Innovations Syst Softw Eng (2009) 5:35–47
DOI 10.1007/s11334-009-0080-9

ORIGINAL PAPER

Model transformations in the UPES/UPSoC development process for embedded systems

Elvinia Riccobene · Patrizia Scandurra

Received: 1 September 2008 / Accepted: 12 January 2009 / Published online: 10 February 2009

Abstract Model-based development (MBD) aims at combining modeling languages with model transformers and code generators. Modeling languages, like profiles of the Unified Modeling Language (UML), are increasingly being adopted for specific domains of interest to alleviate the complexity of platforms and express domain concepts effectively. Moreover, system development processes based on automatic model transformations are widely required to improve the productivity and quality of the developed systems. In this paper, we show how MBD principles and automatic model transformations provide the basis for the unified process for embedded systems (UPES) development process and its unified process for system-on-chip (SoC) (UPSoC) subprocess. They have been defined to foster in a systematic and seamless manner a model-based design methodology based on the UML2 and UML profiles for the C/SystemC programming languages, which we developed to improve the current industrial system design flow in the embedded systems and system-on-chip area.

Keywords Model-based development (MBD) · Model transformations · Unified modeling language (UML) · SystemC · Embedded system design

This work is supported in part by the project *Model-driven methodologies and techniques for embedded systems design and analysis based on UML, Abstract State Machines, and SystemC* at STMicroelectronics, AST R&I of Agrate Brianza (MI), Italy.

E. Riccobene (✉)
Dip. di Tecnologie dell'Informazione,
Università di Milano, via Bramante 65, 26013 Crema (CR), Italy
e-mail: riccobene@dti.unimi.it

P. Scandurra
Dip. di Ingegneria dell'Informazione e Metodi Matematici,
Università di Bergamo, Viale Marconi 5, 24044 Dalmine (BG), Italy
e-mail: patrizia.scandurra@unibg.it

1 Introduction

Model-based development (MBD) is an emerging paradigm in software production that combines domain-specific modeling languages (DSMLs) with model transformers, analyzers, and generators. Metamodel-based modeling languages, such as profiles or extensions of the Object Management Group (OMG) standard Unified Modeling Language (UML), are increasingly being defined and adopted for specific domains of interest (telecommunications, embedded systems, system-on-chip, real-time computing, aerospace, automotive, etc.), addressing the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [35].

MBD emphasizes that software systems should be designed at a high abstraction level and then incrementally refined to contain more specific and detailed information, ending with the implementation of the system on a given platform. MBD principles propose the automation of parts of these refinements through automatic *model transformations* in order to increase both the productivity and the quality of the developed systems. Model transformations are currently a key research topic in the MBD context. The main research interest lies in *vertical transformations*, i.e., transformations between models of different levels of abstraction such as the platform-independent model (PIM) level, platform-specific model (PSM) level, and code level of the OMG model-driven architecture (MDA) [17]. Another important aspect is *horizontal transformations*, where models are translated into other models at the same level of abstraction.

In [31], we presented a MBD-based methodology for embedded systems that we defined according to the *platform-based design* principles [15,45] and by exploiting the OMG MDA as a framework for MBD. The design methodology is based on UML2, on a SystemC UML profile for the hardware parts, and on a multithread C UML profile for the software parts. Both of the UML profiles are consistent sets of modeling constructs designed to lift constructs (both structural and behavioral) of the SystemC and C coding languages, respectively, to the UML modeling level. The design methodology, supported by a modeling environment, allows modeling of the system at higher levels of abstraction from a *functional level* to a register transfer level (RTL), handling both the hardware architecture with the hardware-dependent software (HdS)—i.e. all software that is directly dependent on the underlying hardware—and the (hardware-independent) application software. In [33], we presented the development process unified process for embedded systems (UPES), as an extension of the well-known software engineering unified process (UP) [2] from the authors of the UML, to foster our design methodology. UPES includes a subprocess, called the unified process for SoC (UPSoC), which combines all involved notations together in a systematic and seamless manner for the HdS refinement flow.

