Table II: the temperature model of each component simply forwards the necessary power demand values to the temperature bus (output port `P`), and it receives in input the temperature values, as computed by the centralized RC circuit (input port `T`).

C. Reliability Layer

As already discussed, in this work we use a "compact" measure of reliability (which is a quite general term) and track the *failure rate* of a system, usually denoted with λ, defined as the frequency with which a component (or the whole system) fails, expressed in failures per unit of time.

Many approaches have been proposed in the literature to estimate the failure rate (or related quantities such as the mean time to failure – MTTF) of a system [22], [34]. Most models have an analytical structure, *i.e.*, the failure rate of the component is expressed as a function of (i) physical stress

aspects (*i.e.*, activity) and (ii) of material-specific coefficients (available in the literature [21]). [22], [34] showed that reliability models can be adopted to estimate instantaneous values for the failure rate, depending on the runtime system configuration and on the evolution of other extra-functional properties. This allows to compare the failure rate under stress conditions with respect to idle times, and to determine the impact of a number of factors, including operating voltage, temperature, package configuration and functional duty cycle. These analyses can be used for an effective reliability-aware design space exploration, even at early stages of the design flow.

Different models target different reliability mechanisms, including negative bias temperature instability (NBTI) and time dependent dielectric breakdown (TDDB). Each component has a failure rate model for each reliability mechanism of interest, which are then summed to determine overall failure rate of the component [34].

The overall system failure rate (or MTTF) depends on the actual relation among the components. This information is typically modeled by Reliability Block Diagrams (RBDs), a graphical representation of how the components of the system are reliability-wise connected, which may differ from how the components are physically connected. The reliability bus receives the failure rates of the individual components and determines overall system reliability using the information on system topology provided as an RBD, plus the power, activity, and temperature information from the downstream layers.

VI. Implementation of Extra-Functional Models

The reference language adopted for model implementation is SystemC-AMS, an extension of SystemC for modeling and simulation of analog/mixed-signal functional subsystems [19].

A. Basic Technology: SystemC-AMS

SystemC-AMS provides three different MoCs: (1) *Timed Data-Flow* (TDF), which models discrete time, statically scheduled processes, (2) *Linear Signal Flow* (LSF), which supports continuous-time, non-conservative behaviors, and (3) *Electrical Linear Network* (ELN), which models electrical networks through the instantiation of predefined primitives, such as resistors or capacitors.

SystemC-AMS is the reference language for the proposed framework for a number of reasons. The presence of *multiple MoCs* allows covering a wide range of domains using a single language. Thus, models can be implemented with the most suitable MoC (as shown in Column *Semantics* of Table III). SystemC-AMS natively provides converters between MoCs and a common simulation kernel; it is therefore possible to **simulate different MoCs simultaneously**, still guaranteeing correctness and hiding synchronization details, and to simulate models with different levels of accuracy in a single simulation run. SystemC-AMS is also *modular*, in that it separates the definition of interface and implementation. This allows to compare different models, even implemented at different MoCs, and to decouple the MoC adopted for interface and implementation. Finally, SystemC-AMS is a *standard* language, thus easily extensible and free from compatibility and reuse issues, typical of proprietary tools.

```
1.  SC_MODULE (example_component){
2.  public:
3.      // functional interface (omitted)                    ①
4.      sc_in< bool > clk, reset_in;
5.      ...
6.      // power interface
7.      sca_tdf::sca_de::sca_out< double > V, I;             ②

8.      // temperature interface
9.      sca_tdf::sca_de::sca_out< double > P;
10.     sca_tdf::sca_de::sca_in< double > T;

11.     // reliability interface
12.     sca_tdf::sca_de::sca_out< double > lambda;

13. private:
14.     // inter-layer signals                                  ③
15.     sca_tdf::sca_signal< int > STATUS_P, STATUS_R;

16.     sca_tdf::sca_signal < double > VOLT_T, CURR_T, TEMP_P,
                                       VOLT_R, CURR_R, TEMP_R;

17.     functional_model *fm;                                  ④
18.     power_model *pm;
19.     temperature_model *tm;
20.     reliab_model *rm;

21. public:
22.     example_component(sc_core::sc_module_name name_){
23.         fm = new functional_model("fm");
24.         pm = new power_model("pm");
25.         tm = new temperature_model("tm");
26.         rm = new reliability_model("rm");

27.         // binding to interface ports                      ⑤
28.         fm->clk(clk);              fm->reset_in(reset_in); ...
29.         pm->V(V);                  pm->I(I);
30.         tm->P(P);                  tm->T(T);
31.         pm->lambda(lambda);

32.         // binding to inter-layer signals                  ⑥
33.         fm->status_p(status_p);    fm->status_r(status_r);
34.         pm->status_p(status_p);    rm->status_r(status_r);

35.         pm->volt_t(volt_t);        pm->curr_t(curr_t);
36.         tm->volt_t(volt_t);        tm->curr_t(curr_t);
37.         ...
38. }   };
```

Figure 4. SystemC code generated for a component implementing all extra-functional layers, and including the instantiation of the property-specific SystemC-AMS modules.

B. SystemC-AMS code organization

The flexibility of SystemC-AMS allows one to encapsulate all property-specific models of a component in a single SystemC-AMS module: the interface collects all the property-specific interfaces of the component, while the body includes the implementation of the property-specific models. This is realized by declaring each module as a SystemC `SC_MODULE`, without resorting to a specific SystemC-AMS MoC. This leaves maximum flexibility in the choice of the suitable MoC for the implementation, thus capturing different properties with the most suitable MoC and allowing to separate different concerns. Figure 4 shows an example of a component implementation, that is used in the remainder of this section as a reference.

