Noname manuscript No.
(will be inserted by the editor)

A Model-based approach for the synthesis of software to firmware adapters for use with autogenerated components.

Marco Di Natale · David Perillo ·
Francesco Chirico · Andrea Sindico ·
Alberto Sangiovanni-Vincentelli

Received: date / Accepted: date

Abstract This paper presents the MDE process in use at Elettronica SpA (ELT) for the development of complex embedded systems integrating software and firmware systems. The process is based on the adoption of SysML as the system-level modeling language and the use of Simulink for the refinement of selected subsystems. Implementations are generated automatically for both the software (C++ code) and firmware parts, and communication adapters are automatically generated from SysML using a dedicated profile and open source tools for modeling and code generation.

The process start from a SysML system model, developed according to the platform-based design (PBD) paradigm, in which a functional model of the system is paired to a model of the execution platform. Subsystems are refined as Simulink models or hand coded in C++. An implementation for Simulink models is generated as software code or firmware on FPGA. Based on the SysML system architecture specification, our framework drives the generation of Simulink models with consistent interfaces, allows the automatic generation of the communication code among all subsystems (including the HW-FW interface code). In addition, it provides for the automatic generation of connectors for system-level simulation and of test harnesses and mockups to ease the integration and verification stage. We provide early results on the time savings obtained by using these technologies in the development process.

M. Di Natale
Scuola Superiore S. Anna, TeCIP institute
Tel.: +39 050 882020
E-mail: marco@sssup.it

D. Perillo, F. Chirico and A. Sindico
Elettronica S.p.A.
E-mail: David.Perillo francesco.chirico andrea.sindico@elt.it

A. Sangiovanni-Vincentelli
University of California at Berkeley
E-mail: alberto@berkeley.edu

Keywords System Engineering · Model-Driven Architecture · Model-Based Design · Platform-Based Design · Automatic Code Generation

1 Introduction

The methodology and the process in use at Elettronica SpA for the development of complex distributed Electronic Defence systems [1] benefits from the complementary strengths of domain-specific modeling languages, Model-Driven Architecture (MDA) [4] and Model-Based Development (MBD) [6] and leverages automatic code generation to support the system-level simulation using virtual platforms, the integration of software and firmware on the target and the exchange of test vectors.

Starting from requirement capture, the approach follows the tenets of Platform-Based Design (PBD)[2]. In the architecture design, the SysML models of the system and the subsystems are defined according to the PBD paradigm, separating the functional model from the model of the execution platform. A third model represents the deployment of the functional subsystems onto the computation and communication infrastructure and the HW devices. To define the execution platform and the mapping relationships between the functions and the platform (which defines the model of the software tasks and the network messages, among others) domain-specific SysML extensions are required.

In this work, we focus on several tools and techniques used for the integration, simulation and automatic generation of communication adapters between handwritten C++ components and components generated from Simulink models and implemented in software or firmware (extending the work in [5]).

An example target application is a high speed radar processing system, in which a stream of PDMs (Pulse Descriptor Messages), obtained by sampling RF signals are processed to discover and classify the emitters. To give an idea about the real-time challenges that are typical of these systems, PDM sequences arrive at a rate of 10^6 messages per second and to produce the results within the time constraints, the early processing is partitioned in an FPGA front processor doing frequency analysis and classification and a deinterleaver and supervisor control implemented in SW. The front-processor is controlled by and feeds data to the supervisor.

A general outline of the process stages using the methods and tools presented in this paper is shown in Figure 1 (black arrows represent the way activities are organized in the time domain, gray arrows the data dependencies). The starting point is an architecture-level SysML model of the functions of the system and its component subsystems, derived from the system requirements. The definition of the SysML models is central to our methodology. Following the functional model, another model defines the execution architecture (the hardware and basic software layers, including the OS and the communication protocols), and a third model represents the deployment of the functional subsystems onto the computation and communication infrastructure and the HW devices (these models are defined in the open source tool Papyrus [37] using profiles, as defined in Section 4).

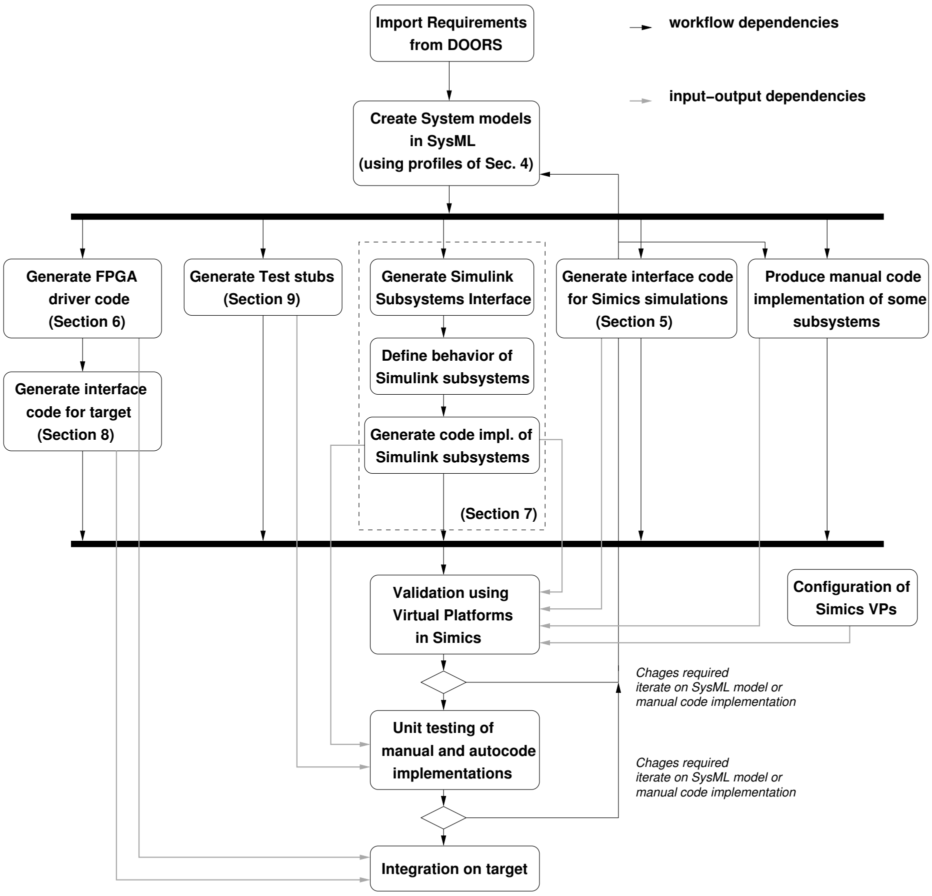
Import Requirements from DOORS
workflow dependencies
input–output dependencies
Create System models in SysML (using profiles of Sec. 4)
Generate FPGA driver code (Section 6)
Generate Test stubs (Section 9)
Generate Simulink Subsystems Interface
Generate interface code for Simics simulations (Section 5)
Produce manual code implementation of some subsystems
Define behavior of Simulink subsystems
Generate interface code for target (Section 8)
Generate code impl. of Simulink subsystems
(Section 7)
Validation using Virtual Platforms in Simics
Configuration of Simics VPs
Chages required iterate on SysML model or manual code implementation
Unit testing of manual and autocode implementations
Chages required iterate on SysML model or manual code implementation
Integration on target

Fig. 1 The process stages supported by the methods and tools in this paper

The SysML models are the reference for manual development of code, or the source of model to code transformations (using the Acceleo open tool under Eclipse) that are aimed at improving the quality and efficiency of the following activities.

- Validation of the functional architecture and the algorithms by virtual prototyping using the Simics platform by Windriver (Section 5)
- Refinement of functionality in Simulink and automatic generation of implementations from Simulink models (Section 7).

- Automatic generation of test stubs based on the SysML specifications (Section 9).
- Automatic generation of the FPGA device drivers and of the communication code for the embedded target (Sections 6 and 8)

As shown in Figure 1, some of the functional subsystems are refined, simulated and prototyped using the Simulink environment [11]. These subsystems execute according to the Simulink (synchronous reactive) semantics. Domain-specific SysML [7] extensions define the execution platform and the mapping relationships between the functions and the platform (which defines the model of the software tasks and the FPGA implementation, among others), the mapping of ports into the programmable HW registers, and the mapping of functional code (including the code generated from Simulink) onto a model of threads and processes and the conditions that trigger the reaction of FPGA-implemented behavior.

Figure 2 depicts the reference SysML project structure used to organize and relate the model elements used in the system design process.

- a SystemRequirements package contains a SysML model of the requirements imported from DOORS;
- an InterfaceDataTypes package contains the SysML description of the Data Types and Interfaces that are provided and required by the system and its parts.
- a System Functional Architecture package contains the functional architecture model, defined as a network of subsystems exchanging data signals.
- an Execution Platform package contains the model of the execution platform with the HW and basic software components, including boards, memories, processing units (cores), network connections, but also device drivers, operating system(s) and middleware.
- a Mapping package contains a SysML model that describes how functional components and behaviors are mapped onto the execution platform, generating the software architecture of tasks and messages. In addition, it defines the mapping of the parameters of the reaction functions implemented in FPGA and the registers of the FPGA. To guarantee independence and reusability as well as visibility of the design entities involved in the mapping, the mapping model imports both the functional and platform models.
- a Test package containing a SysML model defining all the tests by means of which the system requirements shall be verified

The Functional, Platform and Mapping models use stereotypes that extend the standard MARTE profile [8] for real-time and embedded systems. This organization enables the reuse of Interfaces and Data Types and according to the PBD paradigm, allows deploying (by mapping) the same functional model into different platforms of execution.

Leveraging the availability of SysML models developed on the open-source and Eclipse-based Papyrus platform, model-to-text transformations are used to support several stages in the development flow.

