

IP-XACT for Smart Systems Design: Extensions for the Integration of Functional and Extra-Functional Models

Original

IP-XACT for Smart Systems Design: Extensions for the Integration of Functional and Extra-Functional Models / Vinco, Sara; Lora, Michele; Macii, Enrico; Poncino, Massimo. - ELETTRONICO. - (2016), pp. 1-8. (Intervento presentato al convegno Forum on Specification & Design Languages (FDL), 2016 tenutosi a Bremen nel 14-16 settembre 2016) [10.1109/FDL.2016.7880379].

Availability:

This version is available at: 11583/2651259 since: 2020-02-22T22:16:57Z

Publisher:

IEEE

Published

DOI:10.1109/FDL.2016.7880379

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)

IP-XACT for Smart Systems Design: Extensions for the Integration of Functional and Extra-Functional Models

Sara Vinco*, Michele Lora[†], Enrico Macii*, Massimo Poncino*

*Dept. of Control & Computer Engineering,
Politecnico di Torino,
Torino, Italy
name.surname@polito.it

[†]Dept. of Computer Science,
Università degli Studi di Verona,
Verona, Italy
michele.lora@univr.it

Abstract—Smart systems are miniaturized devices integrating computation, communication, sensing and actuation. As such, their design can not focus solely on functional behavior, but it must rather take into account different extra-functional concerns, such as power consumption or reliability. Any smart system can thus be modeled through a number of views, each focusing on a specific concern. Such views may exchange information, and they must thus be simulated simultaneously to reproduce mutual influence of the corresponding concerns.

This paper shows how the IP-XACT standard, with some necessary extensions, can effectively support this simultaneous simulation. The extended IP-XACT descriptions allow to model extra-functional properties with a homogeneous format, defined by analysing requirements and characteristic of three main concerns, *i.e.*, power, temperature and reliability. The IP-XACT descriptions are then used to automatically generate a skeleton of the simulation infrastructure in SystemC. The skeleton can be easily populated with models available in the literature, thus reaching simultaneous simulation of multiple concerns.

I. INTRODUCTION

Compared to classical embedded systems, smart systems are characterized by their *smartness*, that is, the ability to learn from the previous experience and to seamlessly react to the surrounding environment [1]. This tight interaction with the physical environment implies a high level of heterogeneity, in terms of both components and design constraints. Correctness lies indeed not only in the functionality, but also in a number of extra-functional constraints that must be met or optimized, ranging from power consumption to reliability [2]. Furthermore, due to their heterogeneity, many different stakeholders are involved in the design flow, each focused on a particular set of concerns. This impacts on the traditional design flow, that must take into account such extra-functional aspects in all simulation, validation and optimization steps. This is far from trivial, since it requires to integrate models captured in different dedicated languages and tools [3], [4].

Functional languages and standards responded to this evolving scenario with extensions covering new aspects and domains [5]–[7]. As an example, Hardware-Description Languages (HDL), like SystemC and Verilog, support extra-traditional domains, such as power and mechanical systems, thanks to their Analog and Mixed-Signal (AMS) exten-

sions [8]–[11]. Such extensions may allow to include in a single simulation multiple aspects of the same system, all implemented with the same language and managed by the same simulation kernel [12].

The IP-XACT standard for interface specification [13] did not fall behind. Initially designed for digital IP specification, it has indeed been recently extended to cover also AMS descriptions, physical design and power characteristics of the IPs [7]. The extensions are mainly devoted to physical characteristics of circuit and package (*e.g.*, supply nets, combinational paths, and compliance with energy conservation laws). They are thus useful when dealing with the final product, *e.g.*, to integrate it in a larger system or to apply further redesign and synthesis. However, the modeled information focuses on too low-level information, that does not allow to build an integrated simulation framework comprising simultaneously both functionality and extra-functional aspects (*e.g.*, power and heat flows).

To overcome this gap, this work shows how IP-XACT, with necessary extensions, can effectively support the design of smart systems with the integration of extra-functional aspects in a single simulation framework. The novel contributions of this work are:

- the identification of a *suitable simulation framework*, supporting the modeling of different views and concerns, to reconcile different aspects of the same system in a single simulation framework. In particular, the paper exemplifies the idea by focusing on three different extra-functional concerns, particularly important for smart systems design, *i.e.*, power consumption, thermal dissipation and reliability;
- the definition of *extensions to the IP-XACT standard*, that allow to model extra-functional aspects, with the application to the identified concerns;
- the IP-XACT files are then used for *automatic generation* of a skeleton of the simulation code, that can be easily populated with state-of-the-art models to reach seamless simulation of the overall system.

The paper is organized as follows. Section II provides the necessary background. Section III outlines the proposed approach, that is detailed in Sections IV, V and VI. Section VII concludes the paper with some remarks.

This work has been partially supported by the European project CONTREX (FP7-ICT-2013-10-611146).

II. BACKGROUND

A. Multi-view modeling

Broman *et al.* in [3] proposed the application of the ISO/IEEE standard 42010 [14] to the field of heterogeneous systems. The authors considered that any system includes different aspects of interest (called *concerns*), including, *e.g.*, functionality, power, temperature or reliability. Stakeholders involved in the design flow may be interested in more than one concern, *e.g.*, to consider at the same time reliability aspects and power consumption. It is thus necessary to produce models capable of capturing multiple concerns of the same system simultaneously. The system is thus subdivided into different *views*, each focusing on a specific concern and implementing a subset of the overall system behavior. Views may exchange relevant information to reproduce mutual effects, *e.g.*, between power consumption and temperature, or between reliability and functionality. This must be implemented through an information exchange mechanism, reproducing how different concerns may influence each other.

Implementing multi-view modeling allows to cover multiple views in a single simulation framework, thus covering simultaneously heterogeneous aspects of the same system. This would provide a powerful tool to the stakeholders involved in the design of the system.

B. Heterogeneous smart system simulation

Approaches available in the literature for smart system simulation face the challenges posed by heterogeneity and integration of different concerns with a variety of strategies. The most straightforward solution consists of integrating different tools, each native of a specific aspect of the system [15]–[18]. Despite guaranteeing the correctness in the modeling of specific aspects, this solution introduces heavy and error-prone overheads in the integration of the different tools [12].