In this paper, we show how the UPES/UPSoC processes are based on vertical and horizontal model transformations from abstract models toward refined (but still abstract) models and/or software code models. For the UPSoC subprocess, we show how well-established abstraction/refinement patterns from hardware or system engineering and until now only used at code level, can be managed as *model-to-model* transformation steps and therefore applied at the UML level, by the use of the SystemC UML profile, along the modeling process from a high-functional-level model down to a RTL model. We also show how MBD principles are applied to provide the automation of *model-to-code* transformations from abstract UML models toward detailed SystemC code.

The remainder of this paper is organized as follows. Sections 2 and 3 describe the design process UPES and its subprocess UPSoC, respectively. Section 4 gives an overview of the notations adopted in the UPSoC process, while Sect. 5 provides a brief description of the design environment. Section 6 presents automatic model transformations used in the UPSoC process. Section 7 provides some related work. Section 8 concludes the paper and sketches some future directions of our contribution.

2 The unified process for ES (UPES)

The UPES process drives designers during the UML modeling activity from the analysis of the informal requirements to a high-level functional model of the system, down to a RTL model, by supporting current industry best practice and the *platform-based design* principles [45, 15]. The UPES process is an MDA-style Y-development cycle (Fig. 1). It consists on one side of the conventional UP process for software development to define a first executable model of the system (the

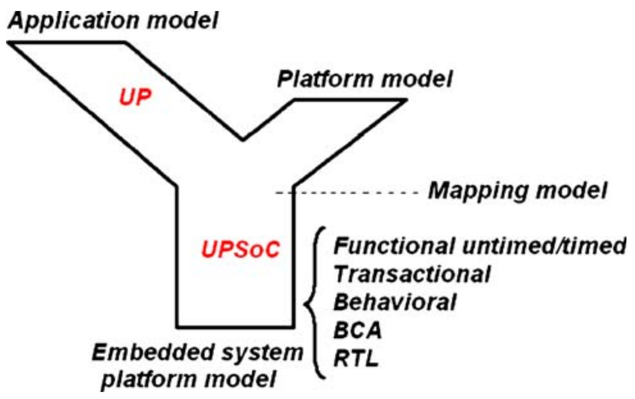


Fig. 1 The UPES Y-chart

application model or *system model*), and on the other side of the selection process, an available hardware platform (the *platform model*) modeled with an appropriate language (e.g., a particular UML extension for SoC design).

The intersection, which is intended as a *model weaving* operation in the model-based context, is the mapping of the application model on the given platform model to establish semantic links between models at specific *join points*. As input, this task requires also a reference model of the mapping (the *mapping model* or *weaving model*) to try, which is specified in terms of UML component and deployment diagrams to denote and annotate the partitioning of the original system in hardware (HW) and software (SW) components. This mapping model establishes the relationships (join points) of the platform resources and services with the application-level functional components. The UPSoC subprocess, an UP extension for SoC design (see the next section), is then followed to accurately refine the embedded system platform from a functional level to the RTL level.

3 The unified process for SoC (UPSoC)

The UPSoC drives system designers during the refinement of the embedded software, after the mapping phase, and therefore after the system components assigned to the HW partition have been mapped directly onto the HW resources of the selected platform. The UPSoC can also be adopted in a stand-alone way also by platform providers to deliver offline models of platforms[1] by designing from scratch an abstract hardware platform—from the analysis of informal requirements about the architecture elements (HW resources and processing elements)—or to directly design it at a desired

[1] Detailed platform models comprise a structural view (provided, e.g., by UML composite structure diagrams) and a behavioral one. Component and deployment diagrams can be used then to provide a black-box view of the application programming interface (API) layer and the micro-architecture layer, respectively.

level of abstraction. The model of this generic platform could then be reused and targeted accordingly to implement a specific hardware platform on which to embed a given system. In both cases, by using, e.g., the UML profile for SystemC [30,34] that provides in a graphical way all modeling elements for the design of the hardware components, the hardware platform can be modeled at different abstraction levels: functional untimed/timed, transactional (TLM), behavioral, bus cycle accurate, and RTL (Fig. 1).