1) Interface construction: The interface of each component is built by collecting all the functional and extra-functional ports from its IP-XACT component descriptions (blocks ① and ②). The interface, all system connections and the simulation backbone are implemented by adopting the *TDF MoC*. Indeed, the execution semantics of TDF accelerates simulation by defining a static schedule, and thus enforces an efficient interaction between components. This is realized by declaring

temperature_model.h

```
1.  SCA_TDF_MODULE (temperature_model){
2.  public:
3.      // temperature interface
4.      sca_tdf::sca_de::sca_out< double > P;        ①
5.      sca_tdf::sca_de::sca_in< double > T;

6.      // ports for inter-layer communication        ②
7.      sca_tdf::sca_de::sca_in< double > volt_t, curr_t;
8.      sca_tdf::sca_de::sca_out< double > temp_p, temp_r;
9.
10.     temperature_model(sc_core::sc_module_name name_){}
11. private:
12.     void set_attributes();
13.     void processing();
14.     double accumulate, temperature;
15.     int counter;
16.     double estimate_power(double, double);
17. };
```

temperature_model.cpp

```
18. #include «temperature_model.h"
19. void temperature_model::set_attributes(){
20.     P.set_timestep(TEMPERATURE_TIMESCALE);       ③
21.     volt_t.set_rate(TEMPERATURE_TS/POWER_TS);
22.     curr_t.set_rate(TEMPERATURE_TS/POWER_TS);
23. }

24. void temperature_model::processing(){            ④
25.     temperature = T.read();
26.     P.write( estimate_power(volt_t.read(), curr_t.read()) );

27.     accumulate += temperature;                   ⑤
28.     temp_p .write(temperature);
29.     counter ++;
30.     if(counter == ratio_r){
31.         temp_r.write(accumulate/ratio_r);
32.         counter = 0;   accumulate = 0.0;
33.     }
34. }
35. double estimate_power(double voltage, double curr){...}
```

Figure 5. Example of SystemC-AMS module implementing the temperature module for component in Figure 4. The code includes model implementation and the timescale conversion with respect to the power and reliability layers.

ports as TDF ports (block ②), and signals as TDF signals (block ③).

2) Body implementation: Layer-specific models of each component are implemented as separate SystemC modules, to encapsulate each property and to enable the execution of each model at its specific property-specific execution rate. Figure 5 outlines the implementation of a simple temperature module, used as reference in the remainder of this section.

The interface of layer-specific modules consists of the layer-specific ports, bound to the external interface of the component, plus extra ports used for inter-layer communication. In Figure 5, the temperature-specific ports (*i.e.*, `P` and `T`, block ①) are integrated by the inter-layer ports `volt_t` and `curr_t`, feeding the corresponding information from the power layer, and `temp_p` and `temp_r`, used to propagate the calculated temperature to the power and reliability layers, respectively (block ②).

As anticipated in the previous section, TDF is used whenever possible. However, the kind of instantiated SystemC module and the MoC adopted for the implementation of the behavior strictly depends on the underlying property-specific model.

Functional and analytical models of extra-functional properties are implemented as TDF modules (`SCA_TDF_MODULE`), to exploit the efficient scheduling of TDF. This is the case of Figure 5. Component evolution is handled by the `processing()` function, that encapsulates the C++ implementation of the model (block ④). The `set_attributes()` primitive of TDF is then used to associate a suitable timestep to the output ports (block ③). This allows associating each layer-specific module to a timestep that is appropriate for the layer (line 20).

Circuit-equivalent models are implemented as standard SystemC modules (`SC_MODULE`) encapsulating the instantiation of ELN primitives, used to map the circuit elements. Native ELN-to-TDF converters are used to convert signals between the ELN MoC and the TDF interface.

Functional behavior is handled as in traditional functional simulation. Functionality fits nicely to the TDF MoC, where the timescale corresponds to the length of the clock period. However, this is convenient only in case of components defined from scratch. Things change whenever the functional implementation is already available in any HDL. Adapting a HDL functional description to TDF would indeed require a massive modification of the starting model and in the scheduling strategy, as it would be necessary to build a static scheduling version of the component functionality [42]. To avoid this effort, the functional model is implemented as a standard SystemC module, obtained through automatic code generation and implemented at the desired abstraction level (*i.e.*, RTL or TLM) [5], [12], [15]. To allow seamless integration of the functionality, also functional ports are mapped to standard SystemC ports, and conversion to TDF occurs via native converters of SystemC-AMS (block ① of Figure 4). Note that functionality processes may be sensitive also to inter-layer ports, thus enhancing the functionality with sensitivity with respect to extra-functional properties.

Note that the complexity of the adopted models does not interfere with the underlying simulation semantics, thanks to the cycle-based semantics of SystemC-AMS [40]. Adopting more complex models impacts on the simulation time (*i.e.*, how long it takes to simulate one time step), but it does not interfere with the management of simulated time nor it introduces time misalignments.

3) Timescale management: Inter-layer communication is naturally enabled by the encapsulation of all property-specific modules in a single SystemC module. Inter-layer signals are implemented as internal TDF signals, binding the inter-layer ports of the property-specific modules (block ② in Figure 5). The *conversion between different time scales* is implemented in the property-specific module by updating each signal. Conversion towards a finer-grain scale is straightforward: signals preserve their value until explicitly updated. Thus, the value is preserved for all the finer-grain ticks. This happens, *e.g.*, with signal `temp_p`, that propagates the estimated temperature to the power layer, that is activated more frequently (line 28 of Figure 5). On the other hand, conversion towards a coarser-grain timescale must be explicitly implemented, by either accumulating the value assumed over time and implementing a moving average (for quantitative information), or by deriving a probabilistic measure (for qualitative information). An example of moving average is implemented in block ⑤ for the inter-layer signal `temp_r`, propagating the estimated temperature to the reliability layer.