Many frameworks have been proposed for simulating different concerns in a single framework or tool. Top-down approaches, like PtolemyII [19], support a number of Models of Computation (MoC), so that any component can be modeled with the most suitable simulation semantics. Even if this allows to cover also physical phenomena, integration of different MoCs is manual, and reuse of existing components requires long configuration and integration times. Extra-functional features and orthogonalization of concerns are explicitly supported by the MetroII [20] simulation environment, that allows to annotate physical quantities into functional descriptions. However, MetroII does not ease the reuse of existing IPs: designers must convert system components to the MetroII semantics, and to annotate the extra-functional values manually.

Other approaches exploit SystemC-AMS flexibility to build a single framework covering different aspects, ranging from communication protocols to physical behaviors [9], [21]. However, this kind of solutions restricts the support to a single aspect (*e.g.*, power [9]) or model extra-functional evolution with high-level models (*e.g.*, waveforms or physical equations), thus missing more complex behaviors like energy flows and thermal evolution [21].

C. IP-XACT

IP-XACT aims at easing IP integration and reuse through the formalization of IP interfaces and protocols [13]. It defines a standard XML format, that supports two main description

schemas. A *component definition* is univocally identifiable by the quadruple called VLNV, composed by the component's *Vendor*, *Library* of IPs, *Name* of the component and its *Version*. A component definition essentially contains the interface of an IP, provided as a list of ports. Each port is defined by a name, a type, a direction, width and usage information. A *design definition* represents the instances of components in a system and the interconnection between them.

Over time, some extensions have been proposed to the IP-XACT standard. A lot of effort has been spent on extending the support to SW and to HW-SW communication [22]–[26]. Accellera proposed an extension supporting analog-mixed signal descriptions and IP characteristics in terms of physical design and power distribution [7]. However, these extensions focus on the implemented physical circuit, and thus can not be used to build a runtime simulation environment.

A successful support for extra-functional simulation has been provided by [9], that extends IP-XACT to the modeling of power flows. Despite of the similarity *w.r.t.* the current approach, [9] limits the scope to the sole power view, thus not fully supporting the simulation of a smart system.

D. SystemC

Despite of being a HDL, SystemC has been widely adopted for the modeling of both digital HW, embedded SW and networked systems [27], [28]. This flexibility is realized thanks to the discrete event semantics (that well matches all the mentioned domains) and to its being a C++-based language. However, SystemC is strictly limited to discrete-time descriptions, thus not covering a wide range of components and behaviors of typical smart systems.

The support for smart systems has been widened by its analog and mixed-signal extension, *i.e.*, SystemC-AMS. To cover a wide variety of domains, SystemC-AMS provides three different modeling formalisms, supporting different communication styles and representations. *Timed Data-Flow* (TDF) models are scheduled statically by considering their producer-consumer dependencies in the discrete time domain. *Linear Signal Flow* (LSF) supports the modelling of continuous time through a library of pre-defined primitive modules (*e.g.*, integration, delay). Finally, the *Electrical Linear Network* (ELN) formalism models electrical networks through the instantiation of pre-defined primitives associated with electrical equations, *e.g.*, resistors or capacitors. As such, SystemC-AMS is expressive enough to allow for the modeling of physical quantities evolution within SystemC models.

III. OVERVIEW

A. Application of multi-view modeling to Smart Systems

The goal of this work is to apply the multi-view modeling presented in Section II-A to the context of smart system design.

To do this, we consider a smart system as a set of different overlapping views, each focusing on a specific concern. Each component of the smart system may participate to a number of views by providing view-specific models and interfaces, thus implementing the features of the component *w.r.t.* a specific design concern. This concept is depicted in Figure 1. The system is modeled through three views (one for the functionality, and two extra-functional views). The same component (Component1) is provided with one model per view,

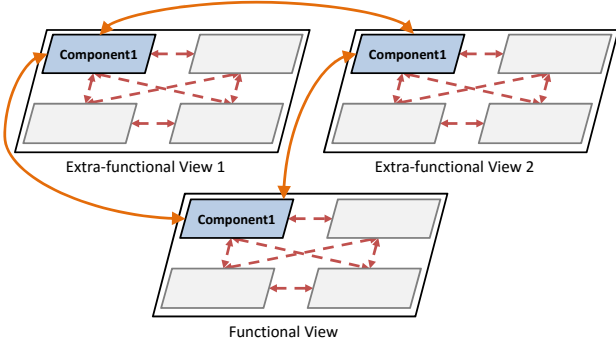


Fig. 1. Multi-view simulation applied to the context of smart system design. The system is provided with a number of views, and each component may provide a model for each view. Solid arrows depict inter-view communication for Component₁, while dashed lines depict intra-view communication.

each implementing the evolution of the component *w.r.t.* the modeled concern.

Each view restricts the focus to a specific concern of the system. *Intra-view communication* is used to model the exchange of information among component models belonging to the same view (*i.e.*, modeling the same concern). This kind of communication is required to share local information globally (within the view), and to compute global information about the modeled concern (*e.g.*, to reproduce energy and heat flows). This type of communication is strictly carried by concern-specific signals and ports (the dashed arrows in Figure 1).

Vice-versa, *extra-view communication* aims at modeling information exchanges between different views of the system (the solid arrows in Figure 1). This kind of communication is necessary to model mutual influences between different concerns, *e.g.*, the impact of functionality on power consumption, or of the latter on thermal dissipation. This kind of communication can employ concern-specific signals and ports, but it may also adopt standard signals to represent custom information that is not necessary for characterizing the single concerns but rather to determine the mutual effect, depending on the information nature.

