Using MDA to Automate the Integration of Virtual Platforms for System-Level Simulation

David Perillo
Elettronica SpA, Roma
Email: david.perillo@elt.it

Marco Di Natale
Scuola Superiore S. Anna, Pisa
Email: marco@sssup.it

***Abstract*—This paper presents the work performed at (removed for blind review) to automate the integration of virtual systems development (VSD) and simulation in its embedded software development process.**

The approach is based on a combination of meta-models, model transformations and design patterns, the SysML standard and the use of the open source Eclipse framework. The purpose is to derive all the design refinements, including the production code and the code used for simulation and verification from a single set of SysML models.

Stereotypes and model transformations are defined to allow the integration of automatically generated interfaces and manually produced code implementing virtual platforms for the simulation of HW/SW heterogeneous systems on the SIMICS platform.

I. Introduction

Embedded software runs in a tight interaction with the hardware, often codesigned or codeveloped. When the complexity of the system grows, the difficulty to investigate malfunctionings and errors increases. Hardware-software interactions provide some of the most critical challenges in the verification process since they may be the source of two types of errors: misalignment between components interfaces, and components interoperability and behavioral interaction errors.

The traditional debugging approach is to run the embedded software on the target hardware to perform the test cases and collect traces to identify errors in the source code. This strategy is inefficient: it occurs late; it requires several iterations before the bug is fixed; and it keeps the target hardware busy for a long time just to solve clerical errors in the code. Most of the times the target hardware is not dedicated to only one software team or is physically not available because it has not been yet realized. In addition, hardware-software interaction errors are typically very difficult to discover and replicate.

The use of virtual platforms (VPs) allows testing and verification long before the hardware is available; gives the possibility to run complex tests, inject faults and test the software in conjunction with multiple hardware configurations. These considerations make the use of VPs a critical tool for developing and debugging the interactions of the software with the hardware layer, the HW/SW partitioning and the analysis of the interfaces.

We defined a design, implementation and verification methodology based on SysML models, code generators, and virtual platforms supported by an eclipse-based framework. The framework includes the Papyrus modeller to describe the system, its interfaces and components activation rules in SysML, following the Platform Based Design (PBD) approach. A generation chain, integrated in the framework and based on the Acceleo open toolset, synthetizes hardware-software interfaces and glue code implementing virtual hardware elements on top of the SIMICS full system simulator.

The main contribution of the proposed methodology is to derive the design refinements and the configuration of the virtual platform from a single set of SysML models, representing the system at a higher level of abstraction. The production code and the code used for simulation and verification are derived from these models together with the hardware documentation. The single source of information guarantees the alignment of interfaces.

The SysML models make use of purposely developed profiles and stereotypes containing concepts related to the representation of low-level details of interface to firmware (FPGA) components, functional and mapping elements that are not directly available in the standard SysML or in the Marte [15] profile.

II. Background and State Of the Art

Our framework is based on a customized profile realized on top of SysML, code generation templates and the use of the SIMICS simulator and provides support for the following activities:

- System Modelling, concerning the use of (semi)formal languages to describe systems at different levels of granularity.
- System-Level Synthesis, consisting in a framework for the automatic generation of software and firmware components.
- Hardware/Software cosimulation, by means of Virtual Platforms (VPs).

In the Platform-based design (PBD) methodology functional elements are put in correspondence with platform elements through a mapping model, defining allocation rules and implementation (refinement) constraints. The Metropolis design environment for Heterogeneous Systems uses the Platform-based design methodology [17] for the representation, analysis, and synthesis of systems represented in SystemC. Metropolis supports several compu-

tation models (MOC) by refinement on an underlying generic metamodel [13]. Ptolemy is an environment for modelling and simulating heterogeneous systems. Ptolemy [14] is mostly targeted at simulation. Code generation of implementations is supported only for the Java language.

PBD System models can also be described using the SysML modelling standard [23], which extends UML to provide a language for Systems Engineering specification, analysis, design, verification, and validation. The Marte profile extends UML and SysML to provide standard stereotypes specific for the embedded systems domain.

The Complex framework [6] enables HW/SW co-design and Design-Space Exploration (DSE) of embedded solutions with a platform based design approach. It is based on UML/MARTE models describing SW and HW components and Use case diagrams. M2T generation facilities convert UML models into IP-Xact specifications and SystemC code implementing the simulation platform. A simulation monitor supports the design exploration process by providing numerical estimates of WCETs and power consumptions. The Complex framework enables fast DSE iterations deriving the design refinements from UML and Simulink models but requires significant SystemC programming for increasing the accuracy of HW models.

The SPRIT consortium formalized the IP-XACT specification of IP component interfaces through an XSD metamodel. SPRIT can be used to model hardware-software interfaces but lacks low-level details like dual page memory banks and default register values.

Manually editing IP-XACT XML models is complex, error prone and time consuming. The Kactus2 framework [7] extends IP-XACT to support SW and HW mapping. The tool provides export facilities to ModelSim; Altera Quartus and Verilog. The need to express timing requirements and to integrate IP-XACT specification in Papyrus is addressed in [8], providing a partial UML/MARTE representation of IP-XACT. The elements in our UML profile provides concepts not directly available in IP-XACT or MARTE, and OCL constraints to validate models.

An example of System-Level Synthesis is the IP-based design methodology providing automatic generation of HW/SW interface components[12]. Plaintext files, Excel sheets, custom XML formats, IP-XACT specifications, and SystemRDL code can be used to describe hardware-software interfaces and register lists [9]. An initial C++ or SystemC register 'stub' model can be automatically generated from SPRIT IP-XACT XML definitions [1]. EDAUtils provides a complete IP-XACT solution for the creation and integration of IP cores. After the generation of the HW/SW interface code, the internal logic of the hardware components can be implemented manually, following the Transaction Level Modelling (TLM) standard. TLM 2.0 is also at the core of the SystemC language[24]; a C++ Library compliant to the TLM 2.0 standard that is widely adopted to create Electronic System Level models (ESL) [10]. SystemC can be used to model hardware interfaces, internal logics and hardware parallelism. Similarly to VHDL and Verilog, the library comes with a simulation kernel that allows both Loosely-Timed (LT) and Approximately-Timed (AT) coding styles depending on the required timing accuracy and simulation speed.