Whenever a module receives data originated by a finer-grain

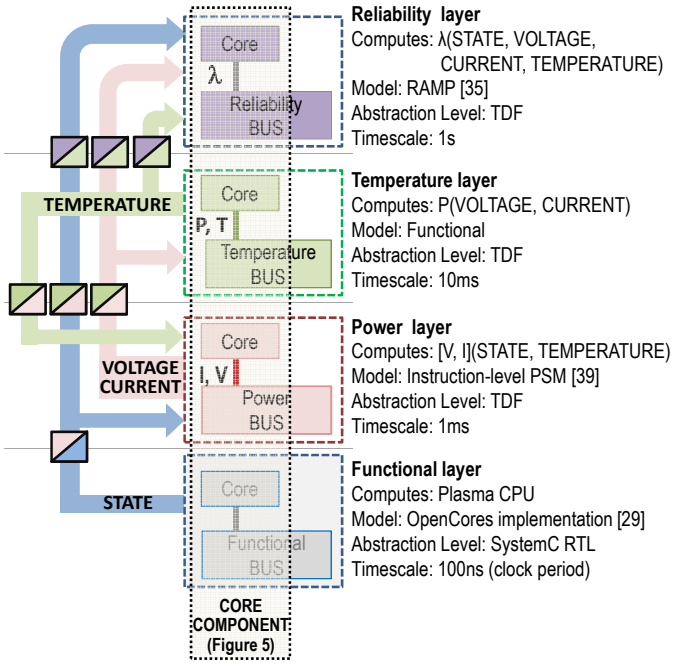


Figure 6. The Multi-Layer Approach Applied to a Single Core Example.

scale, it is necessary to explicit the time-scale ratio to ensure correct scheduling of the TDF modules. This happens in lines 21–22, where the ratio is made explicit for the inter-layer signals `volt_t` and `curr_t`, originated by the power layer.

C. A View of the Framework on the Working Example

Figure 6 pictorially summarizes the technical details of the methodology on our working example consisting of a single processor core. The left-hand side of Figure 6 highlights that the core supports all functional and extra-functional layers, together with the necessary inter-layer signals. The right-hand side details the implemented property-specific behavior, in terms of reference models [29], [35], [39] and of characteristics of the SystemC/SystemC-AMS implementation. Note that each extra-functional model corresponds to a SystemC-AMS module implemented as in Figure 5, while the functionality module is generated via automatic code generation from an existing VHDL implementation by adopting [12]. The vertical box encapsulates all modules of the core, and it corresponds to an instance of the SystemC code outlined in Figure 4.

D. Automatic SystemC-AMS Code Generation

Our framework is enhanced by two tools that implement a semi-automatic generation of the SystemC-AMS code for the overall system.

The *System Architecture Builder* is based on the XML parser of [41] and, based on the IP-XACT component descriptions of each component, builds a SystemC module for each component, together with the corresponding layer-specific modules (as presented in Section VI-B). The layer-specific IP-XACT design descriptions are then used to generate a top-level file, that instantiates all components and that binds their ports by reflecting the IP-XACT connections. Finally, IP-XACT design configuration files are used to populate buses and components with layer-specific data, declared as constants.

The *System Implementation Builder* populates the SystemC-AMS skeletons with property-specific models. This step is currently performed semi-manually by the designer, by implementing available suitable models in SystemC-AMS. Note that automatic generation approaches can be used whenever possible to automatically generate the SystemC-AMS implementation of models [7], [27]. It is worth emphasizing that the flow supports the concept of a *model library*, that includes property-specific models built from scratch or already developed. The only constraint for library models is to stick to the interfaces specified for the various components.

VII. Experimental Results

In this section we demonstrate the effectiveness of the proposed framework on an example industry-strength smart system [14], which includes the digital core used as a working example in Section VI-C, a 256KB SRAM memory, and a bidirectional MEMS accelerometer. Information is exchanged with the surrounding environment through a Radio Frequency (RF) transceiver and a UART interface [24]. All components are connected by an AMBA APB bus. The system is powered by a hybrid power supply consisting of a 700μAh rechargeable thin-film EFL700A39 battery by STMicroelectronics [37] and a Panasonic 0.33F double-layer capacitor [25], each one connected to the power bus through individual DC-DC converters, namely the STLQ015 by STMicroelectronics [38].

The system alternates data acquisition from the accelerometer, processing of these data, and their transmission. Specifically, the core reads acceleration values and it transmits the least significant byte to the UART module, and the acceleration value to the RF Transceiver for packet generation and transmission. It operates with a 66% duty cycle (1/3 idle, 2/3 active).

A. System Organization and IP-XACT descriptions

Figure 7 outlines the smart system in terms of components, modeled layers and connections. All properties are supported, including functionality, power, temperature and reliability. Note that the power layer includes all functional components, together with the power suppliers (*i.e.*, battery and supercapacitor) and the DC-DC converters. On the other hand, the temperature and reliability layers include only the core, the SRAM memory, and the UART and RF transceivers, which can be consistently simulated.

All extra-functional interfaces have been modeled through IP-XACT. Table IV reports the number of lines of XML code necessary for each layer, broken down into lines required for component descriptions, design descriptions and design configuration descriptions, respectively. The IP-XACT descriptions are then used to generate the SystemC simulation skeleton. Code generation took 6.81s.

B. Adopted Layer-Specific Models

Functional Layer: Functional models of the core, the SRAM memory, the accelerometer, and the transceivers (RF and

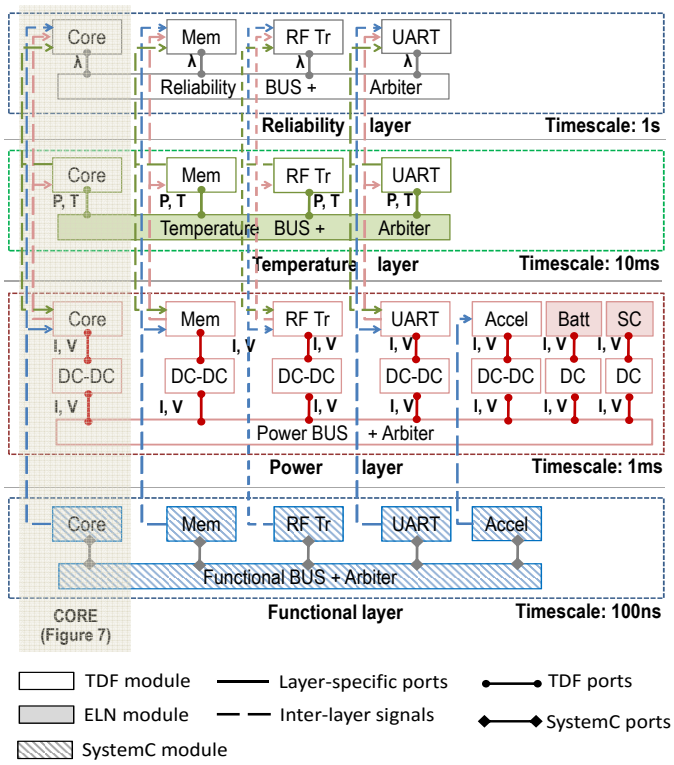


Figure 7. Layered view of the smart system adopted as case study.