The UPSoC adapts the UP to the SoC domain by extending the *design* and *implementation* workflows, while keeping the *use case* and *analysis* workflows.[2] Useful considerations about use-case analysis in the context of systems engineering and SoC design can be found in [43]. The *design* and *implementation* models in the UPSoC can be defined at different abstraction levels and with different modeling guidelines. Figure 2 shows the workflow schema for these models using the SPEM [38] notation. The letter L in the figure denotes a specific level of abstraction from the set {functional, TLM, behavioral, BCA, RTL}; therefore, there are five design/implementation workflows. As in the UP, also a *testing model* (not shown in the figure) can be provided by creating test cases and test procedures housed outside of the modeling tool, but traced back to the models.

From the TLM level to the RTL, previous refinement guidelines [6,40] can be followed to bridge refined elements with more abstract elements such as those regarding the introduction of *wrappers* and *adaptors*, and to restrict the design for the hardware modeling. A concise description of each of the five workflows follows.

Functional design/implementation After analysis, the *design phase* starts by defining the components of the platform as a set of UML subsystems and interfaces. A subsystem is a type of UML component whose functionality is exposed to the outside environment in the form of one or more interfaces representing *operational contracts* (a message-based interface communication concept). At the end of the design phase, a draft architecture and its functionality is available.

The *architectural design* is performed in an incremental way, by creating a subsystem that corresponds to each analysis package (output of the analysis workflow) and then refining those subsystems. UML package diagrams can be used to show the subsystems. Then details are added in each subsystem, dividing each subsystem into a *specification* and a *realization* part. The former contains design use cases obtained by refining the use cases of the corresponding analysis package.

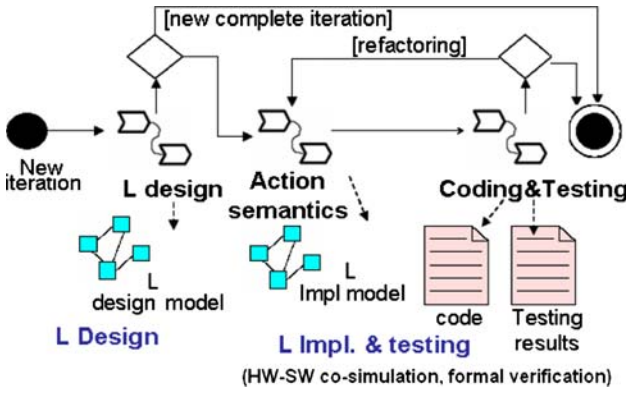


Fig. 2 The UPSoC: L- design, implementation, and testing workflow

The latter contains design classes and interfaces coming from the analysis classes and from an analysis of the design use cases. UML class diagrams are built to show interfaces, classes, and relationships, and UML composite structure diagrams to represent structural hierarchies using parts with ports as basic structural elements, and the notation of provided/required interfaces to denote interface usage/realization dependencies.

The *functional design* is then performed by identifying operations and dynamic behavior of each class. For this purpose, UML state machine diagrams and UML activity diagrams are used, while UML interaction diagrams are useful to get a better understanding of objects' interactions. Finally, subsystems are refined to ensure they are as cohesive and loosely coupled as possible, and that they provide the functionality denoted by their specification part.

The UML model developed in the design phase corresponds to the initial *functional untimed model* of the embedded system. It describes the pure system functionality, and the structural elements do not correspond to blocks in the physical implementation, no time information is provided, and communication is modeled point to point. In order to make it *executable* and *timed*—i.e., a *functional timed model*—the *implementation phase* has to be carried out, where a precise action semantics and time model have to be adopted. An action (or surface) language can be used, supported by a proper execution engine, and the behavioral diagrams can be enriched with actions expressed according to the adopted action and time semantics. Timing diagrams, and annotate interaction diagrams with time-related constraints[3] may also be included. Typical execution engines for the SoC design are those used for programming languages such as C/C++, SystemC, etc. UML profiles targeted to these execution engines and model-to-code transformations provide painless and automatic paths toward these environments.