An intermediate solution is the software-timed accuracy (ST), implemented in the WindRiver Simics simulator [3], which is in essence event-based simulation. Events (such as for instance a read operation from a memory register) are handled by the simulation kernel and result in the execution of a simulated behavior.

Several solutions and commercial tools are available to support the creation of Virtual Platforms (VPs), capable of executing the same binary software as the physical hardware being simulated. Instruction Set Simulation tools (ISS) allow to execute the binary software unmodified, including a full Operating System stack. Differently from cycle-accurate (CAS), phase-accurate (PAS) and timing-accurate (TAS) simulators, which can emulate the processor at the physical signal level [11], ISS cannot simulate superscalar ordering effects [2]. SystemC-TLM implementations of Instruction Set Simulators (ISS) are also available. QEMU [16] is an Open Source ISS emulating a large number of microprocessor and devices. New devices can be modelled in C code using QDev and connected in a device tree in which devices are recursively connected via busses. Direct connection between devices is not possible. QBox and TLMu integrate a SystemC bridge on QEMU ISS, enabling the possibility to increase the timing accuracy. OVPSim (Open Virtual Platform Simulator) is another open solution available only for ARM processors. Simics provides a large set of processors and integrated devices plus a model builder tool for creating models of IP cores using the DML device modelling language.

Our workflow is based on the Model Driven Architecture (MDA) OMG standard, enabling the user to describe complex application logics with a formal semantic, independently from the specific execution platform. OMG has defined the standard specifications for transformation languages: model-to-text, as described in MOFM2T [19] and model-to-model, as described in MOF/QVT [20]. Transformation languages are based on the OCL syntax [18] to constraint expressions on UML models. We decided to implement profiles and transformations on the Eclipse platform [22] because it provides an open implementation of the above OMG standards: Acceleo implements the model-to-text transformation standard; QVT-Operational implements the model-to-model transformation standard, and Papyrus is the graphical UML editor for eclipse. It provides a strict implementation of the OMG UML [21] and advanced customization capabilities.

Intel CoFluent Studio [1], is a commercial tool generating a Simics simulation platform from MARTE models and SystemC implementations of IP cores. It provides advanced features for modelling and simulation. Cofluent is oriented to performances estimation and does not include any hardware-software interface design tool. Similarly to CoFluent we realized a SysML profile and a Simics gen-

eration toolchain, but including an accurated hardware-software profile from which we can also generate C++ drivers and VHDL interface code.

III. Running Example

A (necessarily simplified) Radar Warning Receiver (RWR) demonstrator, currently mounted on small sized Unmanned Aerial Vehicles and operating in the 6-18 GHz is used as a case study and running example throughout the paper. An RWR intercepts (measures and analyses) electromagnetic signals produced by nearby radars through a wideband receiver. The signal processing procedure is composed by the following sequential activities:

1) sampling the electromagnetic spectrum according to precalculated settings;
2) analysis of the sampled signal in the frequency domain, and preprocessing of the samples;
3) a supervisor including the analysis of the sampled signal in the time domain (deinterleaving) to produce a synthetic representation of the surrounding electromagnetic spectrum (the list of all the emitters); and the update of receivers and preprocessing modules driving the above activities.

Activities 1 and 2 are implemented in firmware on FPGA boards, whereas activity 3 is implemented in software on a Power PC (PPC) processor.
A software-firmware data exchange is carried through a system bus and implies a software read/write operation on the hardware-software interfaces exposed by the firmware on the system bus. The firmware activities are triggered by internal clocks or by software write operations (a write operation is equivalent to a signal edge for the firmware). In order to gain efficiency, activities 1 and 2 can be pipelined with activity 3.

The next sections show how this embedded system has been modelled to automatically generate a virtual platform including the PPC ISA, the system bus, the FPGA model with the functionality executing on it, and the hardware-software interfaces.

IV. Extending the Simics Full System Simulator

Simics is a commercially available, instruction-accurate, instruction set architecture simulator that enables the user to execute the software on a virtual platform (VP) model executing on a workstation and matching the execution model of the final target. The configuration of the VP is defined in a .simics configuration file that defines the HW component models to be loaded and how they are connected (such as buses and serial lines).

The simulation is started by executing the Simics binary and passing the previously defined .simics configuration file as a parameter. The simulation can be controlled using the Simics Command Line Interface (CLI) terminal.

Component models emulating commercial hardware are generally available in the Simics Models Library. Custom components models can be created using the *Simics Model Builder* tool, which is a compiler for creating executable models out of Python, C and DML code.

In our case study we load from the Simics Models Library one instance of the Freescale MPU-8641d embedded-processing-unit (EPU) to simulate a PPC architecture. The MPU-8641d component model already includes a dual-core PPC, two ethernet cards, a read-only flash memory with a bootable operative system (a lightweight Linux distribution) a DMA controller, PCI and PCI express system bus slots. The MPU8641 board shall be connected to a Radiofrequency receiver (operating on an analytic RF scenario) and a DSP FPGA through the PCI system bus. Using the Simics Model Builder tool we implemented the latter components.

A custom Simics component model is an aggregation of devices, connectors and interfaces, as shown in fig. 1. A device is described in terms of its hardware-software interface (implemented by means of simics *configuration objects*); status variables organized in a C data structure; and provided interfaces to other simulated devices. Connectors are used to connect devices through their interfaces.

Software running on top of the Embedded Processing Unit interacts with low-level firmware modules accessing its hardware-software interface with read and write operations on the system bus. Simics offers a standard API for interacting with the simulation flow. The API functions allow detecting memory transactions initiated by the software. Configuration objects, used to define registers instantiated at a specified address on a shared memory-space on the system bus, can redefine properties and redeclare methods that have a default implementation in the simics API. For instance, a read/write memory transaction initiated by the software and terminating in the memory space of a configuration object produces the invocation of a (possibly overridden) built-in method *access(operation_descriptor)*. Other methods can be redefined to handle read/write accesses, read/write misses, initializations and register resets (hard/soft).

Configuration Objects can also react to events raised by other objects.

By means of reimplemented built-in methods, interfaces and events it is possible to simulate the reactive behaviour of a custom developed (FPGA or HW) component by:

1) detecting memory transactions corresponding to the software issuing commands to a hardware component;
2) collecting the data stored in the component interfaces;
3) executing the function to be realized in FW or HW, described in C/C++ or SystemC code;
4) update the device status and the component interfaces with the results of the algorithm.