B. Methodology overview

Figure 2 represents the proposed flow, where colored boxes are for the novel contributions. As a first step, we analyzed the *typical concerns of interest* for the modeling and simulation of smart systems (1). This allowed to determine the necessary *extensions of the IP-XACT standard*, with the goal of applying multi-view modeling to the context of smart system simulation (2). This constitutes the main novelty of the current work. The extended IP-XACT format allows to *model all functional and extra-functional interfaces* of system components with the same formalism (3). Note that the construction of the IP-XACT description of a system is manual, as it is specific of the single case study. The IP-XACT files determine a skeleton of the simulation framework, in terms of component interfaces and of system connections. The next step is thus *automatic generation of a skeleton of the simulation framework* (4). In this work, the target language is SystemC, whose extensions allow to cover a wide range of concerns with a single language. Finally, the generated code must be *populated with state-of-the-art models* describing the behavior of system components in the context of each view (5). This consists of implementing the SystemC modules with suitable models available at state-of-the-art. This

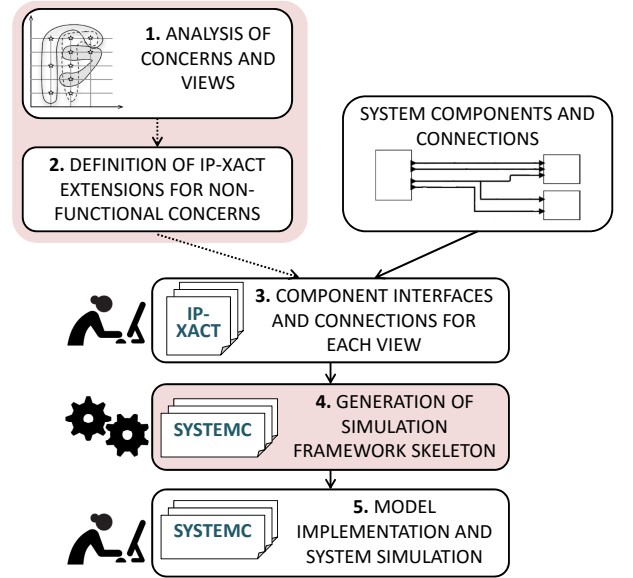


Fig. 2. Overview of the proposed approach. Colored boxes represent the novelty proposed by the current work.

step is strictly dependent on the system under development, and is thus entirely manual.

IV. ANALYSIS OF TYPICAL SMART SYSTEM CONCERNS

The extra-functional characterization of smart system may focus on a number of physical concerns. In this work, we currently focus on three main concerns, that are traditionally considered the most relevant physical phenomena [29]: *power consumption* (or production), *heating*, and *reliability*. Such concerns, together with functionality, constitute the most relevant aspects of a smart system, and the main focus of this work. Each concern determines the construction of a specific view of the system.

Table I reports the views corresponding to each concern, described in terms of modeled behaviors of interest and of mutual influence *w.r.t.* the other views:

- the *functionality view* represents the “computational” be-

TABLE I. TYPICAL SMART SYSTEM VIEWS, WITH CORRESPONDING CONCERNS AND POSSIBLE INTERSECTIONS WITH OTHER VIEWS.

View	Behavior of interest	Mutual influence
FUNCTIONALITY	Functional implementation	State (<i>e.g.</i> , EFSM state, fetched instruction) and duty cycle influencing power consumption and thermal behavior. Hazards due to architectural choices (<i>e.g.</i> , pipeline) impacting on reliability.
POWER	Power (consumption or production), current (consumption or production), operating voltage	Power, current, voltage influencing heating production and reliability. Faulty behavior due to lack of energy.
THERMAL	Temperature (absolute or relative), heat flow	Overheating worsening batteries and capacitors performance, and causing faulty behavior (<i>e.g.</i> , delays). Reliability degradation due to thermal cycling.
RELIABILITY	Expected lifetime, Mean Time To Failure (MTTF), failure rate ($\lambda = \frac{1}{MTTF}$)	May affect component functionality due to lower MTTF and through the use of fault-tolerant circuitry.

havior of devices. Functionality can be modeled in a variety of fashions, ranging from state machines (e.g., for digital HW) to sequential SW. Power consumption may vary with the evolution of the functionality, since different amounts of power may be necessary to perform different computation. Thermal behavior may be influenced by the active portions of the system, or by the operating frequency. Finally, architectural choices, e.g., pipelines, and heavy computation cycles may increase circuit stress and error probability, with the result of worsening reliability;

- the *power view* represents the energy flows inside of the smart system, i.e., how energy flows from power producers (power sources and storage) to consumers. As anticipated, power evolution is influenced by functionality, e.g., by the component operating mode or the fetched instruction (in case of CPUs). Vice-versa, power may influence both functionality and reliability by introducing component failures due to energy lacks or peaks. The power view is strongly connected to the system thermal view, since energy consumption produces heating.
- the *thermal view* focuses on the heat exchanges in the system, and on modeling the evolution of component temperature over time. Temperature is heavily influenced by both power consumption and the functionality, since circuit stress caused by computation increases temperature. Temperature influences power consumption, e.g. by degrading the capacity of energy storage components or by introducing fluctuations in the operating point of components. Temperature peaks and fluctuations may cause an increase in the error rate, and thus affect reliability.
- the *reliability view* measures the expected lifetime of the system, in terms of the Mean Time To Failure (MTTF) or of its reciprocal λ . As anticipated, reliability of a system or a component heavily depends on the other views, as it depends on the power operating point (e.g., on power consumption), thermal range and computation stress on the circuitry. Vice-versa, reliability impacts on the other views, since failures can strike any kind of component, including functional components, energy sources or storage devices, and thermal dissipation devices. Furthermore, in case of fault tolerant systems, it may modify the functional view, e.g., by imposing error correction or detection circuitry.

The outline clearly highlighted the presence of mutual effects between concerns. This proves that composing a stakeholder's viewpoint requires the simultaneous modeling and simulation of all views, not to miss critical dependencies and unexpected evolution caused by the mutual interaction of different concerns. This concept is exemplified by Figure 1: functionality impacts on power consumption (view 1) and thus on temperature (view 2); this may cause an overheating, and force to reduce the functional operating frequency.