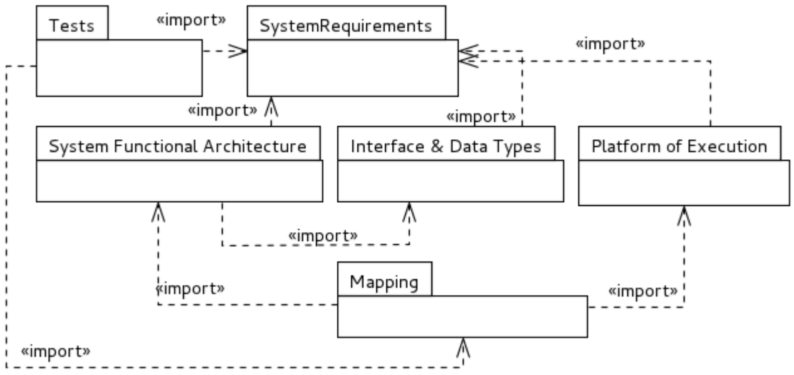

Fig. 2 The structure of SysML projects in ELT

In the early stages of development, when a SW/HW partitioning is evaluated, the models and tools support the automatic generation of interface code for the simulation of the integration of SW and FW using the Simics full-system simulator.

Next, the same models are used to produce the interface specification (ports and port types) of the subsystems to be refined in Simulink. The subsystem is then refined as a Simulink model, validated by simulation, and an implementation for it is generated. For functionality deployed onto a SW thread, a SW implementation is generated (a dedicated C++ class, with an interface defined by the Simulink Coder/Embedded Coder [11] standards). An FPGA implementation is automatically generated for components mapped onto programmable HW.

Our framework generates the communication code that sends and receives data to and from the automatically generated subsystems and those subsystems for which a manual implementation is required. This is done by creating an abstraction layer around each component, with a standard interface that is defined and implemented leveraging the SysML DataFlow port definitions. The (internal) connections between the standard wrapper abstractions and the internal implementations are defined using:

- A standard interface for reading and writing ports for handwritten code.
- A layer that remaps to the standard interface defined by the Mathworks software code generator (for subsystems implemented in Simulink and automatically refined in software).
- A translation to a standard driver interface for reading/writing from/to FPGA registers in the case of an (automatic) firmware implementation.

The (external) connections among the wrapper code abstractions are realized in a different way according to where the component functions (and the wrappers) are allocated for execution.

Finally, when integrating manually coded subsystems and automatically generated subsystems, there is the problem of using a consistent set of test vectors in both environments (Simulink models and C++ code). To support this stage, we automatically generate test harnesses to be used in Simulink simulations (as custom blocks) and when the code is integrated (as C++ stubs) with an XML format for recording, playing and comparing traces on both sides (in the simulation environment, or during code-level unit testing and system integration).

In summary, the main contributions of our work are the following:

- The definition of a SysML profiles that extends MARTE to specify the realization of embedded functionality as software or firmware components. Our profile extensions focus on the specification of a synchronous reactive behavior, the definition of register interfaces for using the FW-implemented functionality, and the mapping relationships that are needed to associate operation parameters to FPGA registers.
- An environment for the generation of communication wrappers towards the automatic generation of implementations from the Simulink environment with deployment onto FPGA (in this case a driver layer is also generated) or in SW. This avoids the need to program code that is mostly tedious, consisting of data marshalling, and automatically selects the mechanisms for data consistency when needed.
- The generation of additional code to help in the simulation of the functionality using the Simics platform and the automatic generation of test stubs that support the exchange of test vectors between the Simulink environment and the code integration stages.
- An implementation that is entirely based on open source tools and standard languages (except, of course, for the integrated Simulink models and the code generated from those).

The main contributions of this new revised version with respect to the work presented at the 2014 MODELS Conference [5] are:

- An extended and improved description of the process, the profiles, and the code generators with additional examples
- A new section describing the generation of code for the simulation of software/firmware integrated functionality under Simics
- A new section describing the automatic generation of Simulink subsystems as a refinement of SysML blocks
- A new section describing the automatic generation of test stubs to exchange test vectors between the Simulink simulation environment and the C++ integration stage.
- A new section describing the impact of the technology described in the paper on the industrial processes.

The organization of the paper is the following. Section 2 provides a quick reference to the technologies, methods, languages and tools that are used in our framework and then an outline to our methods, tools and models for the generation of the adapters (at runtime and for simulation). Section 3 outlines the relationships (and provides a comparison) with previous work in this context.

Section 4 defines all the stereotypes and metamodels used in our development process to represent the design of the system components and the generation of the code for interfacing the SW components with the programmable HW. Secion 5 contains the description of the code generation process for simulation on Virtual Platforms, and Section 6 outlines the generation process for FPGA drivers. Section 7 provides the details of methods and tools for the integration of Simulink components. Section 8 discusses the generation of the communication code, and Section 9 discusses the creation of the support for testing. A discussion on the lessons learnt and the impact from the use of the methodology in the industrial processes are discussed in Section 10. Finally, Section 11 provides the conclusions.

2 Outline of the Process, Models and Code Generation

The main objective of the models and tools presented in this paper is to enforce the consistency of the developed components with respect to a SysML system description and introduce automation in the generation of the code that performs data communication and synchronization among the functional subsystems.

The starting point for our methodology is a SysML model as in the left side of Figure 3. The model is organized according to a layered structure (each layer in a separate package [1]). The *Functional description* consists of a set of SysML blocks communicating through standard and flow ports (top part of the figure). Some of these blocks are identified as subsystems executing according to a synchronous reactive semantics, refined and validated in Simulink.

A separate package in the SysML system model identifies the execution platform for the system, including a model of the execution HW, with computing nodes, boards, cores and FPGAs (bottom-left part of the figure). Each core is associated with the operating system managing the execution of the software processes and threads residing on it. Finally, a third package defines the allocation of the functional subsystems onto the execution platform. This layer defines the model of the software threads and processes, of the communication messages and the allocation of functionality onto threads (for software implementations) or programmable HW.

Following the system-level architecture description in SysML, components are designed, refined, and implemented using different methods and technologies. Components that define complex algorithms or control laws are modeled, simulated, and verified as Simulink models. For these components, an implementation path making use of automatic generation tools is used. Other components are designed in UML and then refined as manually written C++ code.

Two software layers, generated automatically from the SysML model description provide for the interaction between the subsystem functionality and

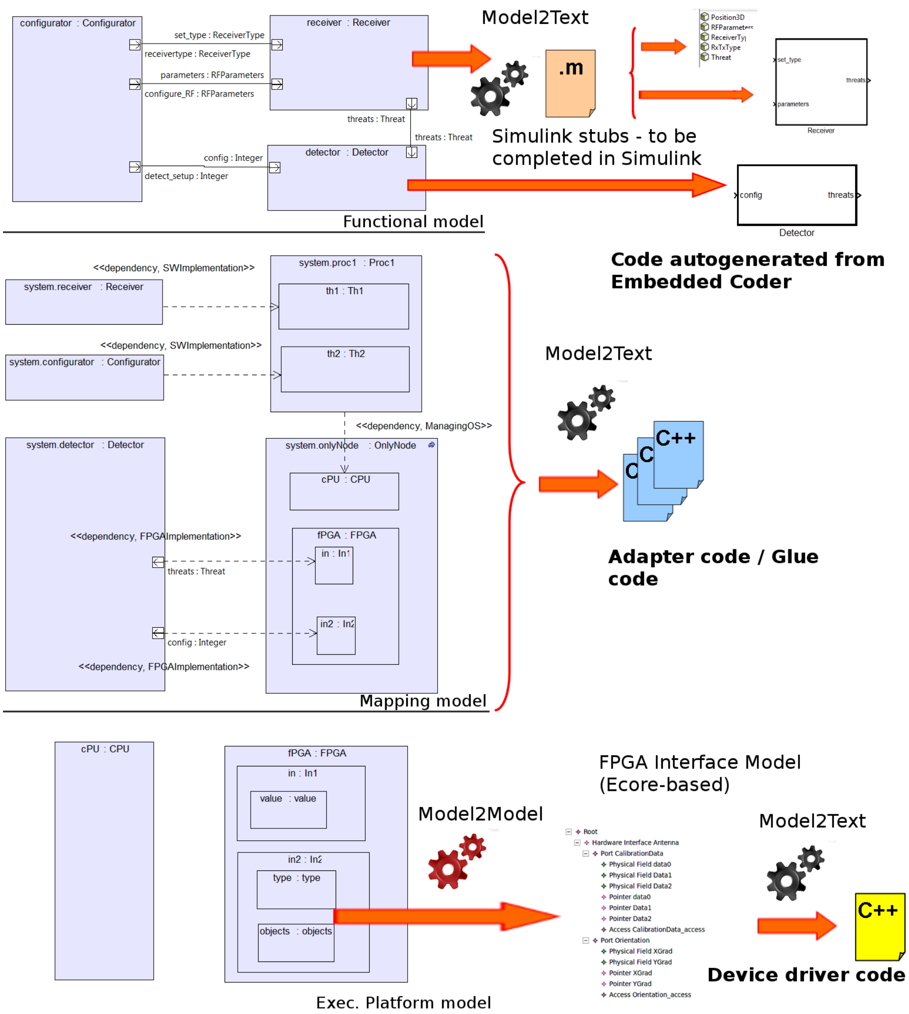

Fig. 3 The generated wrappers provide for the communication among subsystems

the FPGA implementations (access to the HW platform, as described in Section 6) and for the communication and interactions among functional components. The communication among the data ports of all functional subsystems is realized through code automatically generated from the SysML Mapping layer and consisting of a number of software wrappers that provide an API for accessing the data ports specified in SysML with the correct type information (shown in light color, in Figure 3). These wrappers translate from a standard interface for the access to ports (directly used by hand-written components) to the standard interface to the SW functions automatically generated by Simulink and/or the driver functions automatically generated for the access to FPGA functionality in the case of functionality mapped onto programmable HW.