In our approach, the activities in 1 and 2 are realized by DML code that is automatically generated from a SysML model, the activities 3 and 4 are realized in DML and C++ implemented manually or generated automatically from Simulink models.

DML (Device Modelling Language) is a Domain Specific Language developed by Wind River to simplify the coding activity of Configuration objects: It provides specific constructs to define registers, banks, fields; redeclare built-

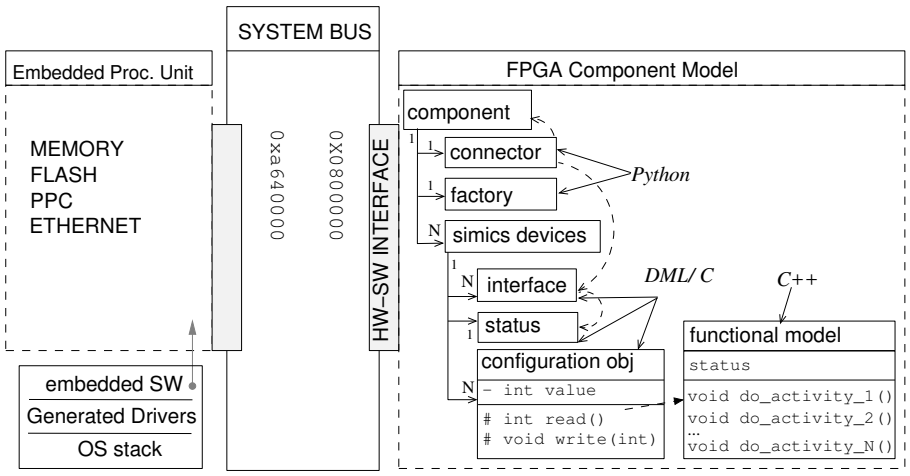

Fig. 1. Simics component models of FPGA and Embedded Processing Unit. The application software runs on the PPC ISA.

in methods and objects properties; and access the simics API transparently. A compiler translates the DML code in plain C code using the simics API.

Listing 1 shows a simple example of a DML specification that models a memory-mapped device with a single 32-bit (4-byte) register at offset 0, which upon a read access will return the value 42 as the result. A Python factory class is required to instantiate the component model and connect it to other components [5].

Listing 1. generated dml code implementing a `MemIF`

```
bank b {
parameter function = 0;
register r0 size 4 @0x0000 {
method read() -> (result) {
result = 42;
}}}
```

V. Outline of the Process and SysML Profiles

The design flow begins after the system level specifications (SSS) are released. It is composed of three main activities: System Modelling, System-level synthesis and Hardware Simulation.

1) System Modelling: SysML models provide a description of the embedded platform to be simulated. Models are organized in Functional, Platform and Mapping packages according to the PBD paradigm. We customized the SysML editor with OCL validation rules, customized palettes, css stylesheets, and icons. The functional models define the functional partitioning with the main subsystems and their input-output dependencies. Then, platform models are created and the Mapping models define the assignment of the functionality to the (HW) components of the execution platform.

2) Synthesis of firmware interfaces, communication and synchronization code: The eclipse-based framework supporting our methodology provides a generation facility to synthetize from SysML the platform models, the component interfaces in DML, VHDL, and C for the simulated components, the real firmware components, and the software drivers.

The Mapping model is used to generate glue code for the virtual platform, connecting the component interface (in DML) to the C++ routines simulating firmware components. The simulation of the firmware components implemented by the FPGA is performed by writing C++ code that is functionally equivalent to the VHDL implementation. We assume that FPGA functionality (often modeled and autogenerated in Simulink) executes according to a Synchronous Reactive behavior. This means that a single reaction operation activated in correspondence to a trigger event will perform (based on the inputs and the iternal state) the update of the state and of the output registers of the FPGA.

3) Hardware Simulation: provides a test bench to validate the system architecture and improve the quality and speed of the software developments. The coherency between simulated and non-simulated firmware components is achieved using the VP as a golden reference for firmware development. When a firmware modification is required, the SysML and the VP models shall be updated in advance.

VI. System Modelling

The profile architecture is organized according to the Platform Based Design and it depends on a common profile (AbstractItems) providing rules, basic stereotypes and model navigation facilities (ref fig. 2).

A set of OCL constraints restricts the relationships among stereotyped items to those that are meaningful. For instance, an OCL rule specifying that a register can own fields but cannot own other registers is shown in listing 2.

Listing 2. OCL rule: child of Registers are PhysicalField

```
self.base_Class.attribute->select( association.oclIsUndefined())
->forAll( getStereotypeApplications()->selectByType(
MemoryLayout::PhysicalField)->size() = 1
```

A. Functional Profile

The functional profile model of fig. 3 contains the definition of the main subsystems in the functional architecture,

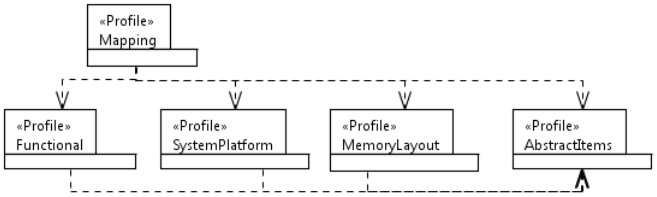

Fig. 2. Profile architecture

including the subsystems to be developed as firmware. Extending the AbstractItems profile, our Functional profile defines the stereotype `FunctionalSystem`, which applies to `SysML::Block` and identifies the root block (or system) in the functional model. `SRFunctionalSubsystem` (applied to `SysML::Block`) defines a subsystem providing a synchronous reactive behaviour.

The output of a synchronous reactive subsystem depends on state and input variables, which are defined by SysML `DataTypes`. Objects of type `SRFunctionalSubsystem` can be composed recursively and can be connected in an input-output relationship using `FlowPorts`. The internal state and the input-output `FlowPorts` of a `SRFunctionalSubsystem` are typed with DataTypes.

`SRInitialize` and `SRUpdate` apply to `SysML::Operation` and identify respectively the unique initialization and step update functions of a synchronous reactive subsystem.

The stereotype `SRUpdateEvent` applies to `Signal` and `Event` and identifies the signal or event that triggers the execution of the update function.