V. EXTRA-FUNCTIONAL IP-XACT EXTENSIONS

This Section presents how multi-view modeling can be supported by IP-XACT, either by adopting (and adapting) the current standard, or by proposing necessary extensions. The changes *w.r.t.* the standard are summarized below, and exemplified in Figures 3– 5. To improve the effectiveness of the proposed approach, all proposed extensions have been formalized in additional XSD schemas, that define the namespace `extra-functional`.

```

1. <ipxact:component xsi:schemaLocation="... /extrafunctional/index.xsd">
2.   <ipxact:vendor>vendor2</ipxact:vendor>
3.   <ipxact:library>fdl16</ipxact:library>
4.   <ipxact:name>cpu</ipxact:name>
5.   <ipxact:version>1.3</ipxact:version>
6.   <extrafunctional:concern>power</extrafunctional:concern>
7.   <ipxact:model>
8.     <ipxact:ports>
9.       <ipxact:port><ipxact:name>cpu_state</ipxact:name>
10.        <ipxact:wire ipxact:allLogicalDirectionsAllowed="false">
11.          <ipxact:direction>in</ipxact:direction>
12.          <ipxact:vector>
13.            <ipxact:left>2</ipxact:left><ipxact:right>0</ipxact:right>
14.          </ipxact:vector>
15.          <ipxact:wireTypeDefs><ipxact:wireTypeDef>
16.            <ipxact:typeName ipxact:constrained="false">std_logic_vector
17.            </ipxact:typeName>
18.            <ipxact:typeDefinition>IEEE.std_logic_1164.all</ipxact:typeDefinition>
19.            <ipxact:viewNameRef>vhdlView</ipxact:viewNameRef>
20.          </ipxact:wireTypeDef></ipxact:wireTypeDefs>
21.        </ipxact:wire>
22.      </ipxact:port>
23.      ...
24.      <ipxact:port><ipxact:name>current_demand</ipxact:name>
25.        <ipxact:wire><ipxact:direction>out</ipxact:direction></ipxact:wire>
26.        <ipxact:vendorExtensions><extra-functional:wire>
27.          <extra-functional:domainTypeDefs><extra-functional:domainTypeDef>
28.            <extra-functional:typeName>current</extra-functional:typeName>
29.            <extra-functional:unit>Ampere</extra-functional:unit>
30.            <extra-functional:value unit="Ampere" prefix="milli">3.00
31.            </extra-functional:value>
32.          </extra-functional:domainTypeDef></extra-functional:domainTypeDefs>
33.          <extra-functional:typeDefinition>/extrafunctional/types.xml
34.        </extra-functional:wire>
35.        <accellera:viewNameRef>power_view</accellera:viewNameRef>
36.      </ipxact:port>
37.      ...
38.      <ipxact:port>
39.        <ipxact:name>cpu_temperature</ipxact:name>
40.        <ipxact:wire><ipxact:direction>in</ipxact:direction></ipxact:wire>
41.        <ipxact:vendorExtensions><extra-functional:wire>
42.          <extra-functional:domainTypeDefs><extra-functional:domainTypeDef>
43.            <extra-functional:typeName>temperature</extra-functional:typeName>
44.            <extra-functional:unit>Celsius</extra-functional:unit>
45.            <extra-functional:typeDefinition>/extrafunctional/types.xml
46.          </extra-functional:domainTypeDef></extra-functional:domainTypeDefs>
47.          <accellera:viewNameRef>temperature_view</accellera:viewNameRef>
48.        </extra-functional:wire></ipxact:vendorExtensions>
49.      </ipxact:port>
50.    </ipxact:ports>
51.  </ipxact:model>

```

Fig. 3. IP-XACT component description generated for the power view of a digital core. Dots represent additional port declarations that are hereby omitted for the sake of brevity.

A. IP-XACT port tags

The main difference between functional and extra-functional descriptions lies in the computed and exchanged quantities, and in the ports used to carry them. IP-XACT supports two types of ports: transactional ports (used for high level modeling) and wire ports (that correspond to RTL bit vectors). It is clear that none of these types suits extra-functional quantities.

To overcome this limitation, this work adopts a strategy similar to the one implemented in the Accellera extensions [7], that define additional tags for referencing language-specific types and to provide a unit measure for port values. Our extension relies on three novel tags:

- `<extra-functional:typeName>` specifies the kind of quantity carried by the port. The Accellera extensions use a `typeName` tag to reference language-specific types (e.g., electrical of VHDL-AMS). On the contrary, we defined a library of “semantic” types, expressing the kind of information conveyed by the port. This library is contained by the XSD schema defining the `extra-functional` namespace, that specifies an enumeration of allowed types

based on the analysis in Section IV. The power view uses the semantic types voltage, current, power; the thermal view uses temperature, while the reliability view introduces MTTF, lambda and lifetime;

- `<extra-functional:unit>` specifies the measure unit of the value. This tag is necessary due to the existence of different measure systems applicable to the same physical quantities (e.g., metric versus imperial units). As for the `typeName` tag, the XSD extension schema defines the enumeration of measure units necessary to model the views considered in Section IV: Volt (V), Ampere (A), and Watt (W) for the power view; Celsius (C), Fahrenheit (F) and Kelvin (K) for the thermal view; second (s) and Percentage for the reliability view (the former for MTTF and lifetime, the latter for λ);
- `<extra-functional:magnitude>` specifies the magnitude of a quantity. The tag is optional, and it allows to ease the specification of system of different scales by providing the typical set of metric prefixes (i.e., Tera, Giga, Mega, Kilo, milli, micro, nano, pico).

An example of usage of these tags is provided at lines 22–33 of Figure 3: the `current_demand` port is defined as of type current (line 26), whose definition is contained in the XSD extensions file (line 29). The port type is paired also with an measure unit (Ampere), a prefix (milli) and a default value (3.0, line 27–28). The usage of the measure unit is crucial at lines 34–45 to remove ambiguous quantities, by specifying that the `cpu_temperature` port expresses temperature in Celsius rather than in Kelvin.

Note that, while the `<extra-functional:magnitude>` tag is mostly used to simplify systems specification, information specified by the first two tags is crucial to check the *semantic validity of IP-XACT descriptions*. The `unit` tag allows to verify that the quantities exchanged through two connected ports are compatible (necessary for temperature, e.g., in case of Celsius and Kelvin degrees), and to check whether any conversion is necessary (i.e., when the ports have different prefixes). On the other hand, the semantic types used for the `typeName` tag allow to guarantee that connected ports carry the same information, thus avoiding incorrect assignments (e.g., bindings between a current port and a temperature port). In this way, the resulting description will always be consistent and sound, thus preventing misconnected model components [4].

B. IP-XACT descriptions

The organization of IP-XACT descriptions provided for each system reflects the multi-view modeling approach, as applied to the context of smart systems (Section III-A).

