An MDA Approach for the Generation of Communication Adapters Integrating SW and FW Components from Simulink

M. Di Natale[2], F. Chirico[1], A. Sindico[1], and A. Sangiovanni-Vincentelli[3]

[1] Elettronica SpA, Roma, Italy, Sindico.Andrea@elt.it
[2] TECIP Institute, Scuola Superiore S. Anna, Pisa, Italy, marco.dinatale@sssup.it
[3] EECS Dept, University of California at Berkeley, alberto@eecs.berkeley.edu

Abstract. We present the tools, metamodels and code generation techniques in use at Elettronica SpA for the development of communication adapters for software and firmware systems from heterogeneous models. 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++. In turn, Simulink models are implemented as software code or firmware on FPGA, and an automatic generation of the implementation is obtained. 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).

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

1 Introduction

In our previous work [1] we described the methodology and the process in use at Elettronica SpA for the development of complex distributed systems. The process benefits from the complementary strengths of different Model-driven approaches such as domain-specific modeling languages, Model-Driven Architecture (MDA) [4] and Model-Based Development (MBD) [5]. Starting from requirement capture, our approach follows the tenets of Platform-Based Design (PBD)[2], in which a functional model of the system is paired to a model of the execution platform. In this work, we focus on the tools and techniques used for the automatic generation of communication adapters between components generated from Simulink models and implemented in software or firmware. The 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 of emitters. PDM sequences arrive at a rate of 10^6 messages per second and to produce the results within the time constraints, the processing is partitioned in an FPGA front processor data is controlled and feeds data to a SW classifier.

The starting point is an architecture-level SysML model of the system and its component subsystems, defined according to PBD, which separates the functional model from the model of the execution platform, and the physical architecture. A third model represents the deployment of the functional subsystems onto the computation and communication infrastructure and the HW devices. Some of the functional subsystems are refined, simulated and prototyped using the Simulink environment [10]. These subsystems must adhere to the Simulink (synchronous reactive) execution semantics. In addition, domain-specific SysML [6] 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.

We define model-to-text transformations for the subsystems refined in Simulink to generate an interface specification (ports and port types) consistent with the SysML subsystem definition. The subsystem is then refined as a Simulink model and 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 [10] standards). An FPGA implementation is automatically generated for components mapped onto programmable HW.

Our framework provides the generation of 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.

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

- The definition of a SysML profiles that extends MARTE to express the realization of embedded functionality as software or firmware components.
- 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 also a driver layer is 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.

– 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 organization of the paper is the following. Section 2 provides an outline to our methods, tools and models for the generation of the adapters , including an outline of the structure of the generated code. 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 flow to represent the design of the system components and the generation of the driver code towards the programmable HW. Section 6 provides the details of methods and tools for the integration of Simulink components and Section 7 discusses the generation of the communication code. Finally, Section 8 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 1. 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 the FPGA implementations (access to the HW platform, as described in Section 5) 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 blue, in Figure 1). 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.

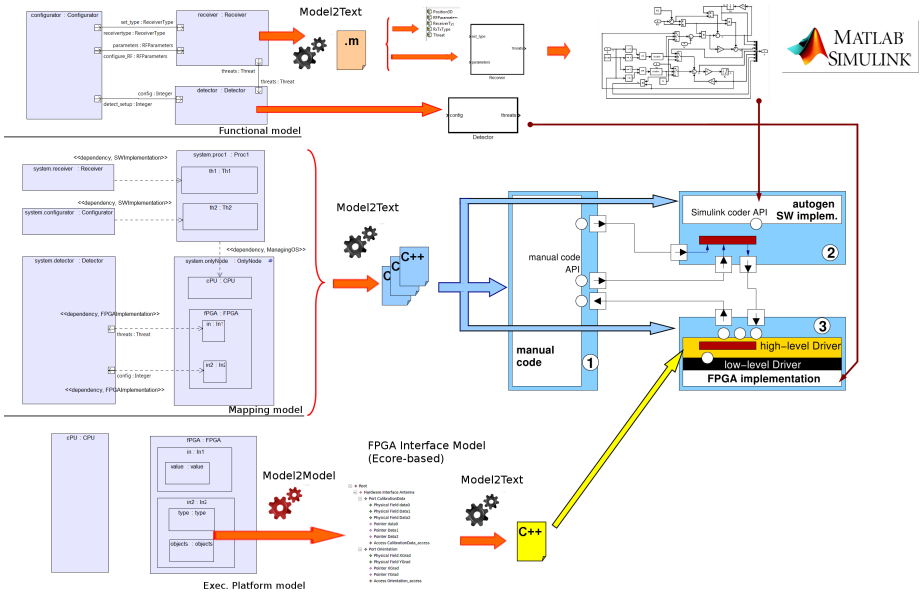

Fig. 1. The generated wrappers provide for the communication among subsystems

3 Related work

The match of a functional and execution architecture is advocated by many in the academic community (examples are the Y-cycle [12] 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) [13].