For all the subsystems to be refined in Simulink, a Simulink subsystem boundary specification is automatically generated from the SysML model (as described in [18] and outlined in Section 7). The subsystem is refined, with the model of its internal behavior, validated, and finally an implementation for it is generated. For subsystems executing as software on a core, the corresponding code is produced (top-right side of Figure 3) by the Simulink Coder/Embedded coder tools. For firmware implementations, the Mathworks tools generate an RTL (Register Transfer Language) description for programming the FPGA.

The first layer of generated SW consists of a high-level FPGA driver, created from the SysML model of the FPGA interface, as described in Section 6. This driver-level code provides functions for accessing FPGA-implemented functionality with input and output ports defined on complex data objects (rather than individual registers) and provides for buffering and synchronization. This layer operates on top of a lower-level driver which is manually written and implementing an interface of simple RAM-mapped register reads and writes.

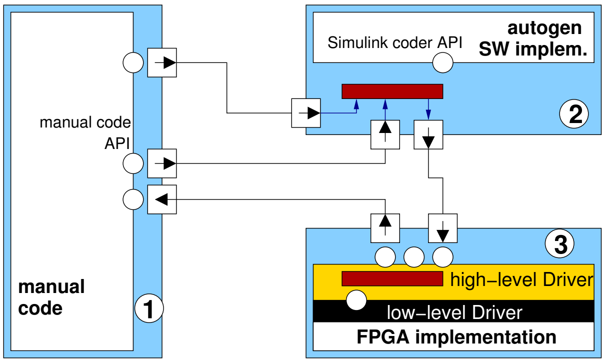

Fig. 4 The generated wrappers provide for the communication among subsystems

The communication among the data ports of all functional subsystems is realized through additional code, automatically generated from the SysML Mapping layer. The generated code consists of a number of software wrappers that provide an API for accessing the data ports specified in SysML with the correct type information (shown in Figure 4). These wrappers translate from a standard interface for the access to ports (that is directly used by hand-written components) to the standard interface to the SW functions automatically generated by Simulink and/or the driver functions automatically generated for the access to FPGA functionality in the case of functionality mapped onto programmable HW.

3 Related work

The match of a functional and execution architecture is advocated by many in the academic community (examples are the Y-cycle [13] and the Platform-Based Design PBD [2]) and in the industry (the AUTOSAR automotive standard is probably the most relevant recent example) as a way of obtaining modularity and separation of concerns between functional specifications and their implementation on a target platform. The OMG [3] and the MDE similarly propose a staged development in which a PIM is transformed into a Platform Specific Model (PSM) by means of a Platform Definition Model (PDM) [14].

The development of a platform model for (possibly large and distributed) embedded systems and the modeling of concurrent systems with resource managers (schedulers) requires domain-specific concepts. The OMG MARTE [8] standard is very general, rooted on UML/SysML and supported by several tools. MARTE has been applied to several use cases, most recently on automotive projects [16]. However, because of the complexity and the variety of modeling concepts it has to support, MARTE can still be considered work in progress, being constantly evaluated [15] and subject to future extensions. In particular, MARTE does not provide enough detail for the description of the HW interface registers that provide access to IO features or to the functionality of programmable HW, and surely does not provde enough detail to allow for the automatic generation of driver or communication code. This was the motivation for our proposed SysML language extensions.

Several other domain-specific languages and architecture description languages of course exist, such as, for example EAST-ADL [31] and the DoD Architectural Framework (DoDAF) [32]. For the details of the HW register interfaces, several languages exist, including the widespread SystemC [39]. Also, domain-specific languages have been proposed for this purpose (one example in [40]). However, we are not interested in the proposal for yet another DSL, but rather to enable the modeling of low-level HW features in the SysML language and in connection with open source tools (such as Papyrus). Also, we want to easily bridge the gap with high-level architecture descriptions and Simulink models, which we believe is easier done in SysML rather than, for example, in SystemC.

Several other authors [20] acknowledge that future trends in model engineering will encompass the definition of integrated design flows exploiting complementarities between UML or SysML and Matlab/Simulink, although the combination of the two models is affected by the fact that Simulink lacks a publicly accessible meta-model [20]. Work on the integration of UML and synchronous reactive languages [21] has been performed in the context of the Esterel language (supported by the commercial SCADE tool), for which transformation rules and specialized profiles have been proposed to ease integration with UML models [22]. With respect to the general subject of model-to-model transformations and heterogeneous models integration, several approaches, methods, tools and case studies have been proposed. Some proposed methods, such as the GME framework [23] and Metropolis [24]) consist of the use of a general meta-model as an intermediate target for the model integration.

Our work spans also other topics, including the modeling of FPGA (interfaces) and the automatic generation of stubs for simulation and testing. However, our models are not aimed at the definition of the FPGA behavior for the purpose of performance evaluation or the selection of implementation alternatives (such as, for example in [28] or [29]), but rather at enabling the automatic generation of driver and communication code starting from standard SysML models (we do not use Domain Specific Languages, or DSLs) and using open source tools.

A large number of works deal with the general subject of integration of heterogeneous models. Examples are the CyPhy/META Toolchain at Vanderbilt [19] and the work on multiparadigm modeling (a general discussion in [17]). In both cases, emphasis is placed on the role of domain-specific languages and model transformations in the general context of large and distributed Cyber-Physical systems. With respect to model-to-model transformations, our aim is very focused to the translation of SysML models into driver code or communication code that is suitable for connection with automatically generated Simulink components.

Other groups and projects [25] have developed the concept of studying the conditions for the interface compatibility between heterogeneous models. Examples of formalisms developed to study compatibility conditions between different Models of Computation are the Interface Automata [26] and the Tagged Signal Language [27]. In our case, we are not trying to cover a large set of possible models of computation, but we are only interested in supporting a connection with a restriction of the synchronous reactive behavior that is allowed by Simulink/Stateflow models [41] [42].

Other efforts have been dedicated to the objective of providing automation for the integration of heterogeneous models and components for simulation. An approach for the automatic synthesis of adapters for simulation units developed with heterogeneous tools and languages is described in [30]. The approach makes use of custom DSLs. More recently, the FMI/FMU interoperability standard [33] aims at obtaining the same objective by providing for a standardized code-level API. The FMU standard is very promising for the purpose of cosimulation of large CPS systems, but is today mostly aimed at the modeling of physical systems, rether than the modeling and simulation of HW implemented functionality. In addition, we are mostly concerned about the automatic generation of glue code, and this objective is currently not within the scope of the FMU/FMI standard.

4 SysML Profiles for PBD

We defined SysML profiles to express concepts that are required for our scope (and of general use to specify resources and complex embedded systems designs). Overall, the stereotype definitions contained in these profiles follow the general organization of *Functional*, *Platform* and *Mapping* models.

4.1 Functional modeling

The functional model contains the definition of the subsystems, at some level of refinement of the system functional architecture. Each subsystem processes input signals and produces outputs, according to a port-based interface. The profiles that apply to the functional model must support the code generation stage allowing the identification of the subsystems with a synchronous execution semantics. The profile *FunctionalModels* defines the stereotypes.
`<<FunctionalSystem>>` applies to Block, and identifies the root block (or system) in the functional model.
`<<SRSubsystem>>` applies to Block and defines a subsystem that processes signals according to a synchronous reactive semantics, that is where the functional behavior consists of a single processing stage or activity (typically activated on a periodic time base), which synchronously samples all inputs, reads the internal state and updates the subsystem state and its output. In the case of a firmware implementation, the reaction can also be triggered by a signal, stereotyped as `<<SRReact>>`. A synchronous subsystem should have one operation stereotyped as `<<SRStep>>`. This is the public operation take computes the system reaction (update of the outputs and internal state).
`<<SimulinkSubsystem>>` specializes SRSubsystem and defines a subsystem that is modelled and defined according to the Simulink semantics.

4.2 Platform modeling

The execution platform and the mapping models define the structure of the HW and SW architecture that supports the execution of the functional model.

The execution platform is defined in a package called *PlatformModels*. Blocks represent hardware components at different levels of granularity, but also classes of basic software, including device drivers, middleware classes and operating system modules.

The MARTE profile provides several concepts that can be leveraged for the definition of the hardware and software platform. For our code generation, we need to identify what subsystems are implemented in SW, running on a core and using services provided by a given operating system, and what subsystems are implemented on programmable HW (FPGA). Also, we need a model describing the register interface of the FPGA, offering not only the register abstraction but also a higher level description of a hardware ”port”.

For the definition of processors, MARTE offers the stereotype definition of «HwProcessor» and the stereotype «HwPLD» for the definition of FPGAs and FPGA interface registers. The modeling elements to specify the register

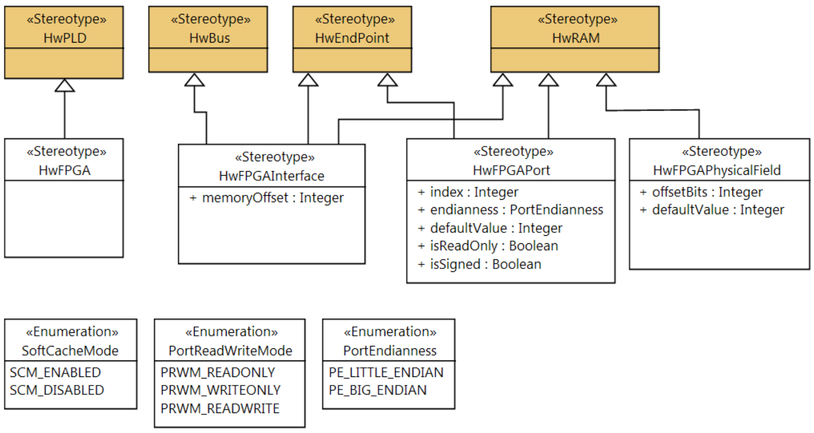

Fig. 5 A SysML profile for the description of Interfaces to programmable HW

interface of an FPGA, however, are not easily found. For this reason, we defined our taxonomy of stereotypes for FPGA components and interfaces. Programmable hardware components are derived as a refinement of the MARTE «HwPLD» (Figure 5). The hardware interface is represented by a stereotype «HwFPGAInterface». The registers in the addressing space can be grouped in contiguous sets intended to be accessed for a homogeneous set of data/information and called «HwFPGAPort». The stereotyped definition of FPGA Port objects is obtained from the SysML Block (not the SysML port, because it is itself the composition of other objects and the Port entity in SysML cannot be a composite of other ports). A Hardware interface block typically consists of a number of FPGA Ports, in turn composed by atomic data items denominated FPGA Physical Field.
The main stereotypes with their properties (Figure 5) are:
`<<HwFPGA>>` refines «HwPLD» and defines an FPGA component.
`<<HwFPGAInterface>>` refines the MARTE stereotypes «HwBus» (to define address and data bus widths), «HwEndPoint» and «HwRAM» (for addressing modes, memory size), which in turn apply to Block. It is used for the description of the Interface to a programmable HW component (the component itself is identified by its interface). It uses the MARTE properties

`addressWidth` (from «HwBus», representing the address bus width).
`wordWidth` (from «HwBus», representing the data bus width). In both cases, legal values are 8, 16, 32, and 64.

and defines the additional property

`memoryOffset` a long representing the physical address of the first word in the programmable HW address space.