Each component is provided with one *IP-XACT component description* per supported concern, thus reflecting the fact that the component participates to multiple views. Such IP-XACT component descriptions define the view-specific interface of the component as made of two types of ports: view-specific ports (e.g., a current port when the concern is power) and ports used for inter-view communication. Figure 3 shows an example of IP-XACT component description of a CPU, where the concern of interest is power. The description lists three ports for the power view of the CPU: an extra-functional view-specific port (i.e., the `current_demand` port, lines

```

1. <ipxact:design xsi:schemaLocation=".../extrafunctional/index.xsd">
2.   <ipxact:vendor>vendor</ipxact:vendor>
3.   <ipxact:library>fdl_submission</ipxact:library>
4.   <ipxact:name>toplevel</ipxact:name>
5.   <ipxact:version>1.0</ipxact:version>
6.   <extrafunctional:concern>power</extrafunctional:concern>
7.   <ipxact:componentInstances>
8.     <ipxact:componentInstance>
9.       <ipxact:instanceName>cpu</ipxact:instanceName>
10.      <ipxact:componentRef vendor="vendor" library="fdl16"
11.        name="cpu" version="1.0" concern="power"/>
12.    </ipxact:componentInstance>
13.    <ipxact:componentInstance>
14.      <ipxact:instanceName>battery</ipxact:instanceName>
15.      <ipxact:componentRef vendor="vendor" library="fdl16"
16.        name="battery" version="1.0" concern="power"/>
17.    </ipxact:componentInstance>
18.  </ipxact:componentInstances>
19.  <ipxact:adHocConnections>
20.    <ipxact:adHocConnection>
21.      <ipxact:name>processor_consumption</ipxact:name>
22.      <ipxact:portReferences>
23.        <ipxact:internalPortReference
24.          componentRef="cpu" portRef="current_demand"/>
25.        <ipxact:internalPortReference
26.          componentRef="battery" portRef="required_current"/>
27.      </ipxact:portReferences>
28.    </ipxact:adHocConnection>
29.    <ipxact:adHocConnection>
30.      <ipxact:name>battery_level</ipxact:name>
31.      <ipxact:portReferences>
32.        <ipxact:internalPortReference componentRef="battery" portRef="level"/>
33.        <ipxact:internalPortReference
34.          componentRef="cpu" portRef="available_power"/>
35.      </ipxact:portReferences>
36.    </ipxact:adHocConnection>
37.  </ipxact:adHocConnections>
38.  <ipxact:ports>
39.    ...
40.    <ipxact:port>
41.      <ipxact:name>cpu_temperature</ipxact:name>
42.      <ipxact:wire><ipxact:direction>in</ipxact:direction></ipxact:wire>
43.      <ipxact:vendorExtension><extra-functional:wire>
44.        <extra-functional:domainTypeDef><extra-functional:domainTypeDef>
45.          <extra-functional:typeName>temperature</extra-functional:typeName>
46.          <extra-functional:unit>Celsius</extra-functional:unit>
47.          <extra-functional:typeDefinition>/extrafunctional/types.xml
48.        </extra-functional:typeDefinition>
49.        <accelera:viewNameRef>temperature_view</accelera:viewNameRef>
50.      </extra-functional:domainTypeDef></extra-functional:domainTypeDef>
51.      </extra-functional:wire></ipxact:vendorExtensions>
52.    </ipxact:port>
53.    ...
54.  </ipxact:ports>
55. </ipxact:design>

```

Fig. 4. IP-XACT design description modeling an energy consumption view and instantiating the CPU in Figure 3. Dots represent additional tags that are hereby omitted for the sake of brevity.

22–32), and two ports for inter-view communication (i.e., the `cpu_temperature`, used to communicate with the temperature view, at lines 34–45, and the `cpu_state` port, used to communicate with the functional view, at lines 8–21).

Each view is described by one *IP-XACT design description*, listing the components participating to the view and their intra-layer connections. Figure 4 shows an example of IP-XACT design description for a power view (containing the CPU in Figure 3 and a battery). The `<componentInstance>` tag is used to list components participating to the view, by referencing the corresponding IP-XACT component descriptions (lines 8–12 for the CPU).

Intra-view communication is modeled through port bindings between components, by using the `<adHocConnection>` tag. As an example, lines 19–25 describe the binding between the `current_demand` port of the CPU with the `required_current` port of the instantiated battery.

Inter-view communication can not be represented inside of the view-specific IP-XACT descriptions, that focus on a single concern. Ports used for communicating with other layers are exported by defining an interface for the view (lines 35–46).

```

1. <ipxact:design xsi:schemaLocation="... ./extrafunctional/index.xsd">
2.   <ipxact:vendor>vendor</ipxact:vendor>
3.   <ipxact:library>fdl16</ipxact:library>
4.   <ipxact:name>smart_system</ipxact:name>
5.   <ipxact:version>1.0</ipxact:version>
6.   <ipxact:concern>multiple</ipxact:concern>
7.   <ipxact:componentInstances>
8.     <ipxact:componentInstance>
9.       <ipxact:instanceName>functional_view</ipxact:instanceName>
10.      <ipxact:componentRef vendor="vendor" library="fdl16"
11.        name="toplevel" version="1.0" concern="functional"/>
12.    </ipxact:componentInstance>
13.    <ipxact:componentInstance>
14.      <ipxact:instanceName>power_view</ipxact:instanceName>
15.      <ipxact:componentRef vendor="vendor" library="fdl16"
16.        name="toplevel" version="1.0" concern="power"/>
17.    </ipxact:componentInstance>
18.    <ipxact:instanceName>thermal_view</ipxact:instanceName>
19.    <ipxact:componentRef vendor="vendor" library="fdl16"
20.      name="toplevel" version="1.0" concern="temperature"/>
21.    </ipxact:componentInstance>
22.    <ipxact:instanceName>reliability_view</ipxact:instanceName>
23.    <ipxact:componentRef vendor="vendor" library="fdl16"
24.      name="toplevel" version="1.0" concern="reliability"/>
25.    </ipxact:componentInstance>
26.  </ipxact:componentInstances>
27.  <ipxact:adHocConnections>
28.    <ipxact:adHocConnection><ipxact:name>cpu_state</ipxact:name>
29.      <ipxact:portReferences>
30.        <ipxact:internalPortReference
31.          componentRef="functional_view" portRef="cpu_state"/>
32.        <ipxact:internalPortReference
33.          componentRef="power_view" portRef="cpu_state"/>
34.        <ipxact:internalPortReference
35.          componentRef="thermal_view" portRef="cpu_state"/>
36.        <ipxact:internalPortReference
37.          componentRef="reliability_view" portRef="cpu_state"/>
38.      </ipxact:portReferences></ipxact:adHocConnection>
39.    ...
40.    <ipxact:adHocConnection>
41.      <ipxact:name>battery_mttf</ipxact:name>
42.      <ipxact:portReferences>
43.        <ipxact:internalPortReference
44.          componentRef="reliability_view" portRef="battery_mttf"/>
45.        <ipxact:internalPortReference
46.          componentRef="thermal_view" portRef="battery_mttf"/>
47.        <ipxact:internalPortReference
48.          componentRef="power_view" portRef="battery_mttf"/>
49.      </ipxact:portReferences>
50.    </ipxact:adHocConnection>
51.  </ipxact:adHocConnections>
52. </ipxact:design>