Table IV
CHARACTERISTICS OF THE IP-XACT DESCRIPTIONS GENERATED FOR THE SMART SYSTEM ADOPTED AS CASE STUDY.

Layer	IP-XACT description			
	Component		Design	Configuration
	(#)	(overall loc)	(loc)	(loc)
Power	15	1,210	743	83
Temperature	5	611	625	177
Reliability	5	126	625	143

UART), were initially available in Verilog or VHDL, and thus required a conversion to SystemC RTL through automatic tools [12]. The functional bus is implemented as a SystemC RTL AMBA APB bus that performs data arbitration.

Power layer: The battery and the capacitor are represented with circuit models [27], [30]. The various DC-DC converters are modeled by expressing their conversion efficiency as a function of the difference between input voltage and output voltage, which is reasonable for LDO converters [26].

The other (functional) components are modeled through power state machines (PSMs) generated as proposed in [10] and characterized by a strict dependence on the functionality:

- the core implements an instruction-level model similar to [39]: load/store instructions require 2mW, arithmetic instructions 1mW, branches 0.4mW and NOPs 0.1mW;
- the SRAM memory consumes 50μW when idle, and 1.5mW when reading or writing;
- the UART power consumes *i.e.*, 9.98μW when idle and 1.43mW when transmitting or receiving;
- the accelerometer has a constant demand of 0.26mW;

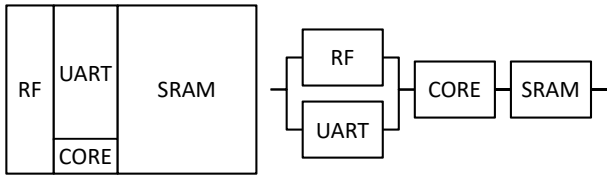


Figure 8. Floorplan (left) and RBD (right) adopted for the functional components, *i.e.*, digital processor, SRAM memory, and transceivers.

- the power consumption of the RF transceiver is low in standby mode (0.1mW), and it increases when transmitting or receiving (10 mW).

The above values are referred to an ambient temperature of 300K, and are based on data for a similar implementation [44]. Power models include the dependence of static power on temperature using the model of [17].

The power bus has a reference voltage of 1.3V. The bus manager handles the energy flows by implementing a simple threshold-based policy: current demands higher than 0.23mA are serviced by the supercapacitor, while lower demands are supplied by the battery. This policy allows to meet the energy demand of the functional components, both in terms of power density and of response to high demand peaks.

Temperature Layer: The heart of the temperature layer is the bus manager, that computes components temperatures by solving an RC-equivalent of the thermal network. Notice that only "electronic" components are considered for this analysis (*i.e.*, core, SRAM, and transceivers). A single-chip implementation is assumed, and data about floorplan and technology implementation are taken from [44]. The resulting floorplan is shown on the left-hand side of Figure 8. The SystemC-AMS implementation of the RC-circuit is generated automatically as presented in [7], and it uses the provided layer-specific data. Models implemented by the components other than the bus consists of a simple wrapper that forwards power consumption values from the power layer to the power layer, and receives updated temperature values from the temperature bus.

Reliability Layer: The reliability models used here rely on RAMP [34], and are implemented by adopting the open source library available at [35]. We currently consider two reliability mechanisms, *i.e.*, NBTI and TDDB. Since RAMP models depend on various parameters relative to workload (*e.g.*, duty cycle), temperature and power consumption, the reliability layer represents a case in which the methodology fully exercises the inter-dependencies among layers.

As for temperature, only the electronic components are considered for this analysis. We assume the components to form a topology described by the RBD on the right-hand side of Figure 8: the RF transceiver and the UART form a parallel connection (*i.e.*, the system fails only if both fail) that is connected in series with the other components.

Inter-layer signals: Dashed lines in Figure 7 represent the inter-layer communication flows. Inter-layer signals are used for all functional components, to represent the mutual effect of functionality and extra-functional properties.

The adopted inter-layer signals reflect the ones adopted for the core as shown in Figure 6 and extend them to all other

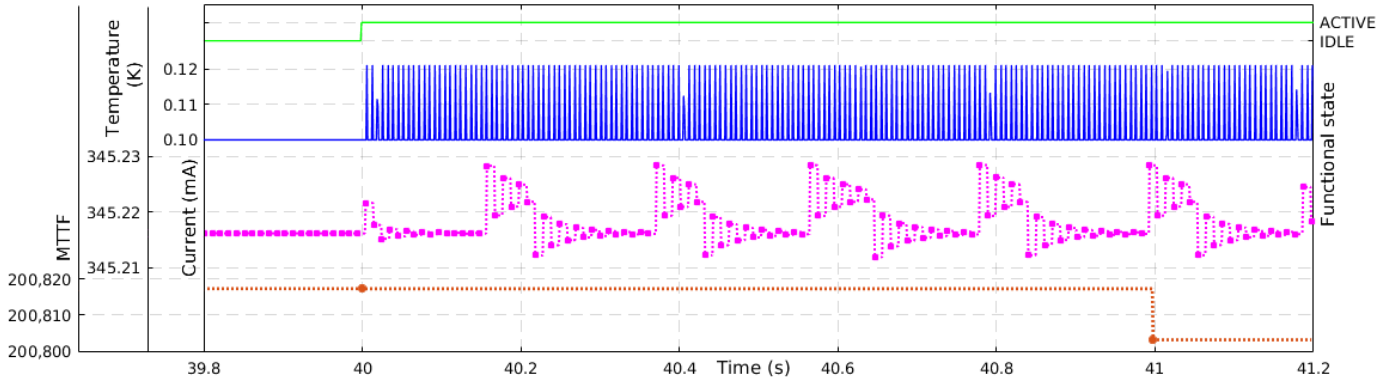


Figure 9. Excerpt of 1.4s of multi-layer simulation for the core. From top to bottom: functional state, current, temperature, and MTTF.

components. Voltage and current, estimated at power layer, are necessary parameters for temperature estimation and for reliability modeling. Temperature affects both the reliability models and the power models [17]. Finally, functional state is shared with the power layer (as it determines the evolution of the PSMs) and with the reliability layer, whose models take into account the component duty cycle. Since functional state is a qualitative information, it is passed to the other layers as a percentage of occurrence of each functional state over one time step of the coarser-grain layers.