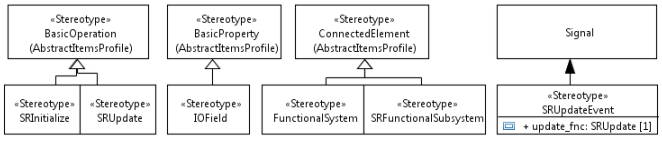

Fig. 3. Functional Profile Elements

B. System Platform Profile

The platform model defines the hardware components supporting the execution of the functional model. The main stereotypes in this profile are shown in Figure 4 (intended to represent the main hardware components of an EW sensor, shown in fig. 5).

`SystemPlatform` is the entry point of the platform model and is a collection of `Rack` elements. Each Rack encloses hardware components of type `HwFpga` (FPGA), `ComputerBoard` (derived from `MARTE::HwDevice`) and `IOBus` (either PCI, VME or PCI express) interconnecting CPU and FPGA components. The `ComputerBoard` represents a board including CPU, memory and basic IO connectors. `CPU` represents a computation unit for software executables. `HwFpga` represents an FPGA connected to the System Bus by a hardware-software interface. `Marte::HwDevice` is used to represents a board including CPU, memory and basic IO connectors. The hardware element `BusBridge` is used to interface different system bus standards.

For modelling serial links (Ethernet, Rs232,Rs485,...) we introduced the stereotypes `HwPort` (applied to Uml::Port) and `HwLink` (applied to Uml::Connector). Two specializations of `HwPort`: `IOBusInterface` for `IOBus`; `HwFpgaInterface` for `HwFpga`, provide the means to specify the hardware-software interface exposed by the FPGA on the system bus, referencing a `MemIF` (ref. para. VI-C).

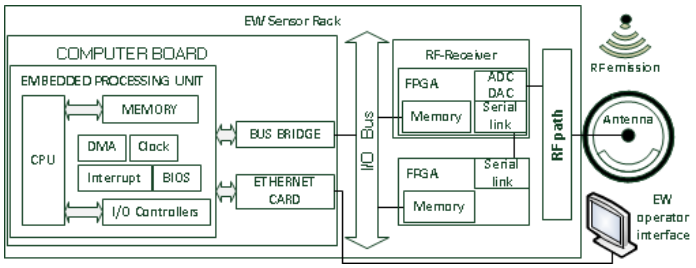

Fig. 5. EW sensor main elements

C. Hardware-software interface Profile

Hardware-software interfaces are described in terms of registers and fields as shown in fig. 6. `MemIF` (applied to `SysML::Block`) is the root element; it is a collection of `MemoryRegisters` grouped by contiguous memory segments (`BoardSection`) with homogenous characteristics like endianess, addressing word size and memory registers size. `Register` defines a register on the system bus owning logical partitions called `Fields`. For every memory-element stereotype, an Instance stereotype (applied to `SysML::Association`) has been defined with the name, the offset, the size and the multiplicity of the owned memory element instance: `MemoryRegister` is instantiated by compositions stereotyped with `MemoryRegisterInstance`; `Group` by `GroupInstance`; `BoardSection` by `BoardSectionInstance`.

D. Mapping profile

The profile Mapping, shown in figure 7, defines the stereotypes of general use for the mapping of functional elements onto the system platform. `MappingRoot` is the root element of the Mapping model. It references elements of platform and functional models with `Dependency` links.

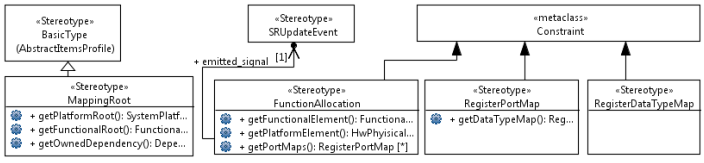

Fig. 7. Mapping profile

In the case of an interface between SW and FW, the mapping information in the model defines:

- The functional subsystem implemented in SW (mapped on a processor core) and the communicating

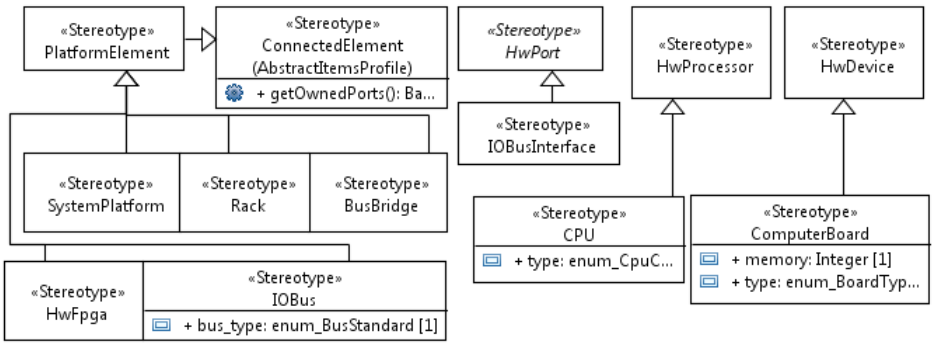

Fig. 4. System Profile Elements

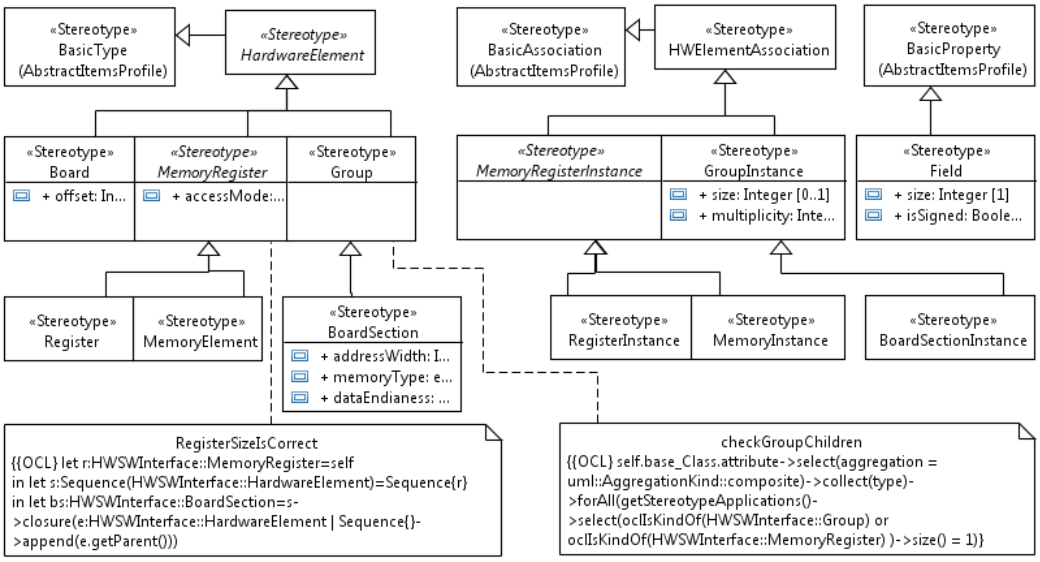

Fig. 6. Hardware-Software Interface Profile Elements

functional subsystem as mapped to an FPGA board (computation mapping).
- How the data communication defined in the functional model is mapped onto the HW register interface (data communication mapping).
- What is the implementation of the Signal/Event that is specified as the trigger of the update function of the subsystem implemented in FW (reaction event mapping).