```

Fig. 5. IP-XACT file generated for the top level of the guiding example. Dots represent tags hereby omitted for the sake of brevity

The overall system is described in a *top-level IP-XACT design description*, that instantiates all the views and implements inter-view communication through port bindings. Figure 5 shows how the power view (described in Figure 4) is instantiated (lines 12–15) and bound to the other views to reproduce inter-layer communication (lines 24–40).

According to the standard, an IP-XACT description is uniquely identified by the VLNV, *i.e.*, an aggregation of four identification tags: vendor, library, name and version. However, IP-XACT files referring to different views of the same component (or system) may have the same identification tags. To overcome this problem, this paper proposes to extend the VLNV to a *VLNVC identifier*, composed by adding a tag called *<concern>*. This additional tag reports the specific concern described by the current IP-XACT file (lines 2–6 of Figures 3–5). This tag must be specified whenever referencing an IP-XACT design or component extra-functional description. To allow compatibility with the current standard, the additional tag is not required when modeling the functional view.

A clear example of the usage of VLNVC is provided by lines 6–23 of Figure 5. The top level instantiates the views by referencing the corresponding IP-XACT design descriptions. Note that all views have the same VLNV attributes, since they

```

1. SCA_TDF_MODULE (cpu_power_view){
2.   public:
3.     // intra-view interface
4.     sca_tdf::sca_out< double > current_demand;
5.     sca_tdf::sca_in< double > available_power;
6.
7.     // extra-view interface
8.     sca_tdf::sca_out< double > cpu_consumption;
9.     sca_tdf::sca_in< double > cpu_temperature;
10.    sca_tdf::sca_in< sc_lv<3> > cpu_state;
11.
12.    // constructor and destructor
13.    cpu_power_view(sc_core::sc_module_name name_){}
14.    ~cpu_power_view(){}
15.
16.   private:
17.     // TDF functions
18.     void set_attributes();
19.     void initialize();
20.     void processing();
21. };

```

Fig. 6. Example of SystemC-AMS TDF code skeleton generated for the digital core in Figure 3.

refer to the same version of the same component (*i.e.*, the values of the vendor, library, name and version tags are the same for all IP-XACT files). Thus, the IP-XACT files would not be distinguishable. The additional *<concern>* tag allows to differentiate and to reference either the functionality (no *<concern>* tag, lines 6–10), or the extra-functional views (power at lines 11–15, temperature at lines 16–19 and reliability at lines 20–23).

Finally, it is worth noting that the *concern* tag for the top-level is set to *multiple* (line 6). This keyword will enable IP-XACT users to develop tool-chains capable of easily identifying multi-view models, and thus of treating them properly.

VI. AUTOMATIC CODE GENERATION OF THE SIMULATION SKELETON

The hierarchy of IP-XACT files generated for the system can be used to enhance the design flow through automatic code generation, by implementing a skeleton of the simulation framework. The generated code can be implemented in a variety of languages, including AMS HDLs and tools such as Matlab/Simulink. In the context of this paper, the choice fell on SystemC-AMS, that proved to efficiently support the implementation of extra-functional behaviors [9], [10].

A. Component code generation

The code generation process reflects the organization of the IP-XACT files. Each *IP-XACT component description* is converted to one SystemC module. As a result, each component of the smart system will be implemented as a number of SystemC modules, one per supported concern. This enhances modularity, since multiple aspect of the same component can be added without affecting the previous implementation.

The instantiated modules can be implemented either as SystemC standard modules (SC_MODULE) or as SystemC-AMS TDF modules (SCA_TDF_MODULE, as in Figure 6). The former choice leaves more freedom to the subsequent module implementation, since no requirement is imposed on the semantics (*e.g.*, in terms of level of abstraction and simulation semantics). A SystemC module can indeed encapsulate both an event-driven implementation (based on processes), the instantiation of a hierarchy of sub-modules (including TDF modules), or a topology of ELN primitives. On the contrary, the declaration of a TDF module imposes the data


```

1. #include "cpu_power_view.h"
2. #include "battery_power_view.h"

3. SC_MODULE(smart_system_power_view) {
4. private:
5.     // instantiated components
6.     cpu_power_view * processor;
7.     battery_power_view * battery;
8.     ...
9.     // connecting signals for intra-view communication
10.    sca_tdf::sca_signal<double> processor_consumption;
11.    sca_tdf::sca_signal<double> battery_level;
12.    ...
13. public:
14.     // ports signals for inter-view communication
15.    sc_core::sc_in<double> cpu_temperature, cpu_mttf,
        battery_temperature, battery_mttf;
16.    sc_core::sc_out<double> cpu_consumption, battery_level;
17.    sc_core::sc_in<sc_lv<3>> cpu_state;

18.    // constructor
19.    SC_CTOR(smart_system_power_view){
20.        processor = new cpu_power_view("processor");
21.        battery = new battery_power_view("battery");
22.        ...
23.        // intra-view signals binding
24.        processor->current_demand(processor_consumption);
25.        battery->required_current(processor_consumption);
26.        processor->available_power(battery_level);
27.        battery->level(battery_level);
28.        ...
29.    };

```

Fig. 7. SystemC code skeleton generated for the power view in Figure 4.

flow semantics, realized as a static scheduling. This choice may be extremely convenient, since it accelerates simulation by building an efficient interaction between components.