C. SystemC-AMS Implementation Details

The color of each block in Figure 7 determines the MoC adopted for its implementation. White blocks are for TDF, the most used MoC across all layers. Functionality is implemented as SystemC blocks (slanted), while circuit models (solid) require an ELN implementation (*i.e.*, battery and supercapacitor models as well as the RC circuit implemented by the temperature bus).

Connectors show the bindings between components in the final system. Solid lines represent connections between layer-specific ports. Connector ends highlight the *port type*: round for TDF ports (mostly used), squared for SystemC ones (used to ease integration of the functionality).

Dashed lines represent *inter-layer signals*, which introduce additional ports on the interface of the involved modules. The arrow-shaped ends of dashed lines indicate the direction of information flows, *i.e.*, the producer and the consumer(s) of each signal. Note that the timescale converters are not made explicit, since they are embedded inside of the producer module (as explained in Section VI-B3).

D. Choice of Layer-Dependent Timescales

The timescale for the functional layer is dictated by the clock cycle, which is set to 100ns in our design. The timescales for the extra-functional layers reflect the "time constants" of the modeled property and their choice is driven by common sense. We chose 1ms for power, 10ms for temperature and 1s for reliability. For the latter, a timestep of 1s is probably too fine-grain for practical uses; however, it is generally not convenient to use too coarse timescales due to the characteristics of the TDF MoC, as it will be described in Section VII-H.

E. Simulation Results

Experiments have been run on a 64-bit server with 8 3.40 GHz cores and 16GB of RAM, and running Kubuntu 14.04 Linux OS. The SystemC versions are SystemC 2.3.1 and SystemC-AMS 2.0 (alpha1) [18], [19].

Using the above described workload with 66% duty cycle, we concurrently simulated functionality and the extra-functional properties. The estimated system lifetime (*i.e.*, availability of charge in the battery and the supercap) is 34,140s (9.5 hours); temperature simulation yields a maximum temperature of 346.24K, with no overheating occurring. Finally, the reliability models estimate that an overall MTTF is 112,684, corresponding to 12.86 years of operations without faults.

Figure 9 pictorially shows how the framework allows tracing the evolution of functional and extra-functional quantities in a single simulation run. The figure exemplifies the simulation of about 1s for the core in isolation; for the sake of illustration, we have assumed a 100% utilization of the processor in the interval considered. The core is activated at time 40.0s (top curve); the execution results in a current demand (second curve from top) with some fluctuations due to the different current consumption of different instructions.

The increase in the power demand determines a corresponding change in temperature (third curve from top); it leaves the stable value reached when the core was idle and tends to increase asymptotically reflecting the power trace. Given the short timescale, temperature variations are very small, affecting only the second decimal digit.

The modified operating conditions of the core determine also a decrease of the MTTF (bottom curve), since higher utilization, temperatures, and power consumption decrease component reliability. As for temperature, MTTF also exhibits a very small variation during this short interval.

The line styles used in the plot highlight also the different timescales in the various layers; even if they are simulated simultaneously, each layer works at a different "rate". In the functional and power layer the lines appears as solid, but this is due to the fine granularity of the timescale. Conversely, for temperature and MTTF markers (*i.e.*, the activation points of the corresponding models) are clearly visible. As an example, the reliability model is updated only after each second in response to the evolution of the other properties: at time 40,

Table V
VALIDATION OF POWER SIMULATION VS. MATLAB/SIMULINK.

Tool	Samples (#)	Lifetime (s)	Avg. error	Speed up
SIMULINK	34,140,008	34,140.5	-	-
SC-AMS	34,140,002	34,140.2	6.0E-6%	28.4X

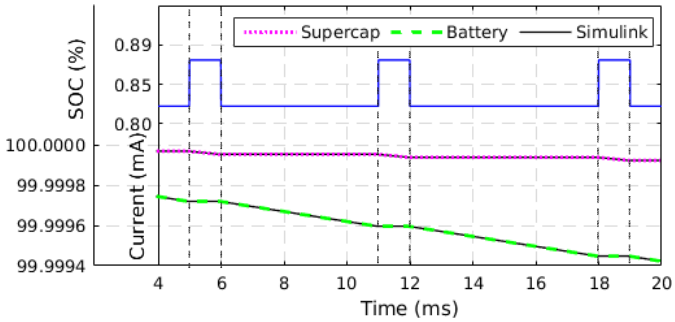


Figure 10. Power Simulation: Comparison against Simulink.

the core is still idle; then at time 41 the model is recomputed and the MTTF is adjusted to reflect the new power, functional and thermal conditions.

F. Validation of Results

Since no existing framework allows the simultaneous simulation of all layers, we validated the power and temperature layers individually using state-of-the-art simulators.

Power validation: For the validation of the power layer, we reproduced the power models used in our work and the information exchange in Matlab/Simulink. For a fair comparison, *i.e.*, to avoid including the overhead of inter-layer dependencies, SystemC-AMS simulation is restricted only to the power layer; information from other layers like temperature and workload is provided as pre-calculated waveforms. Simulations use the same time step of 1 ms. Table V compares the two power simulations; it can be notice that the proposed approach accurately tracks the consumed power and the estimated system lifetime, virtually with no approximation. In order to get a sample measure of the error, we used battery and capacitor voltages; the average error (measured as the absolute difference between the Simulink and SystemC-AMS waveforms) is smaller than 0.0001% (6.0E-6%, with maximum error below 8E-5%). This confirms the visual fidelity resulting from Figure 10, where the Matlab/Simulink and SystemC-AMS curves are totally overlapped. Such a good accuracy is achieved with speedup of simulation time of about 28.4X, due to the fact that the Matlab/Simulink internal solver is heavier than the efficient SystemC-AMS implementation of TDF and ELN [43].