`<<HwFPGAPort>>` refines «HwEndPoint» and «HwRAM» (which apply to Block), from which the property `memorySize` defining the port size (the number of bits required for storing the information carried by the entire Port) is inherited. It is used to identify structured information that the programmable hardware will read or write as a whole and includes the properties

`index` an unsigned integer representing the position of the port among those in the set of the HW Interface.
`defaultValue` if the Port does not contain any PhysicalField, the property contains the default value associated with the information.
`isReadOnly` a boolean: true in case the information cannot be modified.
`endianness` an enumeration.
`isSigned` a boolean: true if the information is to be interpreted as a 2's complement.

`<<HwFPGAPhysicalField>>` refines «HwRAM». It defines a hardware register representing a field of information in a Port. It includes the properties

`offsetBits` the address offset for the first word in the PhysicalField.
`defaultValue` the default value for the corresponding information.

For the software part of the platform, we are interested in defining the Operating system running on a given Processor. In this case, MARTE states that *"Operating systems may be represented through properties of the execution platform or, requiring significantly more detail, modeled as software components"*. For the second option, however, no stereotypes are offered. Therefore, we defined our own stereotype `<<SwOperatingSystem>>`, which only has an enumerated property with the OS name. In our code generation (described in the next section) the operating system information is only used to check whether a communication implementation using the boost library is possible.

4.3 Mapping model

The profile *Mapping* defines the stereotypes of general use for the mapping of functions onto a platform, including the stereotypes for the mapping of functions onto a SW architecture of processes and threads and the messaging.

For our code generation, we are interested in knowing whether the communication between two functional subsystems is implemented as intrathread, interthread, interprocess or remote. Therefore, we need to identify *Processes* and *Threads* in the software implementation model. MARTE provides the stereotype *SwConcurrentResource*, which is cumbersome and possibly confusing. The `<<SwSchedulableResource>>` stereotype is recommended for the well-known concepts of *Process* (which should also inherit from `<<MemoryPartition>>`), *Thread*, or *Task* and comes with 39(!) stereotype attributes defining each and every aspect related to its management.
Our mapping profile contains the definition of the following stereotypes:
`<<MappedSystem>>`, applies to Block, and identifies the root block of the mapping model. The Mapping model includes a functional model, a platform model, a process model and a message model.

`<<ProcessModel>>` applies to Block, and identifies the root block of the model of all the processes in the system. A ProcessModel can recursively contain a ProcessModel or a set of Processes
`<<Process>>` applies to Block and identifies a Process or a SW application. A Process may (should) contain Threads.
`<<Thread>>` applies to Block and identifies a concurrent unit of execution.

In addition, we had to define deployment relations. We built on the MARTE «Allocation» stereotype to define an implementation mapping between the functional layer subsystems and the platform. The provided stereotypes are:
`<<SWdeployment>>` refines Allocation to specify an implementation of a functional subsystem (all the operations and actions in it) by a thread.
`<<FPGAdeployment>>` refines Allocation to specify an Implementation of a functional subsystem (all operations and actions in it) by an FPGA.
`<<AutoGenerated>>` defines a deployment (an implementation) for which automatic generation is supported.
`<<ManagingOS>>` refines Allocation to specify a mapping relationship between a process and the real-time operating system managing it.

4.4 An Example

Figure shows the BDD and IBD views (Block Definition Diagram and Internal Block Diagram, standard SysML views) of a very simple example of functional model, with three subsystems communicating through SysML flow ports: a *Configurator*, a *Detector* and a *Receiver*. The functional model is defined in the package *FunctionalModels*. The types that apply to the flow ports are defined in the package *InterfaceDataTypes*. The model is only meant to provide an example of communication scenarios and is void of any functional content.

The example is only for describing the role of each stereotype in the definition of the models and the mapping of functionality onto platform. A more concrete snapshot of (a portion of) the actual application is shown in Figure 7.

The sample platform model for the functional example is shown in Figure 8, with a single node containing a CPU and an FPGA, which has in turn one interface with three ports. For one of the ports, the details of its physical fields are provided. Finally, the mapping model defines how the functional model is realized on the execution platform. This mapping information is in the package MappingModels to allow full independence and reusability of the functional and platform parts.

The mapping model information for our example is represented in an IBD diagram as in Figure 9. The Detector and Configurator subsystem instances in the functional system model are deployed as software implementations onto

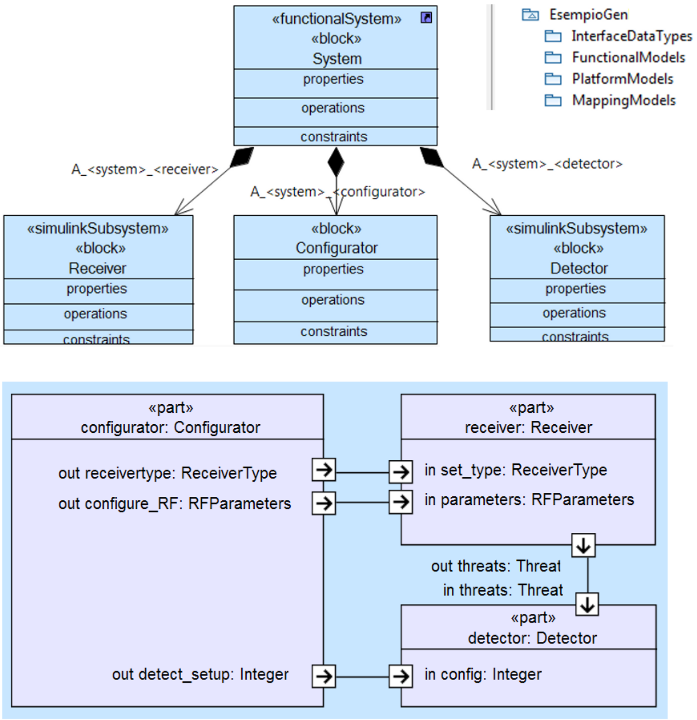

Fig. 6 The ibd showing the port connections for the a sample model

two threads (Thread1, and Thread2, defined in a Process model package, which is part of the mapping model), which are in turn part of a Process Process1, executing on the CPU of our node. The Receiver part is mapped as an FPGA deployment onto the node FPGA. The interface ports of this block are implemented on an FPGA interface. The mapping between ports with primitive types on the functional side and implemented by a single register (no physical field) on the hardware side can be defined directly. For ports with structured types, each single field of the port type must be mapped onto a register (physical field) of the FPGA. This is performed by exposing the internal properties of the structured type (the imported reference to the port type) and building mapping relationships between each type property and a physical field. All mapping relationships (except those originating from the process/thread model) are defined through a stereotyped constraint, which is itself part of the mapping model. This allows to keep the functional and platform models completely independent, while at the same time, providing the necessary information for the code generation stage.

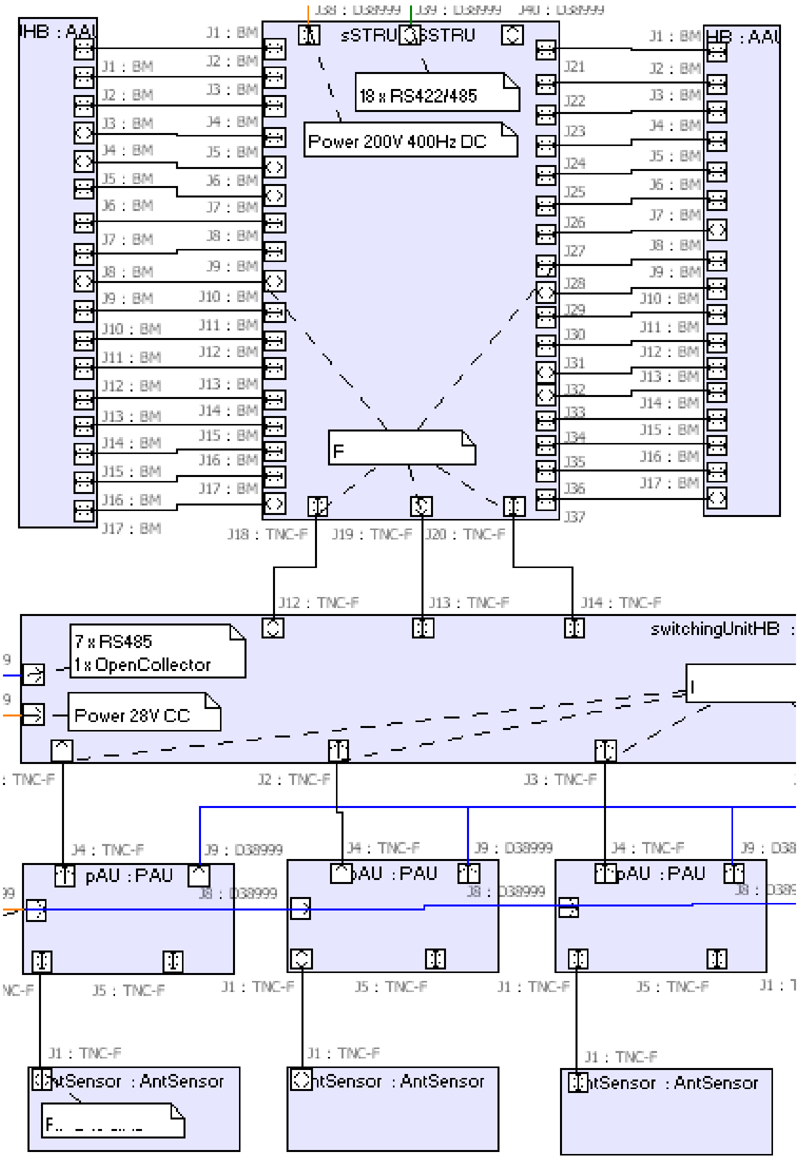

Fig. 7 A portion of the Internal Block Diagram of the EW Integrated System.