All the mappings are obtained by stereotyping a `Uml::Constraint` element and leveraging its associations with the functional and the platform elements. This choice is in contrast with the typical definition of mappings as stereotyped associations. However, there are several advantages in the use of constraints. The first is that the mapping model remains completely independent from both the functional and the platform models, given that the mapping information is in the constraint and the associations originating from it. Also, the constraint allows for the defintion of mapping constraints and trigger conditions. Finally, the creation of associations to constrained elements is very intuitive and easy in Papyrus.

Computation mapping

The stereotype `Mapping::FunctionAllocation` (as all other mapping stereotypes, applies to `Uml::Constraint`) allows the user to associate an `SRFunctionalSubsystem` to a `HwFpga`, or to a `CPU`. Figure 8 shows the allocation of the Supervisor and the Controller respectively onto the CPU and the FPGA.

Data communication mapping

With respect to the mapping of the data interface, in the case of a software to firmware or firmware to firmware interface, the stereotype `RegisterPortMap` and `RegisterDataTypeMap` defines a two level mapping. `RegisterPortMap`, puts in correspondence a functional flow port with a set of interface registers. In case the flow port is typed with a structured (non-primitive) datatype, and the mapping to the hardware registers is one-to-many, each field of the datatype associated with the port needs to be put in correspondence with a hardware interface register field. A set of `RegisterDataTypeMap` constraints (for convenience created in Papyrus using a table, as in Figure 9) put in relation each register field in the set identified by `RegisterPortMap` with a field of the datatype associated with the port.

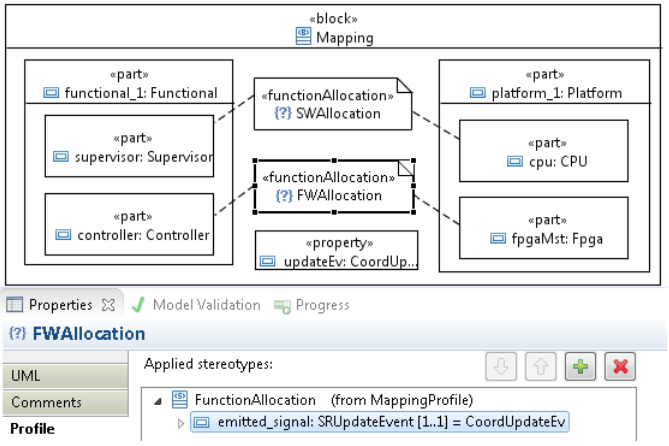

Fig. 8. The allocation of the functional subsystems onto the computing HW

			A
			constrainedElement : Element [*]
0	{?}	xMap	[x_field : Integer, x_ : Integer]
1	{?}	yMap	[y_field : Integer, y_ : Integer]
2	{?}	zMap	[z_field : Integer, z_ : Integer]

Fig. 9. Row Data Table

This mapping convention requires that the flow ports are types with structured data types having all properties of a primitive type. Figure 10 shows the application of the port and data type mapping concepts to our example. For simplicity, only one data type mapping constraint is shown (the elements put in a relationship are shown on the bottom right). Also, these mapping definitions are typically not meant to be shown graphically, but only stored in the model.

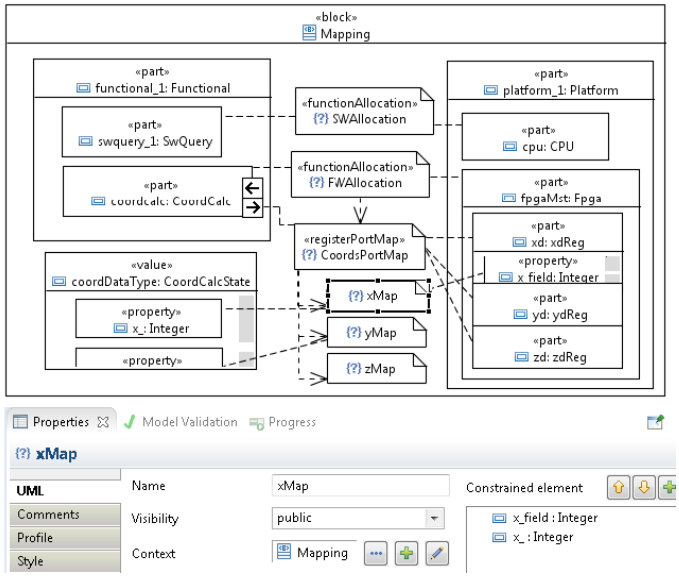

Fig. 10. Mapping ports and data types

Reaction event mapping

The stereotype `FunctionAllocation` has a dependency link towards `RegisterPortMap` for each port of the functional component. Also, it owns the attribute **SRUpdateEvent** (bottom of Figure 8 in the example), referencing the event triggering the execution of the `SRUpdate` function. Constraint stereotyped as `RegisterDataTypeMap` provide the implementation of the triggering mechanism for the SRUpdate function of the depending `RegisterPortMap`. This implementation can be a reference to a clock, if the update function is to be called periodically, or a logic condition on the values of the fields of the ports that are referred by the constraint. In this case, the constraint will indicate as constrained elements all the hardware fields that need to be considered for the trigger condition (for example, a control register) and the constraint formula (in C) will express the condition to be verified on the field bits (such as, for example, a start bit being set at 1).

VII. Glue Code generation for the Simics simulation elements

From the SysML models, a model-to-code Acceleo transformation generates DML, C, Python, and C++ code implementing the executable models of the virtual platform, together with the Simics configuration files.