Temperature Validation: The reference for validating thermal simulation is the widely used Hotspot simulator [36]. As for power, SystemC-AMS simulation is now restricted to the temperature layer and power data are provided as a pre-calculated 2000ms waveform. Simulations use the same time step of 1 ms, that is the default time step for Hotspot.
Table VI shows that both RC thermal networks instantiate the same number of elements (nodes, resistors and capacitors), and confirms the accuracy of the SystemC-AMS simulation by reporting a negligible error (calculated as the average sample-by-sample difference). Figure 11 shows an excerpt of the

Table VI
VALIDATION OF TEMPERATURE SIMULATION VS. HOTSPOT.

Tool	Nodes (#)	Resistors (#)	Capacitors (#)	Avg. error	Speed up
HOTSPOT	28	64	28	-	-
SC-AMS	28	64	28	0.034%	14.57x

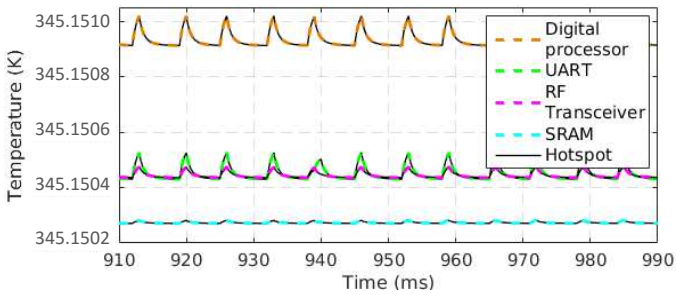


Figure 11. Temperature Simulation: Comparison against Hotspot.

simulation, from which it can be noticed how the HotSpot curves and the SystemC-AMS ones are barely distinguishable. SystemC-AMS simulation proved to be 14.5X faster than Hotspot, thanks to the lighter circuit solvers used by the SystemC-AMS simulation kernel (*i.e.*, Euler and trapezoidal) compared to those of HotSpot (*i.e.*, adaptive Runge-Kutta).

Reliability validation: Since the reliability models use with no modifications the RAMP library available at [35], validation is immaterial; our results simply replicate those of RAMP without any approximation.

G. Impact of inter-layer signals

The simultaneous simulation of all properties of a system in a single simulation run allows to capture their mutual impact and to track them more accurately. Figure 12 that exemplifies this feature for power and temperature.
A time slot of high activity of the core determines a larger current demand for a prolonged time (top plot); this causes an increase of temperature in correspondence to the next time step (arrow ①). Eventually, when current demand decreases, temperature decreases accordingly (arrow ②). Moreover, since temperature affects static power consumption, in the next idle interval of the core (time 637, arrow ③), the resulting idle power appears to be increased from 0.1mW to 0.1005mW.
Such a run-time tracking of the mutual influence of two simulated quantities could not be simulated by trace-based approaches, which need to force a pre-determined order and dependence between properties.
Moreover, this cyclic dependence does not lead to system instability. Both intra- and inter-layer signals are SystemC signals, whose value is updated after a delta of simulation time, rather than instantaneously. This makes the updated values for current and temperature available at the next activation of the models: as an example, the current value updated by the power layer at time 626ms (arrow ②) is made available to the temperature layer at the next timestep (at time 630ms).

H. Extra-functional impact on simulation

One last experiment concerns the impact of the choice of the timescales on accuracy and simulation time. Due

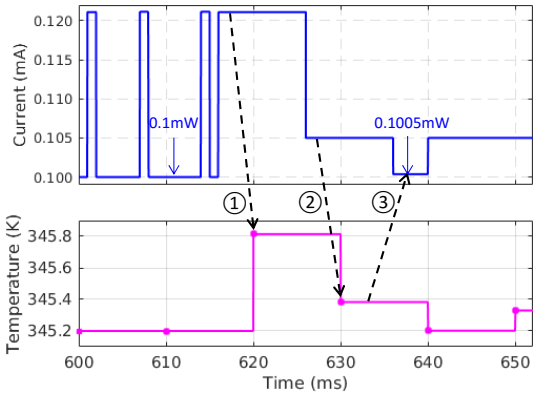


Figure 12. Excerpt of the evolution of the power model and of the temperature model of the core.

to space constraints, in our exploration we preserve the ratios between the three extra-functional time scales (power:temperature:reliability = 1:10:1,000) and vary only the ratio between the functional timescale (*i.e.*, the clock cycle) and the power timescale. We express this ratio by the quantity $TR = \frac{Timescale_{power}}{T_{clock}}$. The experiments reported in the previous section use a value of $TR = 10^4$.

Figure 13 shows the result of this analysis, by plotting the percentage overhead required for simulating the extra-functional layers with respect to the functional simulation time vs. TR. For the later we considered both a SystemC RTL (left plot) and a C++ implementation [12] (right plot).

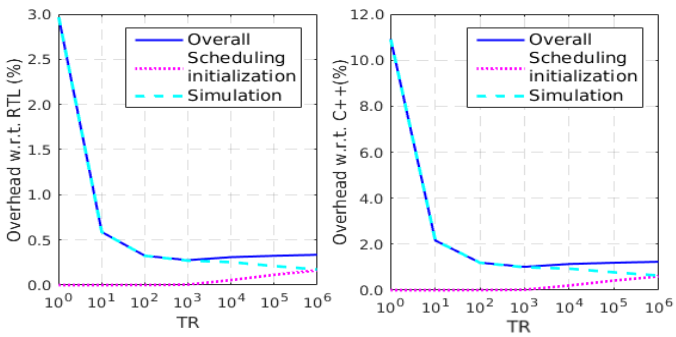


Figure 13. Extra-functional Simulation Overhead vs. TR.

The figure clearly shows that the simulation of extra-functional properties results in an extremely low overhead on functional simulation (solid lines). Given the same implementation and TR, the overhead is slightly higher when the functionality is implemented in C++ (*avg.*, 2.69%) than for RTL (*avg.*, 0.73%), as a result of the higher level of detail of RTL.