5 Interface code generation for the integration of functionality in Simics simulations

To ease the detection of interface issues and to verify the integration of the software and firmware functionality before the final execution architecture is available, the Simics full-system simulator by WindRiver [38] is used in ELT.

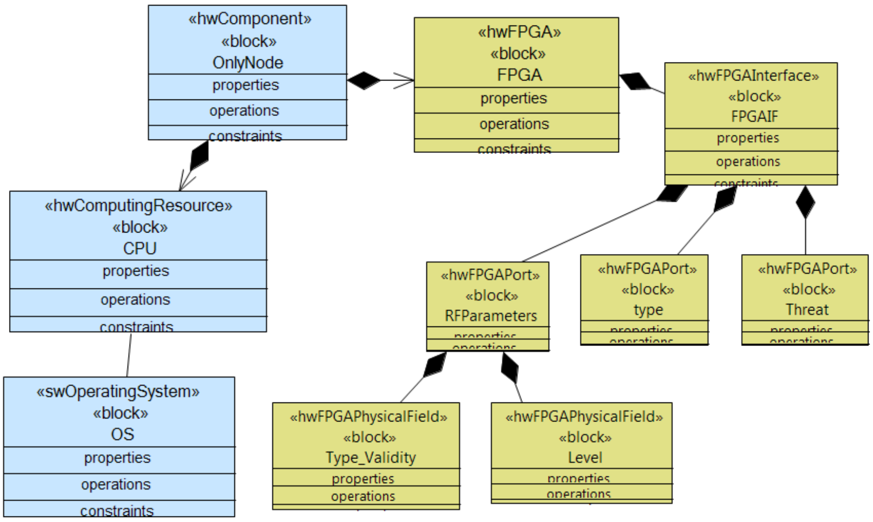

Fig. 8 The platform model for the example

Simics allows to verify the execution of the system software on a virtual platform simulated on a workstation. The platform components can be COTS components (processors, buses, memories) extracted from the Simics library, or custom components defined by means of an interface and behavior specification. When a custom (programmable HW) component is defined, the interface specification defines the registers that are used to communicate with the custom component, with their memory addresses on the bus as base address plus offset. The behavioral part describes the computation or reaction that is implemented in hardware or firmware and modifies the content of the interface registers (as the result of the computation) and, possibly, the internal state of the device.

In Simics, the interface model is typically defined in DML (short for Device Modeling Language). In DML the user specifies a register interface with the width, address and offset of all the register fields. Figure 10 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 of the operation. The device is loaded in a memory space at a specific address from the Simics console.

In the example of the figure, the behavior is represented as a DML function that is invoked upon a read opration and returns the value that is going to be read at simulation time by the software from the register at offset 0x0000. Simics allows to define callbacks upon read and/or write operations on the set of interface registers. These callbacks can be implemented not only in DML, but also in C or SystemC.

The read/write callback mechanism is used in our framework to define a pattern for the integration of software and firmware with automatically gen-

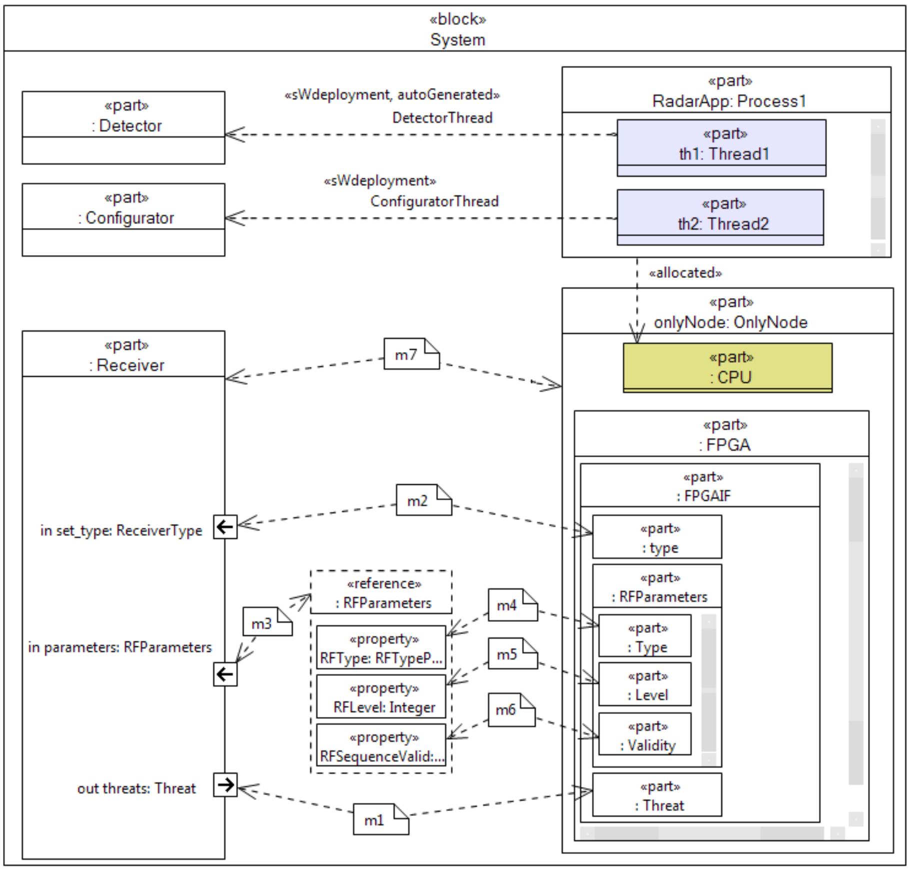

Fig. 9 The mapping model for the example

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

Fig. 10 An example DML code for the specification of a register interface and firmware behavior

erated adapters according to the SysML specification. The functionality to be realized in firmware is implemented as an equivalent C function, according to a reaction pattern. This code needs to be handwritten, or produced by means of automatic generation by third party tools (such as Simulink).

The (unique) reaction function to be implemented in firmware is identified in the functional model by a the stereotype `<<SRStep>>`. The parameters of the operation are mapped onto the interface registers specified in the platform model. In addition, the functional model contains the specification of a signal that triggers the execution of the function. This signal is mapped onto the set of FPGA registers by means of a specification block that defines the register write sequence that triggers the execution of the firmware behavior.

Based on the SysML specification, a set of Acceleo templates generates:

- the DML code for the specification of the register interface according to the SysML platform model. For each input register that appears in the trigger signal specification, a write callback is defined that invokes a predefined condition checker function.
- the body of the condition checker according to the constraint block specification. The code of the function verifies the content of all the interface registers and conditionally invokes the reaction function for the block mapped on the programmable HW.

The reaction function is written by hand and needs to comply with the signature in the functional model. This framework allows to perform the simulations by reusing all the program code that is deployed on the actual target, including the code possibly automatically generated from Simulink and the code for the device drivers that are writing into and reading from the interface registers (as explained in the following sections).

6 Generation of the FPGA driver code

The communication with a functionality implemented by programmable HW is structured in layers. The firmware function is accessed through a set of control and data registers implemented on the FPGA and mapped in the memory space. Access to the FPGA registers is provided by a low-level driver, which is manually developed and provides basic read and write functions, according to an interface defined as *IBusAccess* and used by the upper layers. Read and write operations are overloaded according to the width of the data bus. For example, for a 64-bit data bus the functions are simply:

```
Read(char*address, unsigned long &in)
Write(char*address, unsigned long &in)
```

On top of this driver, an upper layer with set of higher-level operations is automatically generated. This layer maps application objects with structured data types onto elementary (bus-width) data registers and provides for caching, fragmentation and reassembly, notification of events and endianness conversions.

This higher-level layer is automatically generated from the SysML model of the FPGA Interface with a model-to-text transformation, from the Platform model into a set of C++ classes.

The generated code has the following structure. Two classes (in a pair of .ccp and .h files) are generated for the device.

A class called *NAME_HW_INTERFACE*`Cacheddriver` implementing a cache for all FPGA registers. The purpose of the cache class is to save time upon reading and writing into the HW only when values change (commands are requested).
A class *NAME_HW_INTERFACE*`driver` providing port-level access functions for reads and writes. for each port the following operations are generated:

`Get(&t`*NOME_PORT*`_x values)`, to read values from the Port (registers).
`Set(&t`*NOME_PORT*`_x values)`, to write value into all the HW registers associated with the port.
`Reset`*NOME_PORT*`_x ()` to reset the values of all the registers associated with the port to their default values.

In addition, a class constructor is generated, with a reference to the low-level driver functions implementing the reads and writes on the physical registers.

```
NAME_HW_INTERFACEdriver (IBusAccess* bit8,
  IBusAccess* bit16, IBusAccess* bit24, IBusAccess* bit32,
  IBusAccess* bit64, unsigned int offset = 0)
```

A file *NAME_HW_INTERFACE*`Types.h` is created with the definition of all the structured types that are required for the Ports.