1) The .simics configuration file: is generated from the Platform model. It provides the creation and the initialization of the executable models representing commercial hardware components (CPU, Ethernet cards, System Bus) together with the custom hardware components (FPGA). The simulated Ethernet device is connected to the real world network enabling interactions with the EW user interface. An extract of the Acceleo script for th generation of the simics configuration is shown in listing 3.

Listing 3. Acceleo code synthetizing .simics config file

```
1  [template ...(dev : HwPhyisicalElementInstance)]
2    # creation of fpga component [dev.getComponentModuleName()/]
3    load-module [ dev.getComponentModuleName( )/]
4    $[dev.getName()/] = (new-[dev.getComponentModuleName()/]...
5         name = "[dev.getTypeName()/]" )
6    $host_name = "[instance.getName()/]"
7    $ip_address = $user_ip  #configured by the user separately
8    run-command-file "mpc8641-simple-linux-2.6.34-common.simics"
9         $eth_link.connect-real-network-host "stap0"
10        service_node_cmp0.sn.log-type -sub info
11   connect $[sbc.getName()/].[dev.slot()/] ...
12        [dev.cnt()/]=$[dev.getName()/].pci
13   [...]
14 [/template]
```

A. Synchronous reactive glue code

Functional models are elaborated to synthetize the synchronous reactive network implementing the firmware logic. For each `SRFunctionalSubsystem`, a corresponding package is generated, containing: **data structures** holding input, output and state variables of the Functional Subsystem. An Acceleo script generates C/DML data structures from `SysML::DataTypes`. Listing 4 shows the generated code.

Listing 4. input-output-state data structures

```
dml 1.2;
header %{ #include "controllogic_srstate.h" %}
```

```
import "port_def.dml"; //flow ports data types
import "state_def.dml"; //state data types
[...]
extern typedef struct {
       int8* bitmaskrx_param;
       int8* antennatype_param;
} t_actuationinput;
[...]
extern typedef struct {
    struct{ t_actuationinput* in_port_actuation; }inputs;
    struct{ t_samplingcontrol* out_port_sampliccontrol;}outputs;
    t_controllogicstate inner; //S.R. function state
}Controllogic_srstate;
```

The implementation of the **synchronous reactive network**, provides mechanisms to forward the input and output data structure values among the connected functional subsystems, and forward update events to the the C++ class that wraps the Simulink code.

1) The DML code defining an abstract Synchronous Reactive object is given in listing 5 (this code acts like an abstract class in Object Oriented programming, the DML "default" keyword is similar to "virtual" in C++). The init() method is automatically invoked by the Simics engine when the simulation startups; the step_update function is accessible to the DML Mapping code. The generated DML-C glue code provides the redirection from the DML code to the C++ Simulink wrapper class of the ConfiguratorLogic functional subsystem. The glue code also propagates the update reaction to the functional modules connected in cascade to the updated subsystem.

Listing 5. Synchronous Reactive interface DML code

```
dml 1.2;
template SRFunctionalSubsystem {
       method init() { call $this.step_init(); }
       method step_init() default { }
       method step_update() default { }
}
```

2) The C++ class associated to the Synchronous Reactive Functional Subsystem (see listing 6) has a step_init() and a step_update() method to be implemented manually by the user to integrate the code generated from the Simulink coder. This class can access and modify the internal state and the input and output data structures of the functional module declared earlier.

Listing 6. C++ Simulink code wrapper

```
#include "controllogic_sr_def.h" /*functional state def*/
class Controllogic_sr {
private:Controllogic_srstate* state; /*functional state*/
public: Controllogic_sr();/** constructor **/
        void init();  /** holds Simulink init code **/
        void update();  /** holds Simulink update code **/
        Controllogic_srstate *get_state(); /*ret state*/
        void set_state(Controllogic_srstate * arg);
};
```

The definition of the **custom hardware** components. The structural elements of the custom hardware are derived from the Platform and Mapping models. Components of type `HwFpga` are translated into customized Simics components exposing one hardware-software interface and redirecting memory transaction to the mapped synchronous reactive subsystem. The generation script traverses with a depth-first search the HwSwInterface model starting from the `MemIF` element owned by `HwFpgaInterface`. As shown in the Acceleo script of Listing 7, all the memory elements (`Group`, `Register`, `Field`,...) are translated into DML code with offset and size. Write memory transactions are redirected to the DML function *evaluate_transaction* (line 15). This function is generated at the FPGA level and is activated at every memory transaction to check if the data written by the software in the FPGA memory matches the `Uml::Constraint` specification declared in `FunctionAllocation`. When the data stored in the memory registers matches the specification, the corresponding update_step is invoked.

Listing 7. part of the Acceleo script generating hardware-software interfaces

```
1  [template...genInterface(pm :RegisterPortMap)]
2  [for (it_reg : RegisterInstance | pm.getRegister())]
3  [it_reg.genRegister()/]
4  [/template]
5  [template...genRegister(r:RegisterInstance)]
6   register [r.getGroupName()/] size [r.calculateSize()/] {
7     [r.generate_reg_accessMode()/] // read/write
8     parameter offset = $parent.offset + [r.calculateOffset()/];
9     [for (it_f : Field | r.getFields())]
10    [it_f.genField(it_f.calcOffset())/]
11    [/for]   } [/template]
12 [template... genField( f:Field, offset:Integer)]
13  [f.field_signature(offset)/]{
14    method after_write(value){
15       $dev.evaluate_transaction();
16  } } [/template]
```

The generated hardware-software interface is part of a more extensive DML file (reported in listing 8) containing the definition of the FPGA device and the System Bus properties (line 5 and 9). From `Mapping::FunctionAllocation` elements the SR Functional Subsystem is instantiated onto the FPGA (line 18). From line 21 the *Data Communication Mapping* is translated in memory references, so that when the software reads or writes an `HwSwInterface::Field` it inderectly updates the corresponding field of the datatype associated with the functional `FlowPort`. At line 38 the *evaluate_trigger()* method is defined and implemented with the OpaqueExpression specified in the `RegisterPortMap` element. At line 28 the method *call_update* invokes the *step_update* of the constrained Synchronous Reactive C++ class. The *timed_update* function at line 30 shows how *timed events* are managed in DML with the after() function provided by the Simics API.