As expected, the figure clearly shows that for both C++ and RTL the overhead decreases for increasing values of TR. For instance, executing the power models every 10 clock cycles ($TR = 10^1$) provides about a 6x reduction of the overhead with respect to $TR = 1$, *i.e.*, invoking the power model at each clock cycle. The overhead tends to stabilize for values of TR in the range $10^2 - 10^4$), in which the overhead is about 0.3% vs. C++ and 1% vs. RTL.

Interestingly, for larger values of TR, the overhead tends to slowly increase. This behavior can be explained by considering that any SystemC-AMS execution is divided into initialization and simulation. As TR increases, the power models are computed less frequently, thus lowering the impact of simulation (dashed lines); on the other hand, larger time scales affect the construction of the TDF schedule, performed in the initialization phase (dotted lines). As a matter of fact, for $TR > 10^6$, SystemC runs out of memory when building the entire TDF schedule. We can notice how the chosen set of timescales ($TR = 10^4$) falls in the flat region where the overhead is minimum.

VIII. Conclusions

This work targeted the high degree of heterogeneity of smart systems by proposing a methodology for comprehensive simulation of both functional and extra-functional properties. The methodology is bus centric, to separate information flows and management of the specific properties, and highly modular, to allow easy extension to further components and properties. The effectiveness of the proposed approach is proved on a smart system case-study, featuring functionality, power, temperature and reliability. The methodology proved to be correct and efficient in the modeling of single properties with respect to state of the art frameworks, with substantial speedup and average errors lower than 0.1%. Furthermore, the methodology allowed to reproduce all properties in a single simulation run, thus capturing at run time the mutual impact of properties and overcoming the limitations of state of the art tools.

References

[1] Accellera. *Recommended Vendor Extensions to IEEE 1685-2009 (IP-XACT)*, 2013. www.accellera.org.

[2] A. Al-Hammouri. A comprehensive co-simulation platform for cyber-physical systems. *Comput. Commun.*, 36(1):8–19, 2012.

[3] A. Bauer, J. Hanisch, and E. Ahlswede. An Effective Single Solar Cell Equivalent Circuit Model for Two or More Solar Cells Connected in Series. *IEEE PHOT*, 4(1):340–347, Jan 2014.

[4] B. Beckmann, Y. Eckert, M. Arora, S. Gurumurthi, S. Reinhardt, and V. Sridharan. A comprehensive timing, power, thermal, and reliability model for exascale node architectures. In *MODSIM*, 2013.

[5] N. Bombieri, F. Fummi, and G. Pravadelli. Automatic abstraction of RTL IPs into equivalent TLM descriptions. *IEEE TCOMP*, 60(12):1730–1743, 2011.

[6] D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *ACM EMSOFT*, pages 2:1–2:12, 2013.

[7] Y. Chen, S. Vinco, E. Macii, and M. Poncino. Fast thermal simulation using SystemC-AMS. In *Proc. of ACM GLSVLSI*, pages 427–432, 2016.

[8] Y. Choi, N. Chang, and T. Kim. DC-DC Converter-Aware Power Management for Low-Power Embedded Systems. *IEEE TCAD*, 26(8):1367–1381, 2007.

[9] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *ACM/IEEE DATE*, page 16, 2011.

[10] A. Danese, G. Pravadelli, and I. Zandonà. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Proc. of IEEE/ACM DATE*, pages 606–611, 2016.

[11] A. Davare, D. Densmore, T. Meyerowitz, R. Pinto, A. Sangiovanni-vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next generation design framework for platform-based design. In *Accellera DVCon*, 2007.

[12] EDALab. *HIFSuite - EDA Software Tools for HDL and Virtual Platforms*, 2016. www.hifsuite.com.

[13] A. Elsheikh, E. Widl, and P. Palensky. Simulating complex energy systems with Modelica: A primary evaluation. In *IEEE DEST*, pages 1–6, 2012.

[14] 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 *IEEE/ACM DATE*, pages 1–4, 2014.

[15] L. D. Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni, and S. Vinco. UNIVERCM: The UNIversal VERsatile computational model for heterogeneous embedded system design. In *IEEE HLDVT*, pages 33–40, 2011.

[16] L. Guo, Q. Zhu, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee. Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems. In *IEEE/ACM CODES+ISSS*, pages 1–10, 2014.

[17] H. Huang, G. Quan, and J. Fan. Leakage temperature dependency modeling in system level analysis. In *Proc. of IEEE ISQED*, pages 447–452, 2010.

[18] IEEE Standard. Standard SystemC language reference manual. *IEEE Std 1666-2011*, pages 1–638, 2012.

[19] IEEE Standard. Standard SystemC analog/mixed-signal extensions language reference manual. *IEEE Std 1666.1-2016*, pages 1–236, 2016.

[20] IEEE/IEC Standard. IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows. *IEC 62014-4 IEEE Std 1685-2009*, pages 1–373, 2015.

[21] JEDEC solid state technology association. *Failure Mechanisms and Models for Semiconductor Devices*, 2006. JEP122C.

[22] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Multi-mechanism reliability modeling and management in dynamic systems. *IEEE TVLSI*, 16(4):476–487, 2008.

[23] I. Koren and C. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., 2007.

[24] Maxim Integrated. *MAX3108 SPI/I2C UART*, 2015. www.maximintegrated.com.

[25] Panasonic. *EECS0HD334 Electric Double Layer Capacitors Datasheet*, 2015. industrial.panasonic.com/ww/products/capacitors/edlc.

[26] S. Park, Y. Wang, Y. Kim, N. Chang, and M. Pedram. Battery Management for Grid-connected PV Systems with a Battery. In *ACM/IEEE ISLPED*, pages 115–120, 2012.

[27] M. Petricca, D. Shin, A. Bocca, A. Macii, E. Macii, and M. Poncino. An automated framework for generating variable-accuracy battery models from datasheet information. In *ACM/IEEE ISLPED*, pages 365–370, 2013.

[28] W. Peukert. Über die Abhängigkeit der Kapazität von der Entladestromstärke bei Bleiakkumulatoren. In *Elektrotechnische Zeitschrift*, page 20, 1897.

[29] S. Rhoads. Plasma CPU Core, 2001. opencores.org.