Figure 6 provides an example of SystemC TDF code generated for the power view of the digital processor (see the IP-XACT component description in Figure 3). Note that the code generated for each component is only a *skeleton of the SystemC module*, containing only the signature of constructor and destructor (lines 10–12), and, in case of TDF modules, of member functions imposed by the SystemC-AMS standard (e.g., processing, initialize, lines 15–17).

The module interface is derived by the IP-XACT component description. It lists both ports for inter-view and intra-view communication (e.g., line 4 is the same `current_demand` port as in line 22 of Figure 3). Ports inherit both the name and type from the IP-XACT file. The semantic type of extra-functional ports is mapped onto the `double` type. This allows to represent continuous values typical of physical quantities during simulation, without burdening the execution with checks on type compatibility, that is ensured by the consistency of the IP-XACT model.

B. View code generation

The IP-XACT design description of each view leads to the automatic generation of a SystemC module (`SC_MODULE`), that instantiates all components participating to the view and the signals used for intra-layer communication. Figure 7 sketches the code generated for the power view in Figure 4.

Since the instantiated components may be implemented either in SystemC or in TDF (lines 6–7), the type of signal used for port binding must reflect the semantics of each module: `sc_signal` when both modules are implemented in SystemC, `sca_tdf::sca_signal` when both modules are TDF (e.g., lines 10–11), and SystemC-TDF converters when connected modules follow different semantics.

```

1. #include "smart_system_functional_view.h"
2. #include "smart_system_power_view.h"
... // all the necessary header files
3. SC_MODULE(smart_system) {
4. private:
5.     // instantiated components
6.     smart_system_functional_view * functional;
7.     smart_system_power_view * power;
8.     smart_system_thermal_view * thermal;
9.     smart_system_reliability_view * reliability;
10.    ...
11.    // connecting signals
12.    sc_core::sc_signal<double> cpu_temperature;
13.    sc_core::sc_signal<sc_lv<3>> cpu_state;
14.    ...
15.    // constructor
16.    SC_CTOR(smart_system){
17.        functional = new smart_system_functional_view("functional");
18.        power = new smart_system_power_view("power");
19.        thermal = new smart_system_thermal_view("thermal");
20.        reliability = new smart_system_reliability_view("reliability");
21.        ...
22.        // port binding
23.        power->cpu_temperature(cpu_temperature);
24.        thermal->cpu_temperature(cpu_temperature);
25.        ...
26.        functional->cpu_state(cpu_state);
27.        power->cpu_state(cpu_state);
28.        thermal->cpu_state(cpu_state);
29.        reliability->cpu_state(cpu_state);
30.        ...
31.    };

```

```

1. #include "smart_system.h"
2. int sc_main(int argc, char* argv[]) {

3.     // instantiate system top level
4.     smart_system* toplevel = new smart_system("toplevel");
5.     // start simulation
6.     sc_start();
7.     delete(toplevel);
8. };

```

Fig. 8. Example of top level and main code generated for the guiding example, by reflecting the IP-XACT description in Figure 5. Dots represent additional constructs that are hereby omitted for the sake of brevity.

Ports used for communication with other views (e.g., the `cpu_temperature` port, line 10) are implemented as standard SystemC ports (i.e., `sc_in` or `sc_out`, lines 14–17).

C. Top level generation

The simulation top level is built automatically starting from the top level IP-XACT design description. The top level is implemented as an additional `SC_MODULE`, named after the name tag in the VNLV of the IP-XACT design description (as depicted in top of Figure 8). The module instantiates all system views, as determined by the `<componentInstance>` tags of the IP-XACT design description (lines 6–9 and 17–20). IP-XACT port connections are then implemented through port bindings and the instantiation of support SystemC signals (lines 12–14 and 22–30).

The proposed approach additionally generates a main simulation file (bottom of Figure 8), that instantiates the top level module (lines 3–4) and starts simulation (lines 5–7). This file may be extended by the designer to add a custom testbench, configuration parameters, or tracing instructions.

D. Population of the simulation framework and simulation

The SystemC modules generated for each component must be filled with the implementation of concern-specific models of each component. The following of this Section reports pointers to models available at state-of-the-art. However, the designer may choose any model. The only constraint is that models must

be implemented by using constructs provided by C++, SystemC and SystemC-AMS. Note that the implemented models can be reused at later times, thus easing the population of the SystemC simulation framework. Future work will include the definition of a library of models that can be easily instantiated, thus enhancing the effectiveness of the proposed approach.

a) *Functional models*: the functionality of each component can be either natively implemented in SystemC, or translated from any HDL by adopting automatic tools [30], [31].

b) *Power models*: the most widespread models for power consumption are functional models, that express power demand as equations, state machines or waveforms over time [9], [32]. Functional models can be implemented in either C++, standard SystemC or TDF. Energy providers (e.g., batteries and super-capacitors) may require the adoption of circuit models, that emulate the power behavior through an electrical equivalent circuit [33]. Such models can be easily implemented by using the ELN model of computation [9].

c) *Temperature models*: thermal simulation usually falls back on circuit simulators, that represent temperature in terms of voltages and heat flows as currents [34]. Similarly to power circuit models, also thermal models can be easily implemented with SystemC-AMS ELN primitives [10].

d) *Reliability models*: they are mainly analytical, i.e., they determine system failure rate (or derivative quantities, i.e., MTTF and life time) as a function of: physical stress (e.g., functional duty cycle), operating conditions (e.g., voltage and current), and material-specific coefficients [35]. These models are easily implementable in C++, SystemC or TDF.

VII. CONCLUSIONS

This paper showed how the IP-XACT standard, with few extensions, can support the simultaneous simulation of multiple aspects of a smart system. The extended descriptions allow to uniformly model different functional and extra-functional views of the system, and to gain automatic generation of a skeleton of a SystemC simulation infrastructure. This infrastructure can be easily filled with the desired models by the designer, thus gaining a simulatable implementation of the system that covers both functionality and extra-functional aspects. Future work will further automate the proposed approach, through the automatic construction of the IP-XACT descriptions from a graphical representation, and the construction of a library of models that can be easily instantiated to populate the SystemC simulation framework.