Listing 8. DML template defining Configurator hwSwInterfaces

```
1  dml 1.2;
2  device configurator_dev;
3  parameter desc = "this is the Configurator device";
4  parameter documentation = "D01...";
5  bank pci_config { is bnk_Configurator; }
6
7  // declare the hardware-software interface
8  template bnk_Configurator {
9      parameter register_size = 4;
10     parameter byte_order = "big-endian";
11     group bspGrp {
12       register antenna_Reg size 6 {
13         field antennaType_fi [16:1] {
14           method after_write(value){$dev.evaluate_transaction();}
```

```
} } } }

// instantiate the SR Functional Subsystem "Controllogic"
bank Controllogic_SRbnk is ( Controllogic_SRtpl ) { }

data double __update_time_ms;
method map_fields_on_SrStatus() {
$this->state.inputs.inport_actuation.antennatType_param =
  cast ( &$dev.pci_config.bsp.antennatType_fi , unit8* );
dataMap.antennaType =
  cast ( &$dev.pci_config.bsp.antennatType_fi , unit8* );
}
method call_update() {
call $dev.Controllogic_SRbnk.step_update(); }
method timed_update( ) {
    call_update( );
    after(__update_time_ms / 1e9) call this.schedule_ms();
}
method set_timed_update( double update_time_ms) {
    __update_time_ms = update_time_ms;
    timed_update();
}
method reset_timed_update( ) { __update_time_ms = 0; }
method evaluate_transaction( ){
    if (dataMap.antennaType==0x32) {
        start_timed_update( 500 );
    }else { stop_timed_update( ); }}
```

VIII. Case Study

1) Case study: functional modelling: An example of functional architecture applied to our case study is shown in fig. 11. The BDD and the IBD models represent the main `SRFunctionalSubsystems` of the RWR.

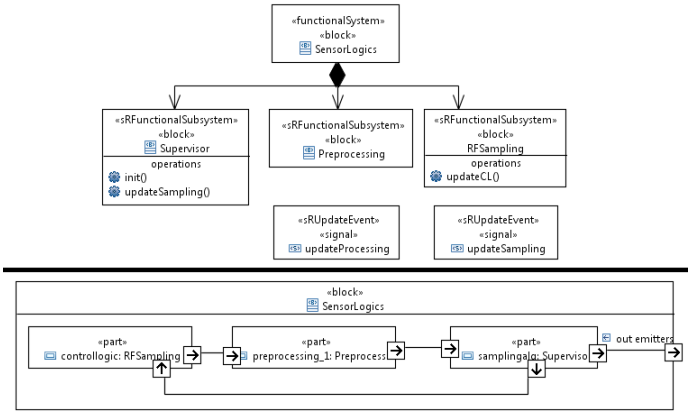

Fig. 11. Functional (BDD and IBD) models of the RWR

2) Case study: System level Platform modelling: The System level platform architecture for our case study is outlined in fig. 12 containing (on the left side) the BDD of the main RWR platform elements. In particular, Receiver and Configurator are `HwFpga` elements, each one referencing an `HwSwInterface` root element (`MemIF`); `ComputerBoard` references a `CPU` with a *PPC* architecture, and PCI bus provides the communication link for the hardware components. The `Rack` IBD diagram shows how the hardware components are connected through `Uml::Ports`.

3) Case study: Mapping model: A small portion of the RWR Mapping model is shown in fig. 13. The image shows the `FunctionalAllocation` elements constraining *sampling* and *pre-processing* `SRFunctionalSubsystems` respectively to *wo_receiver* and *configurator* `HwFpga`. The C-language expression inside the `RegisterDataTypeMap` constraint specification provides the activation rule of the *sampling* functional subsystem. The expression defines a cyclic scheduling at 50ms of the SamplingInput::update function depending on the field ctrlMask: when a software-initiated write operation occurs at the field ctrlMask, the field value is compared with the bitmask 0x7 to determine whether the update routine must be started or stopped.

IX. Results

The proposed framework has been applied to real industrial applications. The modelling activity began from specifications provided with non-standard format (which prevents an automatic import). The modelling activity of the FPGA memory layouts required about 20 man-days for each FPGA. A first iteration of the remaining PBD models has been realized by the first author in about 15 days. After the full modelling activity, further customization for other programs required less than ten ma- days. After modelling the first industrial case, further customization of the PBD SysML models for other programs required on avarage five days per FPGA (a typical sensor integrates 4 FPGAs with 1K fields each). The development of the driver code generator required about two man-months.

From the above models we have generated 2400 lines of code of C drivers with no coding error detected so far and the associated design documentation. Considering a COCOMO II software cost estimation model [8] for embedded projects, the cost of implementing 2400 lines of embedded code is 8 man-months, more than the time required to create interface models from scratch and implement the C++ driver generator. The interface documentation is automatically generated, contextually with the code.

X. Conclusions and Future work

We presented a work for the SysML specification of HW/SW interfaces and the automatic synthesis of interface code and stubs for the simulation of the system in Simics. The framework heavily relies on the SysML standard and open source modelers and model-to-text transformation tools. Our future work includes a revised study of the interaction patterns at the HW-SW interface and a possible extension of the event mapping semantics to capture a more general set of activation conditions and improve the usability of the tool.

References

[1] http://www.intel.it/content/www/it/it/cofluent/cofluent-studio-overview-benefits.html

[2] F. Ghenassia, A. Clouard, *Transaction Level Modeling with SystemC*, Frank Ghenassia , Springer US, 2005

[3] Aarno D., Engblom J. *Software and system development using virtual platforms: full-system simulation with Wind River Simics*, Morgan Kaufmann, Elsevier Inc., 2015. âĂŤ 346 p

[4] Peter S. Magnusson, et al *Simics: A full system simulation platform*, Computer 35 (2002), 50âĂŞ58.