[30] M. A. Sakka, H. Gualous, N. Omar, and J. V. Mierlo. *Batteries and Supercapacitors for Electric Vehicles*, pages 135–164. InTech, 2012.

[31] K. Skadron, M. Stan, W. Huang, S. Velusamy, et al. Temperature-aware microarchitecture. In *Proc. of ACM ISCA*, pages 2–13, 2003.

[32] K. Skadrony, M. Stanz, M. Barcellaz, A. Dwarkaz, et al. Hotspot: Techniques for modeling thermal effects at the processor-architecture level. In *Proc. of THERMINIC*, pages 1–4, 2002.

[33] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili. KitFox: Multiphysics libraries for integrated power, thermal, and reliability simulations of multicore microarchitecture. *IEEE TCPMT*, 5(11):1590–1601, 2015.

[34] J. Srinivasan, S. Adve, P. Bose, and J. A. Rivers. Lifetime reliability: toward an architectural solution. *IEEE Micro*, 25(3):70–80, May 2005.

[35] J. Srinivasan and P. Ramachandran. The RAMP Lifetime Reliability Model (Version 2.0), 2005. rsim.cs.uiuc.edu/ramp/ramp20.

[36] M. Stan, K. Skadron, M. Barcella, W. Huang, K. Sankaranarayanan, and S. Velusamy. Hotspot: a dynamic compact thermal model at the processorarchitecture level. *Elsevier Microelectronics Journal*, 34:1153–1165, 2003.

[37] STMicroelectronics. *EFL700A39 EnFilm rechargeable solid state lithium thin film battery datasheet*, 2013. www.st.com.

[38] STMicroelectronics. *STLQ015150 mA - ultra low quiescent current linear voltage regulator*, 2015. www.st.com.

[39] V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee. Instruction level power analysis and optimization of software. In *IEEE ICVD*, pages 326–328, 1996.

[40] A. Vachoux, C. Grimm, and K. Einwich. Towards analog and mixed-signal SOC design with SystemC-AMS. In *Proc. of IEEE DELTA*, pages 97–102, 2004.

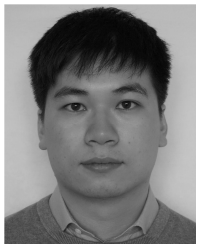
[41] D. Veillard. *The XML C parser and toolkit of Gnome*. xmlsoft.org.

[42] S. Vinco, V. Guarnieri, and F. Fummi. Code manipulation for virtual platform integration. *IEEE TCOMP*, 65(9):2694–2708, 2015.

[43] S. Vinco, A. Sassone, F. Fummi, E. Macii, and M. Poncino. An open-source framework for formal specification and simulation of electrical energy systems. In *IEEE/ACM ISLPED*, pages 287–290, 2014.

[44] Y. Zhang, F. Zhang, , Y. Shakhsheer, J. D. Silver, et al. A Batteryless 19mu W MICS/ISM-Band Energy Harvesting Body Sensor Node SoC for ExG Applications. *IEEE SCC*, 48(1):199–213, Jan 2013.

Sara Vinco (M09) received the Ph.D. degree in computer science from the University of Verona, Italy, in 2013. She is currently a Post-Doctoral Research Associate at the Department of Control and Computer Engineering, Politecnico di Torino, Italy. Her main research interests are energy efficient electronic design automation and techniques for simulation and validation of heterogeneous embedded systems.



Yukai Chen (M15) received the M.Sc. degree in Computer Engineering at Politecnico di Torino, Italy, in 2014. He is working toward the PhD degree in the Department of Control and Computer Engineering at Politecnico di Torino. His main research interests focus on computer-aided design for integrated circuits and electrical energy systems, with particular emphasis on modeling and simulation of extra-functional properties of cyber-physical systems.

Franco Fummi (M92) received the Ph.D. degree in electronic engineering from Politecnico di Milano, Italy, in 1995. He is currently the Head of the Department of Computer Science, University of Verona, Italy, where he is a Full Professor since 2000, and where he became an Associate Professor in computer architecture in 1998. Since 1995, he has been with the Department of Electronics and Information, Politecnico di Milano, as an Assistant Professor. He is a co-founder of EDALab, an EDA company developing tools for the design of networked embedded systems. His current research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of embedded systems.

Enrico Macii (SM02, F07) Enrico Macii is a Full Professor of Computer Engineering at Politecnico di Torino, Italy. Prior to that, he was an Associate Professor (1998-2001) and an Assistant Professor (1993-1998) at the same institution. From 1991 to 1997 he was also an Adjunct Faculty at the University of Colorado at Boulder. He holds a Laurea Degree in Electrical Engineering from Politecnico di Torino (1990), a Laurea Degree in Computer Science from Università di Torino (1991) and a PhD degree in Computer Engineering from Politecnico di Torino (1995). Since 2007, he is the Vice Rector for Research at Politecnico di Torino; he was also the Rector's Delegate for Technology Transfer (2009-2015) and for International Affairs (2012-2015). His research interests are in the design of electronic circuits and systems, with particular emphasis on low-power consumption, optimization, testing, and formal verification. In the last few years, he has been growingly involved in projects focusing on the development of new technologies and methodologies for smart cities and bioinformatics. In the fields above he has authored over 450 scientific publications. Enrico Macii is a Fellow of the IEEE.

Massimo Poncino (M97,SM12) received the Ph.D. degree in Computer Engineering and the Dr.Eng. degree in Electrical Engineering from the Politecnico di Torino, Italy. He is currently a Full Professor of Computer Engineering at Politecnico di Torino. His research interests include several aspects of design automation of digital systems, with particular emphasis on the modeling and optimization of low-power systems. He is the author or coauthor of more than 300 journal and conference papers. He is an Associate Editor of the ACM Transactions on Design Automation of Electronic Systems and of IEEE Design & Test. Prior to that, he was an Associate Editor of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2006-2012). He was the Technical Program Co-Chair (in 2011) and the General Chair (in 2012) of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED). He serves on the Technical Program Committee of several IEEE and ACM technical conferences, including DAC, ICCAD, DATE, ISLPED, ASP-DAC, CODES-ISSS, and GLSVLSI.