7 Refinement of Simulink Subsystems

A top-down development flow makes use of transformations from the SysML `<<SRSubsystem>>` block into the specification of a Simulink Subsystem, complete with its ports and datatype specifications as *Bus Objects* (the tool-specific type/class declarations). The Simulink subsystem is then further developed in the Mathworks environment by modeling its internal behavior. More often, however, the functionality to be developed has already been prototyped in Simulink and a reverse transformation generates a SysML block. In this case, a MATLAB script generates an XML file compliant with an Ecore metamodel developed *ad hoc* for the representation of Simulink subsystems in EMF. A QVT model-to-model transformation then generates the SysML block from the Ecore model.

An Acceleo module transforms the SysML block and generates a Matlab script `SubsystemName_Simulink_types.m` that creates in the Matlab environment a set of Bus Object specifications mirroring the definitions of the data types in the SysML model; one or more files with name `EnumeratedTypeName.m` for each enumerated type in the SysML type specifications that apply to the subsystem ports; and a script `SubsystemName_Simulink.m` that generates the boundary of the subsystem with its ports (as described in [12], an example is shown in Figure 11). The subsystem is then defined internally and simulated, until its behavior is defined in a satisfactory way.

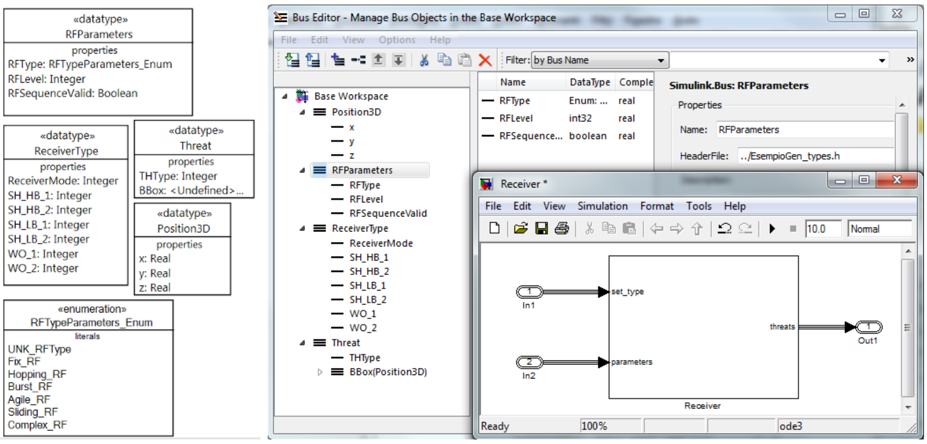

Fig. 11 Generating simulink subsystems and types from SysML

The generated FPGA implementation communicates with the other subsystems using a set of memory-mapped registers, accessed using the drivers described in the previous section. The C++ generated code follows the conventions of the code generator: for each subsystem, a class is generated with name *SubsystemName*`ModelClass`. The class has operations for the subsystem initialization and (if required) termination, and a `step` operation for the runtime evaluation of the block outputs given the inputs and the state. The Simulink Coder conventions define how the interface ports translate into arguments of the `step` and allows to define the data types in an external (user provided) file. Listing 1 shows the code generated for the Receiver subsystem in our example.

Listing 1 Code generated for the Receiver subsystem

```
class ReceiverModelClass {
 public:
  void initialize();             /* model initialize function */
  /* model step function */
  void step(const ReceiverType &arg_In1,
              const RFParameters &arg_In2,
              Threat *arg_Out1);
  ReceiverModelClass();          /* Constructor */
  ~ReceiverModelClass();         /* Destructor */
}
```

8 Subsystem deployment and communication code generation

Some of the subsystems defined in the SysML functional model are refined in Simulink and an implementation is automatically generated for them. Other subsystems are developed as hand-written code or implemented by purposely designed HW or firmware. The software infrastructure that provides communication and synchronization among blocks, and realized as port and subsystem wrappers is automatically generated from the SysML model using Acceleo transformations that create application-specific classes (and objects) refining library classes. For the communication between hand-written C++ code implementing functional subsystems and subsystems generated in SW from Simulink model, Acceleo scripts automatically generate the wrappers that provide the marshalling of parameters to the variables implementing the input ports and retrieving the data from the output port variables. The Step function implementing the subsystem runtime behavior is invoked in the context of a software thread executing at the appropriate rate.

For the case of communication between firmware (FPGA) implementations and software subsystems, the read and write interface operating on the shared registers and memory locations is automatically generated based on the information provided in the SysML mapping model, defining the position of data signals in memory and/or HW registers.

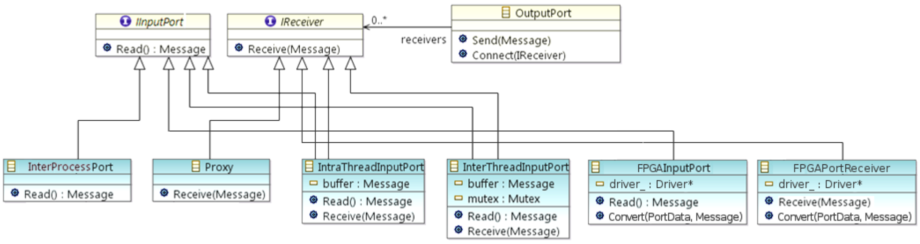

Fig. 12 Hierarchy of classes for subsystems ports.

The class hierarchy defining the subsystem wrappers is simple. A virtual base class *SubsystemWrapper* is at the root of the hierarchy. Two classes are derived from it: *SubsystemSimulinkWrapper*, the base class for subsystems modelled by Simulink, and *SubsystemCppWrapper*, the base class for subsystems developed in C++ by hand (FPGA-implemented components do not have a wrapper). These classes are statically defined in a library. The Acceleo scripts define subsystem-specific classes derived from them. The communication between subsystems takes place through instances of port classes, whose hierarchy is depicted in Figure 12. The following template classes are defined:

`OutputPort<Message>` (base class for output ports): a concrete class implementing the following methods:

  - `Send(Message)`, to send data (at runtime) to the connected blocks,
  - `Connect(IReceiver)`, invoked at initialization time to connect the port to an instance of the *IReceiver* class in a corresponding input port or stub (for interprocess communication).
- `IInputPort<Message>` (base class for input ports): an abstract class defining the method:
  - `Read(Message)`, to read the data received on the port from the subsystem methods.
- `IReceiver<Message>`: an abstract class defining the method:
  - `Receive(Message)`, to receive data from an OutputPort.

Listing 2 shows the code of the `OutputPort` class. The Send method forwards the data to all connected IReceiver(s) that provide the data buffers. Concrete instances of input ports inherit from *IInputPort*. They also inherit from *IReceiver* when connected to output ports in the same process. *IntraThreadInputPort* and *InterThreadInputPort* inherit from the abstract interfaces *IInputPort* and *IReceiver*, allowing direct transmission of the Message data between different subsystems in the same process. Both store the Message data in an instance variable upon reception. The class *InterThreadInputPort* provides thread-safe access to its internal buffer using the protection method provided by the OS on the CPU hosting the process (currently only *boost mutexes* are supported).

Listing 2 Code of the Output port class

```
template<typename Message>
class OutputPort
{
public:
  OutputPort() {}
  virtual ~OutputPort() {}
  virtual void Send(const Message &message) {
    for (typename ReceiversVector::const_iterator
         i=receivers_.begin();
         i != receivers_.end(); ++i)
        (*i)->Receive(message);  }
  virtual void Connect(IReceiver<Message> *receiver) {
    receivers_.push_back(receiver);  }
protected:
  typedef std::vector<IReceiver<Message>*> ReceiversVector;
  ReceiversVector receivers_;
};
```

The separation between *IReceiver* and *IInputPort* is necessary when the output port and the connected input port belong to components mapped into different processes.

In this case, the *OutputPort* instance will be connected to a proxy object derived from *IReceiver* (living in the same process), which will then implement a (currently socket-based) inter-process communication to send data to the matching *IInputPort* instance on the other process. In Figure 12, this is represented by the classes *InterProcessInputPort*, derived from *IInputPort*, and *Proxy*, derived from *IReceiver*. This allows the users to ignore the details of specific implementations and only rely on the Send/Received methods with maximum portability.

8.1 C++ wrapper

The classes generated for the communication of **C++ hand-written** subsystems inherit from *SubsystemCppWrapper* and provide only the concrete definition of the communication ports and read/write operations for accessing them. The behavior of the subsystem is then manually coded (the listing of the generated code is quite straightforward and omitted for space reasons). Listing 3 shows the code generated for our example C++ subsystem Configurator.

Listing 3 Code generated for the Configurator subsystem

```
class SubsystemConfigurator : public SubsystemCppWrapper {
public:
    SubsystemConfigurator();
    OutputPort<ReceiverType> *getReceivertype();
    OutputPort<RFParameters> *getConfigure_RF();
    OutputPort<Integer> *getDetect_setup();
private:
    OutputPort<ReceiverType> receivertype_;
    OutputPort<RFParameters> configure_RF_;
    OutputPort<Integer> detect_setup_;
};
```

8.2 Simulink wrapper

The **Simulink wrapper** instantiates the ports to communicate with the other subsystems and provides two methods `Init` and `Step`, that encapsulate the corresponding automatically generated methods.