[2] Each UP phase (*inception*, *elaboration*, *construction*, and *transition*) usually consists of a number of *iterations*. During each iteration, several activities, called *workflows*, are performed in parallel. The *core* workflows of UP are *requirements* (or *use cases*), *analysis*, *design*, *implementation*, and *testing*.

[3] The SysML modeling language [39] and the MARTE UML profile [37], e.g., provide a set of constructs for expressing time-related aspects.

Transactional design/implementation At the TLM level, the architecture is designed and validated in terms of functionality as characterized by high-level block input and output events, and interblock data transfers. Communication details among processing elements are separated from the details of computation. Communication is modeled by channels, while transaction requests take place by calling interface functions of these channel elements. Transactions are protocols exchanging abstract data between blocks, with associated timing information. Elements of a TLM model can correspond or not to blocks in the physical implementation. Channels' interfaces encapsulate low-level details to enable higher simulation speed than pin-based interfaces and also to speed up the modeling task.

The SoC architecture is designed with Intellectual Property (IP) blocks connected via application program interfaces (APIs) to implementation-independent high-level bus models. System IP components and buses may be modified or replaced with greater ease than at RTL, and simulated more than 1,000 times faster. Thus designers can quickly optimize the design to achieve what works best.

To make a transactional model executable (implementation model), UML profiles for languages such as SpecC and SystemC supporting the transaction-level model can be considered. Similar considerations apply to the next levels.

Behavioral design/implementation A *behavioral model* is pin-accurate and functionally accurate at its boundaries. It is not cycle accurate, i.e., internal and input/output (I/O) events are not scheduled in a cycle-accurate manner. There is a certain freedom left to implement the model in a certain number of clock cycles.

BCA design/implementation A *bus-cycle-accurate model* is pin-accurate and functionally accurate. It is also cycle-accurate at its boundaries, i.e., in the number of clock cycles it takes to perform its functionality (I/O interactions are scheduled at the proper clock cycle; internal events are not scheduled).

RTL design/implementation At RTL level the hardware is described in terms of transfers among registers though functional units like adders, multipliers, arithmetic logic units (ALUs), etc. The *RTL model* is accurate at the clock cycle level, both at the boundaries and internally (all actions are scheduled at the appropriate clock cycle), and also pin-accurate.

The RTL platform model serves as input to the lower-level very-large-scale integration (VLSI) design flow, where logic simulation and synthesis are performed, leading to the final product.

4 UPSoC notations

A brief overview is provided here of the two main notations adopted in the UpSoC subprocess: the SystemC coding language as the target language of the model-to-code transformations, and the SystemC UML profile as the main modeling notation used at different levels of abstractions and, therefore, source/target language of the model-to-model transformations.

4.1 SystemC overview

SystemC [1,21] is an IEEE standard controlled by the major companies in the electronic design automation (EDA) industry. It is a system-level design language intended to support the description and validation of complex systems in an environment completely based on the C++ programming language.

SystemC is defined in terms of a C++ class library, organized according to a layered architecture, shown in Fig. 3 (taken from [40]). The *core language* and *data types* are the so-called *core layer* (or layer 0) of the standard SystemC, and consist of the event-based and discrete-timed SystemC simulation kernel, the core design primitives and data types. The *primitive channels* represents, instead, the layer 1 of SystemC; it comes with a predefined set of interfaces, ports and channels for commonly used communication mechanisms such as signals and FIFOs. Finally, the external libraries layer on top of the layer 1 are not considered as part of the standard SystemC language.