REFERENCES

- [1] S. Vinco and C. Pilato, "Editorial: Special issue on innovative design methods for smart embedded systems," *ACM TECS*, vol. 15, no. 2, pp. 22e:1–22e:2, 2016.
- [2] N. Bombieri, M. Poncino, and G. Pravadeili, Eds., *Smart Systems Integration and Simulation*. Springer, 2016.
- [3] D. Broman *et al.*, "Viewpoints, formalisms, languages, and tools for cyber-physical systems," in *Proc. of ACM MPM*, 2012, pp. 49–54.
- [4] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber-physical systems," *Proc. of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.
- [5] Accellera, *SystemC-AMS*, www.accellera.org/downloads/standards/systemc.
- [6] —, *Verilog-AMS*, www.accellera.org/downloads/standards/v-ams.
- [7] *Recommended Vendor Extensions to IEEE 1685-2009 (IP-XACT)*, Accellera, 2013, www.accellera.org.
- [8] K. Caluwaerts and D. Galayko, "SystemC-AMS modeling of an electromechanical harvester of vibration energy," in *Proc. of IEEE/ECSI FDL*, 2008, pp. 99–104.
- [9] S. Vinco *et al.*, "An open-source framework for formal specification and simulation of electrical energy systems," in *Proc. of IEEE/ACM ISLPED*, 2014, pp. 287–290.
- [10] Y. Chen *et al.*, "Fast thermal simulation using systemc-ams," *Proc. of IEEE GLS-VLSI*, pp. 1–8, 2016.
- [11] A. Holovatyy and V. Teslyuk, "Verilog-AMS model of mechanical component of integrated angular velocity microsensor for schematic design level," in *Proc. of IEEE CPEE*, 2015, pp. 43–46.
- [12] F. Fummi *et al.*, "Moving from co-simulation to simulation for effective smart systems design," in *Proc. of IEEE/ACM DATE*, 2014, pp. 1–4.
- [13] *IEEE Standard 1685-2009 (IP-XACT)*, Accellera, 2010, www.accellera.org.
- [14] "ISO/IEC/IEEE 42010-2011: Systems and Software Engineering, Architecture Description, Recommended Practice for Architectural Description of Software-intensive Systems," www.iso.org/iso/catalogue_detail.htm?csnumber=50508, 2011.
- [15] M. A. Faruque and F. Ahourai, "A Model-Based Design of Cyber-Physical Energy Systems," in *Proc. of IEEE ASPDAC*, 2014, pp. 97–105.
- [16] D. Atienza, G. D. Micheli, L. Benini *et al.*, "Reliability-aware design for nanometer-scale devices," in *Proc. of ACM/IEEE ASPDAC*, 2008, pp. 549–554.
- [17] F. Fummi *et al.*, "Heterogeneous co-simulation of networked embedded systems," in *Proc. of IEEE/ACM DATE*, vol. 3, 2004, pp. 168–173.
- [18] M. Hsieh *et al.*, "SST + Gem5 = a scalable simulation infrastructure for high performance computing," in *Proc. of ACM SIMUTOOLS*, 2012, pp. 196–201.
- [19] J. Eker *et al.*, "Taming heterogeneity - the Ptolemy approach," *Proc. of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan 2003.
- [20] A. Davare *et al.*, "METROII: A design environment for cyber-physical systems," *ACM TECS*, vol. 12, no. 1s, p. 49, 2013.
- [21] J. Molina, X. Pan, C. Grimm, and M. Damm, "A framework for model-based design of embedded systems for energy management," in *Proc. of IEEE MSCPES*, 2013, pp. 1–6.
- [22] S. Pendharkar, "On extending IP-XACT for device driver software generation," 2014, vayavayalabs.com/pdf/ipxact_ve.pdf.
- [23] A. Kamppi *et al.*, "Kactus2: Environment for embedded product development using IP-XACT and MCAP," in *Proc. of Euromicro DSD*, 2011, pp. 262–265.
- [24] A. E. Mrabti *et al.*, "Extending IP-XACT to support an MDE based approach for SoC design," in *Proc. of IEEE/ACM DATE*, 2009, pp. 586–589.
- [25] A. Kamppi *et al.*, "Extending IP-XACT to embedded system HW/SW integration," in *IEEE SoC*, 2013, pp. 1–8.
- [26] E. Vaumorin, "SPIRIT IP-XACT extensions and exploitation for verification software methodology," 2006, www.dempa.co.jp/magillem/pdf/SPIRIT_Methodology.pdf.
- [27] Accellera, *SystemC standards*, www.accellera.org.
- [28] F. Fummi *et al.*, "A SystemC-based framework for modeling and simulation of networked embedded systems," in *Proc. of ECSI/IEEE FDL*, 2008, pp. 49–54.
- [29] D. Brooks *et al.*, "Power, thermal, and reliability modeling in nanometer-scale microprocessors," *IEEE Micro*, vol. 27, no. 3, pp. 49–62, 2007.
- [30] N. Bombieri *et al.*, "HIFSuite: Tools for HDL code conversion and manipulation," *EURASIP JES*, pp. 1–20, 2010.
- [31] EDAUtils, *Verilog to SystemC Translator*, www.edautils.com.
- [32] V. Tiwari *et al.*, "Instruction level power analysis and optimization of software," in *Proc. of IEEE ICVD*, 1996, pp. 326–328.
- [33] M. Petricca *et al.*, "An automated framework for generating variable-accuracy battery models from datasheet information," in *ACM/IEEE ISLPED*, 2013, pp. 365–370.
- [34] M. R. Stan *et al.*, "Hotspot: a dynamic compact thermal model at the processorarchitecture level," *Elsevier Microelectronics Journal*, vol. 34, pp. 1153–1165, 2003.
- [35] J. Sriniivasan, S. Adve *et al.*, "Lifetime reliability: toward an architectural solution," *IEEE Micro*, vol. 25, no. 3, pp. 70–80, May 2005.