The SubsystemReceiver class generates for our example (shown in Listing 4) defines the `parameters` and `set_type` ports. These ports receive input from the Configurator subsystem, which is mapped to another thread. Hence, their implementation is thread-safe. The user has the responsibility of writing

Listing 4 Code generated for the Receiver subsystem (of Simulink type)

```
class SubsystemReceiver : public SubsystemSimulinkWrapper {
public:
    SubsystemReceiver();
    virtual void Init();
    virtual void Step();
    InterThreadInputPort<ReceiverType> *getSet_type();
    InterThreadInputPort<RFParameters> *getParameters();
    OutputPort<Threat> *getThreats();
private:
    InterThreadInputPort<ReceiverType> set_type_;
    InterThreadInputPort<RFParameters> parameters_;
    OutputPort<Threat> threats_;
    ReceiverModelClass simulink_receiver_;
};
...
void SubsystemReceiver::Init(){
    simulink_receiver_.initialize();
}
void SubsystemReceiver::Step() {
    ReceiverType input1 = set_type_.Read();
    RFParameters input2 = parameters_.Read();
    Threat output1;
    simulink_receiver_.step(input1, input2, &output1);
    threats_.Send(output1);
}
```

the periodic thread that invokes the `Step` method of the generated subsystem wrapper class after the `Send` methods are called for all the output ports connected to the input ports of the subsystem block.

8.3 FPGA port wrapper

The **FPGA communication code** consists of port and receiver wrappers that encapsulate the high level driver functions and connect to the input and output ports of the components communicating with an FPGA subsystem (Listing 5). The library code consists of a base class *FPGAInputPort* used to derive the Acceleo-generated classes implementing the input ports of a SW component connected to an FPGA subsystem output port.

When reading, the `Read` operation forwards the request to a `Get` operation from the FPGA driver port. For FPGA input ports, a dedicated Receiver is provided. A Send to a port connected to an FPGA input results in a `Set` on the FPGA driver. In both cases, the Acceleo-generated code mainly consists in overriding the definition of the `Convert` operation, translating the fields of the data port type into the PhysicalFields of the FPGA physical port, according to the mapping specified in the SysML model.

Listing 5 library classes for FPGA ports and receivers

```
template<typename Message, class Driver, typename PortData>
class FPGAInputPort : public IInputPort<Message> {
public:
  FPGAInputPort(Driver *driver) : driver_(driver) {}
  virtual Message Read() {
    PortData data; Message message; driver_->Get(data);
    Convert(data, message);
    return message;
  }
protected:
  virtual void Convert(const PortData &data, Message &msg)=0;
private:
  Driver *driver_;
};
...
template<typename Message, class Driver, typename PortData>
class FPGAPortReceiver : public IReceiver<Message> {
public:
  explicit FPGAPortReceiver(Driver *driver):driver_(driver) {}
  void Receive(const Message &message) {
    PortData data; Convert(message, data);
    driver_->Set(data);
  }
... };
```

8.4 Initialization code and port connections

Finally, an additional code section is generated for each process to perform the **initialization** of all the components in the threads/processes and connecting their ports. A reference to the FPGA driver managing the FPGA registers accessed by the subsystems in the process is passed to the reading components and a receiver class is defined for each input FPGA port. The following Listing 6 shows (part of) the initialization code for our example.

9 Shared testing environment

Another set of Acceleo scripts (reusing many templates from the previous code generation stage) is used to automatically generate two testing stubs for each subsystem to be refined in Simulink and two wrappers to be used on the generated code (in the software integration stage). The two stubs are used as the code implementation of two custom blocks that are used in Simulink simulation runs, one acting as a gateway on the input data, the other on the output data (as shown in Figure 13). The two code wrappers/stubs are used in a C++ programming environment to test the code implementation of the SUT.

Listing 6 Initialization code for the example

```
class ThreatsReceiver : public FPGAPortReceiver<Threat, DetectorFPGACacheddriver,
tThreatPortData>{
public:
    explicit ThreatsReceiver(DetectorFPGACacheddriver *driver) :
FPGAPortReceiver(driver) {}
protected:
    void Convert(const Threat &message, tThreatPortData &data) {
        data.setX(message.BBox.x); data.setY(message.BBox.y);
        data.setType(message.THType); data.setZ(message.BBox.z);
    }
};
...
SystemProc1::SystemProc1(DetectorFPGACacheddriver *DetectorFPGA){
    DetectorFPGA_ = DetectorFPGA;
    configReceiver_ = new ConfigReceiver(DetectorFPGA_);
    threatsReceiver_ = new ThreatsReceiver(DetectorFPGA_);
    configurator_.getReceivertype()->Connect( receiver_.getSet_type());
    configurator_.getConfigure_RF()->Connect( receiver_.getParameters());
    configurator_.getDetect_setup()->Connect(configReceiver_);
    receiver_.getThreats()->Connect(threatsReceiver_);
}
```

The stubs generated for the C++ implementation act in a similar fashion, intercepting the input and output flows in the C++ (integration) environment. The four stubs share a similar format for storing the trace data and reading from the trace files. This allows to record traces and exchange test cases (and compare outputs) between the two environments.

In Simulink, the input and output stubs have their inputs and outputs defined according to the port specifications of the module under test and will perform the save and compare of the simulation traces in the Simulink environment. These stubs will make use of the BusObject type definitions generated in the previous Matlab file and provide the internal code for creating two Simulink custom blocks, to be connected as pass-throughs at simulation-time to the Simulink subsystem under-test (as in Figure 13). These custom blocks are defined as follows.

The input block has as many inputs and outputs ports as the input ports of the SUT, with types matching the corresponding types for the input ports of the SUT. In addition, the input stub has a Boolean input port. The Boolean input configures the operation of the input stub. If the corresponding input value is true, the stub operates in record trace mode. When the simulation is run, the values arriving as input to the SUT are recorded in an XML file. The same values are passed through, unaltered, to the output ports. When the Boolean input is false, the block operates in playout mode. It ignores the values on its input ports and puts on its output ports the trace values that are read from the trace file.

The output stub has as many input and output ports as the output ports of the SUT. In addition, it has a Boolean input and a Boolean output. When the Boolean input is set to true, the output stub works in record trace mode: When the simulation is run, the values produced as output by the SUT are recorded in an XML file. The same values are passed through, untouched to the output ports for use by the other subsystem in the simulation. The additional Boolean output is ignored. When the input Boolean port is set to false, the block operates in compare mode. It compares the values produced by the SUT at simulation time with the trace data stored in a trace file. The XML files are managed using a lightweight XML parser library available as open source in C++: rapidxml (http://rapidxml.sourceforge.net/).

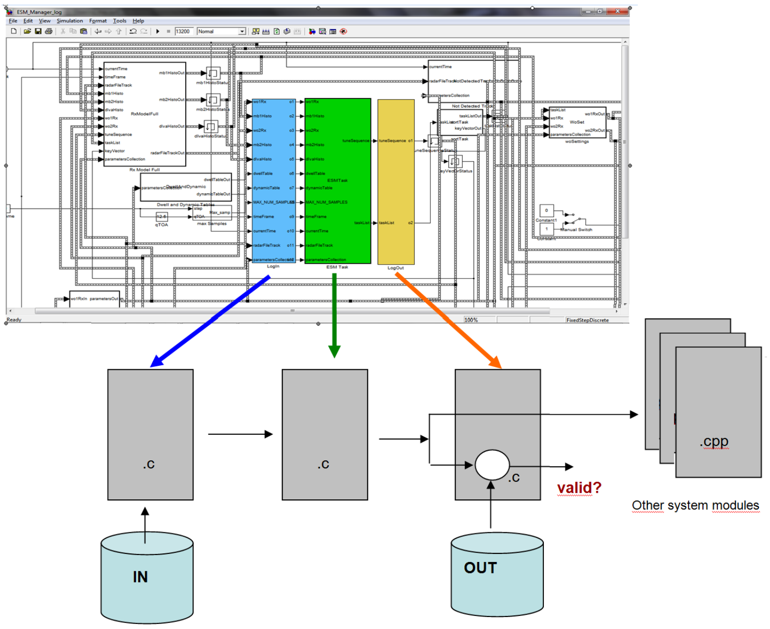

Fig. 13 Test proxys in Simulink

The stubs to be used in the code integration stage work in a similar way. They intercept the invocation of the step function that is generated to perform the reaction of the block in C++. In addition, they allow to record the inputs of a test run or playback the XML file that recorded the inputs of a run in the Simulink environment.

This allows to exchange test input vectors by simply exchanging the XML file of the recorded input traces between the two environments.

10 Impact on industrial processes and lessons learnt

The automatic generation of implementations from SysML specification is currently in use in an increasing number of projects at ELT. As of the end of 2015, 10 projects are using the automatic generation of software-firmware adapters or stubs for simulation. The ten projects share three different types of programmable hardware, for a total of 160,000 lines of code that are automatically generated for the drivers (not including the simulation code and the test stubs). 19 other projects are using similar generation facilities to automate the implementation of messaging, for an additional 70,000 lines of code.

To provide an initial assessment of the costs and benefits, at present time, the creation of SysML models for the representation at system level of all the interfaces (starting from the interface requirements specification stage and carried over to the design stages), required considerable more time in the models, as opposed to the traditional set of documents. This additional effort is estimated in 90 additional man-hours of work on average for each HW-SW interface in a project (3 vs. 1 man/months for the description of the approximately 900 locations that are used for each project), and produced savings for 75 hours in the development of the detailed design (this is because the design can be obtained by refining the SymML models instead of starting from textual requirements) and savings for additional 700 man-hours for the development and testing of interface code. Also, the development of the interface drivers benefits from an average of 2400 lines of code automatically generated for each driver, with an expected saving of 4 months of development time (from cocomo estimates) These numbers refer to the average for each interface, considered for projects that have on average 5 such interfaces (with commonalities across the projects).

In addition, the same SysML models, are used to drive other simulation and testing stages, with additional automation and savings. The use of automatic document generation tools (DocGen2) allows to tradeoff the document creation time for modeling time. Although this required an initial effort for the training of personnel in the use of the SysML Papyrus modeling tool, the time that is currently needed to create an interface specification in SysML is practically the same as the time that was needed to create the corresponding specification document (in Word), approximately 1 day for each average size interface (a protocol of 10 messages).