Figure 4 depicts a simplified metamodel (an abstract syntax provided in terms of a class diagram) of the main SystemC terms and concepts. The design of a system in SystemC is essentially given by a containment hierarchy of *modules*. A *module* is a container class able to encapsulate *structure* and *functionality* of hardware/software blocks. Each module may

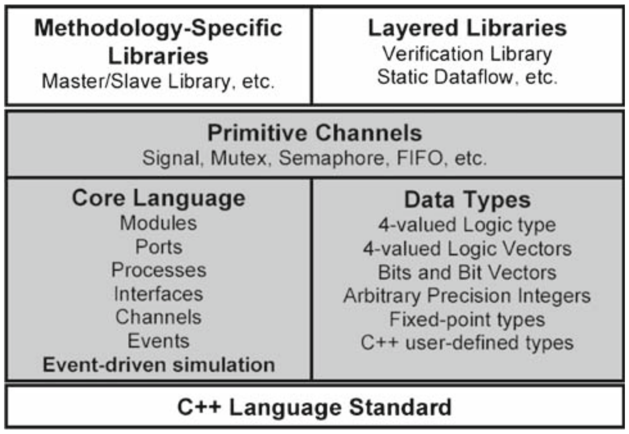


Fig. 3 SystemC language architecture

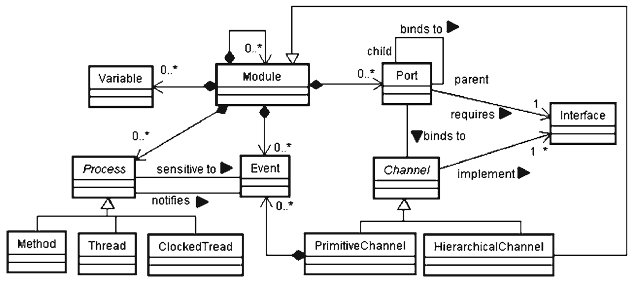


Fig. 4 A simplified SystemC metamodel

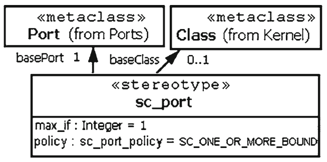
contain *variables* as simple data members, *ports* for communication with the surrounding environment and *processes* for performing module's functionality and expressing concurrency in the system. Two kinds of processes are available: *method* processes and *thread* processes. They run concurrently in the design and may be sensitive to *events* which are notified by other processes. A port of a module is a proxy object through which the process accesses a channel interface. The *interface* defines the set of access functions for a channel, while the channel provides the implementation of these functions to serve as a container to encapsulate the *communication* of blocks. There are two kinds of channels: *primitive* channels and *hierarchical* channels. Primitive channels do not exhibit any visible structure, do not contain processes, and cannot (directly) access other primitive channels. A hierarchical channel is a module, i.e., it can have structure, it can contain processes, and it can directly access other channels.

In 2006, SystemC received a major revision (2.2) and became an IEEE standard [1]. This last revision includes new structural and behavioral features for transaction-level modeling (TLM) according to the Open SystemC Initiative (OSCI) TLM standard.

4.2 The SystemC UML profile

The UML2 profile for SystemC [34] is a consistent set of modeling constructs designed to lift both structural and behavioral constructs of the SystemC coding language (including events and time features) to the UML modeling level.

A UML2 profile is a set of *stereotypes*. Each stereotype defines how the syntax and the semantics of an existing UML2 construct (a class of the UML2 metamodel) is extended for a specific domain terminology or purpose. A stereotype can define additional semantic *constraints*—the *well-formedness rules* expressed in the object constraint language (OCL) [20] over the base metaclass—to enforce a semantic restriction of the extended modeling element, as well as *tags* to state additional properties. Figure 5 shows an example of stereotype definition for the SystemC `sc_port` together with an example of OCL constraint. At model level,



English: A port can have exactly one required interface and no provided interfaces.
OCL: basePort.required->size()=1 and basePort.provided->size()=0

Fig. 5 `sc_port` stereotype

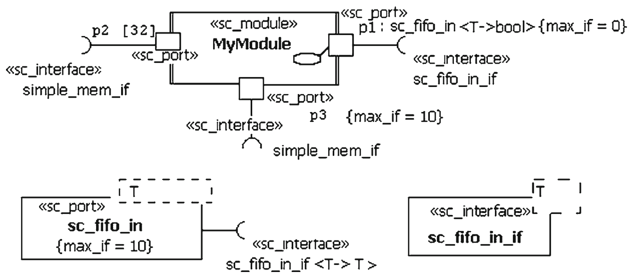


Fig. 6 Modules, ports, and interfaces

when a stereotype is applied to a model element (an instance of a UML metaclass), an instance of a stereotype is linked to the model element. From a notational point of view, the name of the stereotype is shown within a pair of guillemets above or before the name of the model element and the eventual tagged values displayed inside or close as name–value pairs. Examples of stereotypes application are provided here (see Fig. 6, e.g., for the `sc_port` stereotype) and in Sect. 6.1.