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 [7] 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 [15]. However, because of the complexity and the variety of modeling concepts it has to support, MARTE can still be considered as work in progress, being constantly evaluated [14] and subject to future extensions. Several other domain-specific

languages and architecture description languages of course exist, such as, for example EAST-AADL and the DoD Architectural Framework.

Several other authors [18] 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 [18]. Work on the integration of UML and synchronous reactive languages [19] 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 [20]. 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 [21] and Metropolis [22]) consist of the use of a general meta-model as an intermediate target for the model integration.

A large number of works deal with the general subject of integration of heterogeneous models. Examples are the CyPhy/META Toolchain at Vanderbilt [17] and the work on multiparadigm modeling (a general discussion in [16]). In both cases, emphasis is placed on the role of domain-specific languages and model trasformations in the general context of large and distributed Cyber-Physical systems. Other groups and projects [23] 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 [24] and the Tagged Signal Language [25].

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.

`<<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.

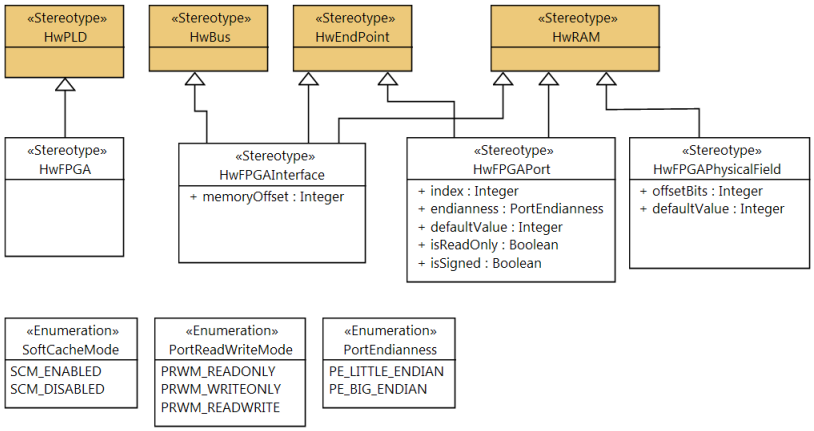

Fig. 2. A SysML profile for the description of Interfaces to programmable HW

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 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 2). 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 2) are:
`<<HwFPGA>>` refines «HwFPGA» 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 (the description of its properties is omitted for space reasons).
`<<HwFPGAPhysicalField>>` refines «HwRAM». It defines a hardware register representing a field of information in a Port.

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 (not representative of the real industrial application)

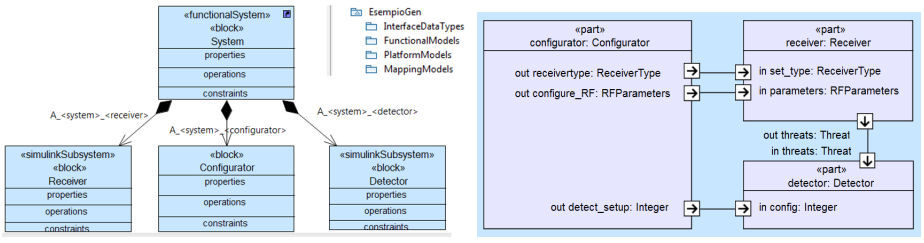

Fig. 3. The ibd showing the port connections for the a sample model

The corresponding platform model is shown in Figure 4, 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.

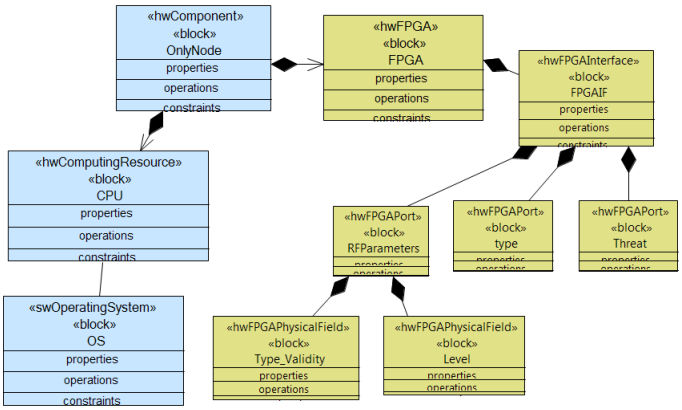

Fig. 4. The platform model for the example

The mapping model information for our example is represented in an ibd diagram as in Figure 5. The Detector and Configurator subsystem instances in the functional system model are deployed as software implementations onto 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 allow to keep the functional and platform models completely independent, while at the same time, providing the necessary information for the code generation stage.