In addition, for the abovementioned 10 projects, models are also used for the automatic generation of configurators of Wireshark dissectors [34] that are used to test subsystems interactions over a network, and stubs for the automatic generation of messages at testing time. The expected savings from these automatic dissectors and message generators is of 1.5 man/months for the simplest dissectors (testing 10 messages). Message generators for testing more complex communications require approximately 70.000 lines of code and 5 man/months (Cocomo II estimates) that are now completely generated from specifications, with better quality and reduced maintenance and bug fixing.

Finally, automatic generation from SysML specifications guarantees that documentation, models, target code, simulation and testing tools (including dissectors) are always aligned and consistent.

Lessons learned

The experience with the modeling and generation framework presented in this paper clearly indicates that:

- open source tools for modeling and code generation have a level of maturity that is adequate for supporting industrial processes;

- automatic code generation allows to achieve significant savings in the development times provided that it operates on models that are produced as part of the standard development flow (no additional artifacts are required). In addition, the development of software components for communication, testing and simulation reduces the coding effort for development that are tedious and repetitive;
- the introduction of models and MDA tools and techniques can and should be leveraged for as many process steps as possible (documentation, simulation, coding, test) in order to reduce the impact of the nonrecurring cost of creating the SysML models.

With respect to the last point, the introduction of a model-based flow is not painless. The current savings have been achieved at the price of developments for 28 man/months (m/m). In detail, the generators described in this document took 17 m/m of development for the HW/FW interfaces (reusing 4m/m of previous generators from DSLs, 4m/m for the SysML porting, the updates that were required on the code generation templates, and the testing infrastructure), 3 m/m for the integration into eclipse (including the development of plugins, customizations, 6 m/m for the integration of the Eclipse-based framework in the process, and corrective and incremental maintenance), plus 1 m/m for the document generation templates and 10 m/m for the support to the simulation activities.

In our experience at ELT, we believed firmly in the opportunity offered by open source tools (see also [35]) and the Eclipse environment, but the final configuration of the framework required backtracking and redoing developments a few times. For example, the initial selected tool was Topcased [36] and only later we selected the Papyrus tool as the SysML modeler. Also, the initial definition of HW/SW interfaces made use of a custom metamodel and only later we decided to avoid the use of DSLs and instead providing a definition of the hardware platform components as stereotyped SysML elements. The need to involve the firmware designers in the creation of the models (and their limited expertise/knowledge of the UML language and tools) required the development of several OCL constraints and customizations, to ensure the correctness of the produced models and ease the use of the tool. Finally, the need to share models, profiles and libraries required the development of several eclipse plugins.

11 Conclusions and Future Work

We presented the flow and related tools (mostly open source, the backbone is provided by the open source Eclipse Modeling Framework (EMF) [9] and its metamodeling, model-to-model and model-to-code transformation capabilities) used for the automatic generation of communication adapters to software and firmware components that are generated from Simulink or hand-coded. The generated adapters guarantee conformance with a SysML specification and adherence to the Simulink execution semantics and conformance with a generic model of an FPGA driver interface, which alleviates the tedious programming of selecting and coding the appropriate data passing pattern. Future work includes the full extension to adapters for networked (distributed) communication on heterogeneous stacks and an extension of the simulation and test capabilities, to allow the automatic generation of boundary and robustness tests.

References

1. A. Sindico, M. Di Natale, A. Sangiovanni-Vincentelli, ”An Industrial Application of a System Engineering Process Integrating Model-Driven Architecture and Model Based Design”, ACM/IEEE 15th MODELS Conference, Innsbruck, Austria.
2. Sangiovanni-Vincentelli, A., ”Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design”, Proceedings of the IEEE, Vol. 95, No. 3, pp. 467-506, Mar. 2007.
3. The Object Management Group: http://www.omg.org
4. Mukerji,J.,Miller,J., ”Overview and Guide to OMG's Architecture”, http://www.omg.org/cgi-bin/doc?omg/03-06-01
5. M. Di Natale, F. Chirico, A. Sindico, and A. Sangiovanni-Vincentelli, ”An MDA Approach for the Generation of Communication Adapters Integrating SW and FW Components from Simulink”, Proceedings of the Models Conference 2014, Valencia, Sept. 2014
6. Paterno,F. ”Model-Based Design and Evaluation of Interactive Applications, Springer-Verlag”, London, 1999
7. The System Modeling Language: http://www.sysml.org/docs/specs/OMGSysML-v1.1-08-11-01.pdf
8. Modeling Analysis of Real Time Embedded Systems (MARTE) profile: http://www.omg.org/spec/MARTE/1.0/PDF/
9. The Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/
10. Acceleo: http://www.acceleo.org/pages/home/en
11. SIMULINK: http://www.mathworks.it/products/simulink/
12. Sindico, A., Di Natale, M., Panci, G., ”Integrating SysML With SIMULINK Using Open Source Model Transformations”, SIMULTECH 2011: 45-56
13. B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers, “A methodology to design programmable embedded systems - the y-chart approach,” in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*. London, UK, UK: Springer-Verlag, 2002, pp. 18–37.
14. Stephen J. Mellor, Scott Kendall, Axel Uhl, Dirk Weise, MDA Distilled Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004
15. Ali Koudri, Arnaud Cuccuru, Sebastien Gerard, Francois Terrier Designing Heterogeneous Component Based Systems: Evaluation of MARTE Standard and Enhancement Proposal. Proceedings of the MODELS Conference 2011, pages 243-257
16. Ernest Wozniak, Chokri Mraidha, Sebastien Gerard, Francois Terrier: A Guidance Framework for the Generation of Implementation Models in the Automotive Domain. EUROMICRO-SEAA 2011: 468-476
17. Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. Simulation, in *Transactions of the Society for Modeling and Simulation International*, 80(9):433-450, September 2004. Special Issue: Grand Challenges for Modeling and Simulation.
18. Sindico, A., Di Natale, M., Panci, G., ”Integrating SysML With SIMULINK Using Open Source Model Transformations”, SIMULTECH 2011: 45-56
19. Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang, Towards a Science of Cyber-Physical System Integration, in *Proceedings of the IEEE, Special Issue on Cyber-Physical Systems*, 100(1), 29-44, January 2012

20. Y. Vanderperren and W. Dehaene, “From uml/sysml to matlab/simulink: current state and future perspectives,” in *Proceedings of the conference on Design, automation and test in Europe*, DATE ’06 Leuven, Belgium.
21. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, no. 1, Jan. 2003.
22. G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. Science of Computer Programming, 19(2), 1992.
23. G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, “Composition and cloning in modeling and meta-modeling,” *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling*, vol. 12, pp. 263–278, 2004.
24. F. Balarin, L. Lavagno, C. Passerone, and Y. Watanabe, “Processes, interfaces and platforms. embedded software modeling in metropolis,” in *Proceedings of the Second International Conference on Embedded Software*, ser. EMSOFT ’02. London, UK: Springer-Verlag, 2002, pp. 407–416.
25. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, ”Taming Heterogeneity—the Ptolemy Approach,” Proceedings of the IEEE, v.91, No. 2, January 2003.
26. L. de Alfaro and T. Henzinger, Interface automata, Proc. of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international Symposium on Foundations of software engineering, Vienna, Austria, 2001
27. E. Lee and A. Sangiovanni-Vincentelli, ”A Unified Framework for Comparing Models of Computation”, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, No. 12, pp. 1217-1229, Dec. 1998.
28. S. Mohanty and V. K. Prasanna, A Model-Based Extensible Framework for Efficient Application Design Using FPGA ACM Transactions on Design Automation of Electronic Systems, Vol. 12, No. 2, Article 13, Publication date: April 2007.
29. M. Areno, B. Eames, A. Dasu. An automated Micro-architecture design tool for FPGAs, Proc. of the 2007 Reconfigurable Summer Systems Institute (RSSI), pp.1-10, July 2007.
30. G Hemingway, H Neema, H Nine, J Sztipanovits Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach Simulation, SAGE Journal, October 2012.
31. The EAST-ADL Association, Consortium web page (Dec 2015) “http://www.east-adl.info/.
32. The DoD Architecture Framework Version 2.02. Web page available at (Dec 2015) “http://dodcio.defense.gov/Portals/0/Documents/DODAF/DoDAF˙v2-02˙web.pdf
33. Functional Mockup Interface standard, available at (Dec 2015) “https://www.fmi-standard.org/.
34. The Wireshark project web page. Available at (Dec 2015) “https://www.wireshark.org/.
35. F. Bordeleau, Future of MBE/MDE/MDD in the Industry âĂŤ Open Source is the Only Solution!, Keynote speech at the MODELS 2014 Conference, Valencia, Sept. 2014, slides available at “http://models2014.webs.upv.es/index2˙htm˙files/Keynote2.pdf.
36. The Topcased project web page (Dec. 2015) “https://www.polarsys.org/topcased.
37. The Papyrus project web page (Dec. 2015) “https://eclipse.org/papyrus/.
38. Windriver, Simics Full System Simulator. Product web page (Dec 2015). “http://www.windriver.com/products/simics/
39. *IEEE Standard SystemC Language Reference Manual*, IEEE Computer Society, 1666-2005, 31 March, 2006.
40. F. Schirrmeister, M. Meindl and S. Krolikoski *Hardware/Software Interfaces Design for SoC*, Embedded Systems Handbook, Second Edition: Embedded Systems Design and Verification
41. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi Defining and translating a ”safe” subset of Simulink/Stateflow into Lustre. 4th ACM International Conference on Embedded Software (EMSOFTâĂŹ04), Pisa, Italy, September 2004.
42. Guoqiang Wang, Marco Di Natale, and Alberto L. Sangiovanni-Vincentelli. “Optimal synthesis of communication procedures in real-time synchronous reactive models.” in *IEEE Trans. Industrial Informatics*, 6(4): 729–743,2010.
43. M. Di Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. in *ACM Trans. Embed. Comput. Syst.*, 7(3):1–32, 2008.