The SystemC UML profile is defined at two distinct levels—the SystemC core layer (or layer 0) and the SystemC layer of predefined channels, ports and interfaces (or layer 1)—reflecting the layered architecture of SystemC.

The core layer—*the basic SystemC profile*—is the foundation upon which specific libraries of model elements or also other modeling constructs can be defined. It is logically structured as follows.

A *structure and communication* part defines stereotypes for the SystemC building constructs (modules, interfaces, ports, and channels) for use in UML structural diagrams such as UML class diagrams and composite structure diagrams. UML class diagrams are used to define modules, interfaces, and channels. Figure 6 shows an example of a SystemC module exposing a multi-port, an array port, and a simple port, together with the port type and the interface definitions of the simple port.

The internal structure of composite modules, especially of the topmost-level module (representing the structure of the overall system), is captured by UML composite structure diagrams. From these diagrams several UML object diagrams can be created to describe different configuration scenarios. This separation allows the (also partial) specification of different HW platforms as instances of the same

UML Notation	SystemC Notation

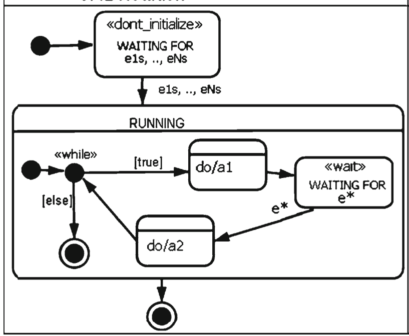
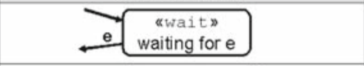
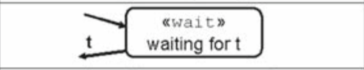
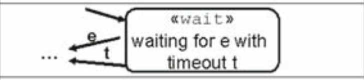
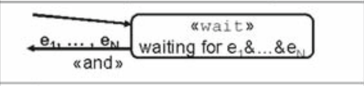
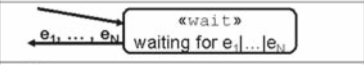
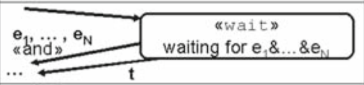
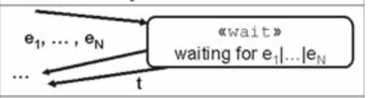
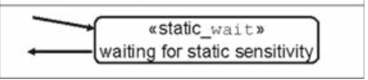
	`...`
	`SC_CTOR(my_module)`
	`{`
	`SC_THREAD(my_thread);`
	`sensitive << e1s << .. << eNs;`
	`dont_initialize();`
	`}`
	`...`
	`void my_module:: my_thread()`
	`{`
	`  while(true) {`
	`    a1;`
	`    wait(e*)`
	`    a2;`
	`  }`
	`}`

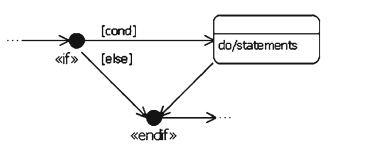
Fig. 7 A thread process pattern

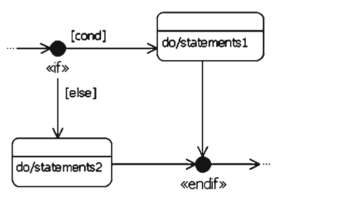
SystemC	UML Notation
`wait(e)` wait for an event e	
`wait(t)` wait for t time units	
`wait(e,t)` wait for an event with timeout	
`wait(e1&..&eN)` wait for an AND-list of events	
`wait(e1|..|eN)` wait for an OR-list of events	
`wait(t,e1&..&eN)` wait for an AND-list of events with timeout	
`wait(t,e1|..|eN)` wait for an OR-list of events with timeout	
`wait()` wait for static sensitivity	