[5] Virtutech *Simics Reference Manual*, http://www2.cs.sfu.ca/ fedorova/Tech/simics-guides-3.0.26/simics-reference-manual-public-all.pdf

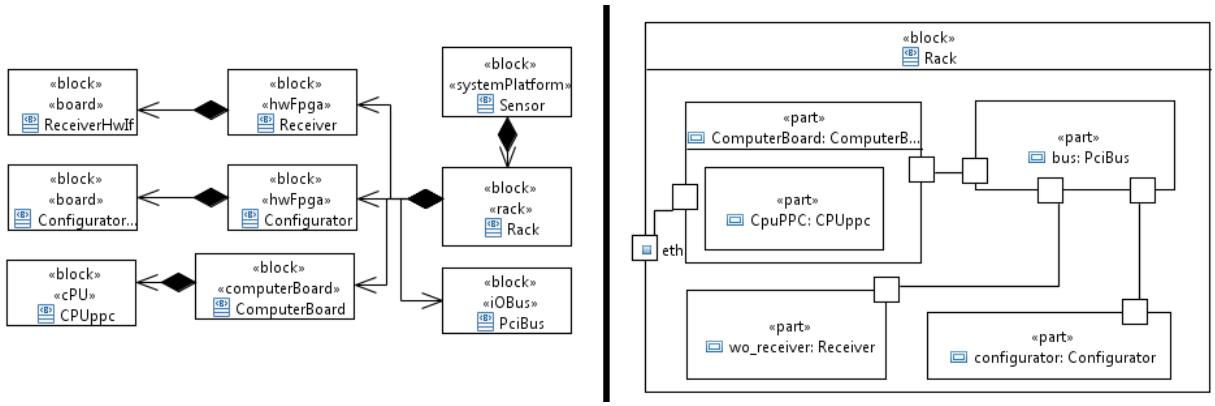

Fig. 12. System Platform BDD and IBD of the RWR

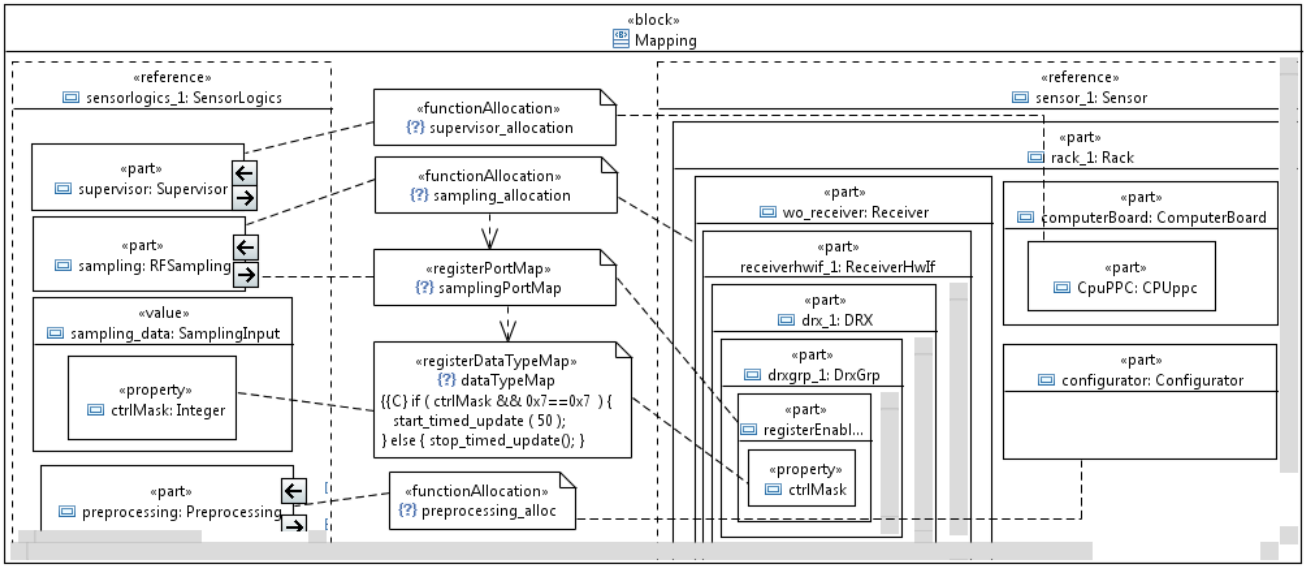

Fig. 13. RWR Mapping Model

[6] F. Herrera, H. Posadas, P.PeÃśil, E.Villar, F.Ferrero, R.Valencia, G.Palermo *The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems* Journal of Systems Architecture

[7] A.Kamppi *Kactus2: Extended IP-XACT metadata based embedded system design environment*

[8] C.Andrey,F.Mallety,A.Mehmood Khan,R.de Simone *Modeling SPIRIT IP-XACT with UML MARTE*

[9] Processor and System-on-Chip Simulation Editors: Leupers, Rainer, Temam, Olivier

[10] Panda, P.R. *A modelling platform supporting multiple design abstractions*, Proceedings of the 14th International Symposium on System Synthesis, 75-80, Montreal, Quebec, Canada.

[11] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto *A Hardware-Software Co-simulator for Embedded System Design and Debugging*, Proceedings of the 14th International Symposium on System Synthesis, 75-80, Montreal, Quebec, Canada.

[12] F. Schirrmeister, M. Meindl and S. Krolikoski *Hardware/Software Interfaces Design for SoC*, Embedded Systems Handbook, Second Edition: Embedded Systems Design and Verification

[13] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, *Metropolis: An Integrated Electronic System Design Environment*, IEEE Computer, vol. 36, no. 4, pp. 45âĂŞ52, April 2003 http://embedded.eecs.berkeley.edu/metropolis/index.html

[14] Edward A. Lee and Sanjit A. Seshia *Introduction to Embedded Systems, A Cyber-Physical Systems Approach , Second Edition*, http://LeeSeshia.org, 2016

[15] The Object Management Group *The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems*, Official web page www.omgmarte.org, 2016.

[16] The QEMU project *main web page* available at http://wiki.qemu.org/

[17] A. Sangiovanni-Vincentelli, *Quo Vadis SLD: Reasoning about the Trends and Challenges of System Level Design*, Proceedings of the IEEE, vol. 95, no. 3, pp. 467 506, March 2007.

[18] *Object Constraint Language, v1.0* , formal/2014-02-03, http://www.omg.org/spec/OCL/2.4

[19] *MOF Model to Text Transformation Language, v1.0*, http://www.omg.org/spec/MOFM2T/1.0/PDF

[20] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, http://www.omg.org/spec/QVT/1.2

[21] *Unified Modelling Language specification*, http://www.omg.org/spec/UML/2.4.1/

[22] *The Eclipse Foundation Project*, http://www.eclipse.org/

[23] The Object Management Group *The SysML standard*, Available at www.omg.org

[24] *IEEE Standard SystemC Language Reference Manual*, IEEE Computer Society, 1666-2005, 31 March, 2006.