5 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

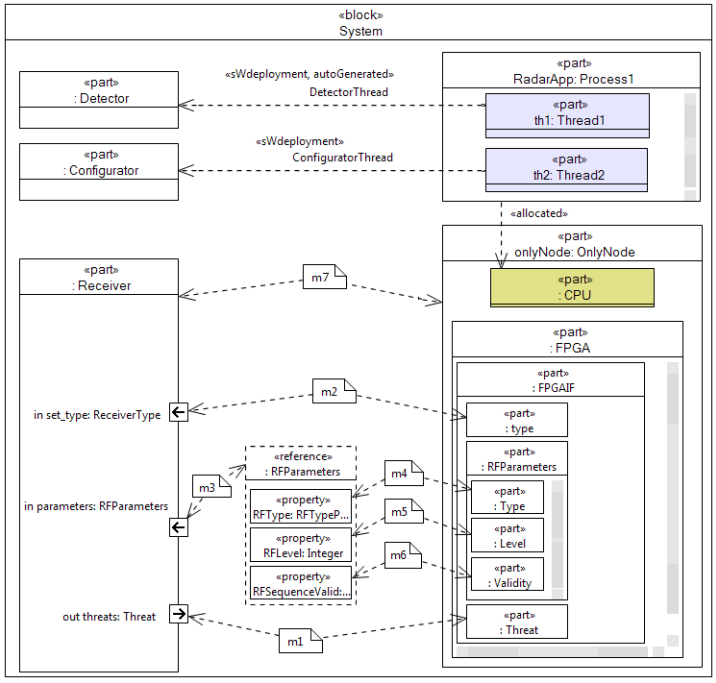

Fig. 5. The mapping model for the example

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.

6 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 specification as *Bus Objects* (the tool-specific type/class declarations). An Acceleo [9] module transforms the SysML block and generates a Matlab script that creates in the Matlab environment a set of Bus Object specifications mirroring the definitions of the data types in the SysML model; one file for each enumerated type in the SysML type specifications that apply to the subsystem ports; and a script that generates the boundary of the subsystem with its ports (as described in [11]). The subsystem is then defined internally and simulated, until its behavior is defined in a satisfactory way. When the Simulink model is completed, the automatic generation of its FPGA (if firmware) or C++ code implementation (if software) implementation is performed using Simulink Coder.

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 defines how the interface ports translate into arguments of the `step` and allows to define the data types in an external (user provided) file. Listing 1.1 shows the code generated for the Receiver subsystem in our example.

Listing 1.1. 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 */
}
```

7 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.

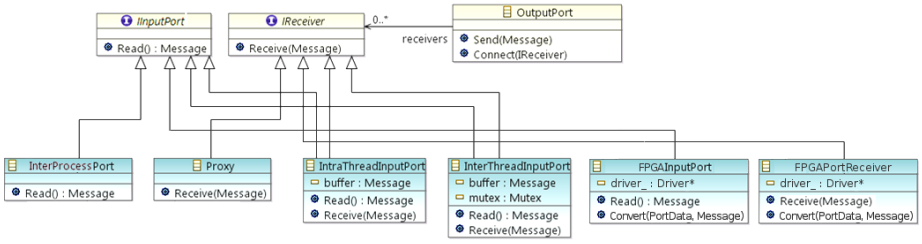

Fig. 6. Hierarchy of classes for subsystems ports.

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 6. 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 1.2 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 1.2. 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 6, 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.

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).

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.

Listing 1.3. 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 SubsystemReceiver class generates for our example (shown in listing 1.3) 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 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.

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 1.4). 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

Listing 1.4. 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);
  }
... };
```

data port type into the PhysicalFields of the FPGA physical port, according to the mapping specified in the SysML model.

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.

8 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) [8] and its metamodeling, model-to-model and model-to-code transformation capabilities) used for the automatic generation of communication adapters to automatically generated software and firmware components (from Simulink) and hand-coded classes. The generates 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.

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. Paterno,F. "Model-Based Design and Evaluation of Interactive Applications, Springer-Verlag", London, 1999
6. The System Modeling Language: http://www.sysml.org/docs/specs/OMGSysML-v1.1-08-11-01.pdf
7. Modeling Analysis of Real Time Embedded Systems (MARTE) profile: http://www.omg.org/spec/MARTE/1.0/PDF/
8. The Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/
9. Acceleo: http://www.acceleo.org/pages/home/en
10. SIMULINK: http://www.mathworks.it/products/simulink/
11. Sindico, A., Di Natale, M., Panci, G., "Integrating SysML With SIMULINK Using Open Source Model Transformations", SIMULTECH 2011: 45-56
12. 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.
13. Stephen J. Mellor, Scott Kendall, Axel Uhl, Dirk Weise, MDA Distilled Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA Â©2004
14. Ali Koudri, Arnaud Cuccuru, Sebastien Gerard, FranÃğois Terrier Designing Heterogeneous Component Based Systems: Evaluation of MARTE Standard and Enhancement Proposal. Proceedings of the MODELS Conference 2011, pages 243-257
15. Ernest Wozniak, Chokri Mraidha, Sebastien Gerard, FranÃğois Terrier: A Guidance Framework for the Generation of Implementation Models in the Automotive Domain. EUROMICRO-SEAA 2011: 468-476
16. 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.
17. 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
18. 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.
19. 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.
20. . G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. Science of Computer Programming, 19(2), 1992.

21. 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.
22. 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.
23. 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.
24. 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
25. 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.