«wait» waiting for e
«wait» waiting for t
«wait» waiting for e with timeout t
«wait» waiting for e_1&...&e_N
«and»
«wait» waiting for e_1|...|e_N
«static_wait» waiting for static sensitivity

Fig. 8 Dynamic sensitivity of a thread

UML Notation	C++ Notation

	`if (cond){`
	`  statements`
	`}`

	`if (cond){`
	`  statements1`
	`}`
	`else {`
	`  statements2`
	`}`

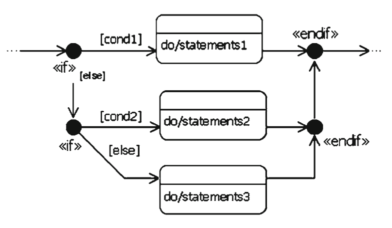
	`if (cond1){`
	`  statements1`
	`}`
	`else if(cond2){`
	`  statements2`
	`}`
	`else {`
	`  statements3`
	`}`

Fig. 9 Conditional controls

parametric model (i.e., the composite structure diagram). Concrete examples of composite modules are provided in Sect. 6.1.

A *behavior and synchronization* part defines special state and action stereotypes which lead to an extension of the UML state machines, the *SystemC process state machines* [29]. This formalism has been appositely included in the profile definition to model the control flow and the reactive behavior of SystemC processes (methods and threads) within modules. A finite number of *abstract behavior patterns* of state machines have been identified. Figure 7 depicts one of these behavior patterns together with the corresponding SystemC pseudocode for a thread that: (i) is not initialized, (ii) has both a static (the event list $e_{1s}, \ldots, e_{Ns}$) and a dynamic sensitivity (the `wait` state), and (iii) runs continuously (by the infinite `while` loop). Note that the notation used for the `wait` state in the pattern in Fig. 7 stands for a shortcut to represent a generic `wait(e*)` call where the event `e*` matches one of the cases reported in Fig. 8. Moreover, activities `a1` and `a2` stand for blocks of sequential (or not) code without `wait` statements.

It should be noted that the state machine pattern depicted in Fig. 7 can be more complex if one or more `wait` statements are enclosed in the scope of one or more nested control structures. In this case, as part of the UML profile for SystemC, the control structures `while`, `if`, etc. need to be explicitly represented in terms of special stereotyped junction or choice pseudostates[4] combined together in order to stand out the state-like representation of the `wait` calls. The conditional control and loop controls, e.g., can be modeled as shown in Figs. 9 and 10, respectively. These stereotypes allow us to generate code effectively from state diagrams in a style that reflects the nature of constructs of the target implementation language, despite well-know techniques (such as state pattern, stable table pattern, etc.) currently used to achieve this goal.

A *data types* part defines a UML class library to represent the set of SystemC data types.

In addition, predefined channels, ports, and interfaces of layer 1 of SystemC are provided either as a UML class library, modeled with the basic stereotypes of the SystemC core layer, or as a group of stand-alone stereotypes—*the extended SystemC profile*—specializing the basic profile.

An extended version of the SystemC UML profile encapsulating some new structural and behavioural features

[4] Choice pseudostates must be used in place of junction pseudostates whenever the head condition of the `while` loop is a function of the results of prior actions performed in the same run-to-completion step.

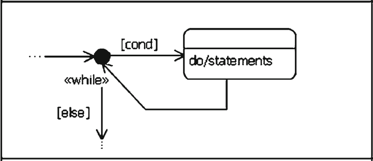
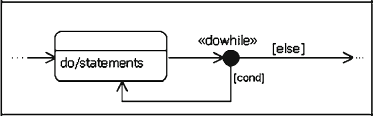
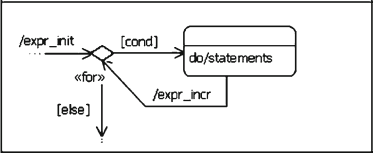
UML Notation	C++ Notation
	while (cond){ statements }
	do{ statements }while (cond);
	for(expr_init; cond; expr_incr){ statements }

Fig. 10 Loop controls

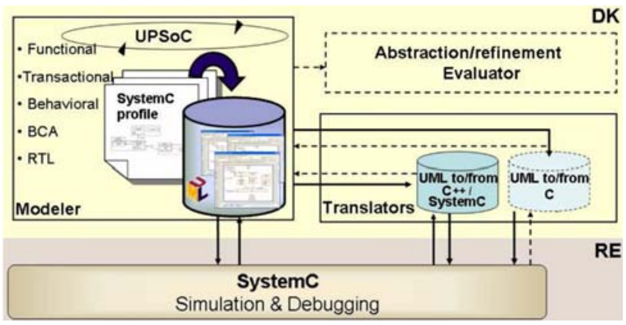


Fig. 11 Tool architecture

(i.e., *dynamic processes* and *fork/join synchronization* mech-a-nisms) necessary to model a TLM library conforming to the OSCI SystemC TLM library was presented in [3].

5 UPSoC design tool

For industrial application, the availability of appropriate tool support is crucial. In order to cover the full spectrum of UPSoC by using C/C++ and SystemC as action languages, we developed a design tool working as front-end for consolidated lower-level co-design tools.

Figure 11 shows the tool architecture. Components visualized inside dashed lines are still under development. The tool consists of two major parts: a development kit (DK) with design and development components, and a runtime environment (RE) represented by the SystemC execution engine.

The DK consists of a UML2 *modeler* supporting the SystemC UML profile, *translators* for the forward/reverse engineering to/from C/C++/SystemC, and an *abstraction/ refinement evaluator* to guarantee traceability and correctness along the refinement process from the high-level abstract description to the final implementation. A more detailed description of the tool is provided in [32].

Initially, the UML2 modeler was based on the commercial Enterprise Architect (EA) tool [10] by SparxSystems. Figure 12 shows a screenshot of EA. The SystemC data types and predefined channels, interfaces, and ports are modeled with the core stereotypes, and are available in the `project view` with the name `SystemC_Layer1`.

Recently, we have implemented the UML2 modeler also in the Papyrus framework [23], an open-source Eclipse-based UML modeler which supports the UML2 standard as defined by the OMG and allows the integration of model transformation services. The latter are handled by transformation engines, such as the ATL engine [13] (the one we adopted), developed within the Eclipse Modeling Project [12] as implementation of the OMG Queries–Views–Transformations (QVT) [27] standard.

6 UPSoC model transformations

The MDA-style separation of models of UPES/UPSoC demands automatic model transformations as support to *evolution activities* [18] in general, such as refinement/ abstraction, model refactoring, model inconsistency management, etc.

According to the classification given in [8], for refinement/abstraction purposes we identify two main kinds of model transformations to be adopted specifically in the UPSoC process: *model-to-model* transformations and *model-to-code* transformations. They are better explained in the following sections by providing concrete examples.

6.1 Model-to-model transformations

Model-to-model transformations of UPSoC allow model refinement along the (not necessarily successive) abstraction levels: functional, transactional, behavioral, BCA, and RTL. These transformations must be intended as *vertical* transformations as they imply a change of the level of abstraction. Each model is here intended as an instance of the SystemC UML profile metamodel, and therefore is an *implementation model* with a fixed action semantics.

In the model-to-model category, we adopt a hybrid approach based on both direct manipulation and structure-driven approach [8]. The idea is to collect and reuse precise abstraction/refinement transformation patterns, coming from industry best practices. The transformation patterns are once and for all proved correct and complete, and can be used to guide the refinement process or to point out missing elements in a refinement.

We have implemented some of these patterns in the EA-based environment by using the EA MDA transformers, and in the Papyrus-based environment using the ATL engine. By applying the model transformations defined for a

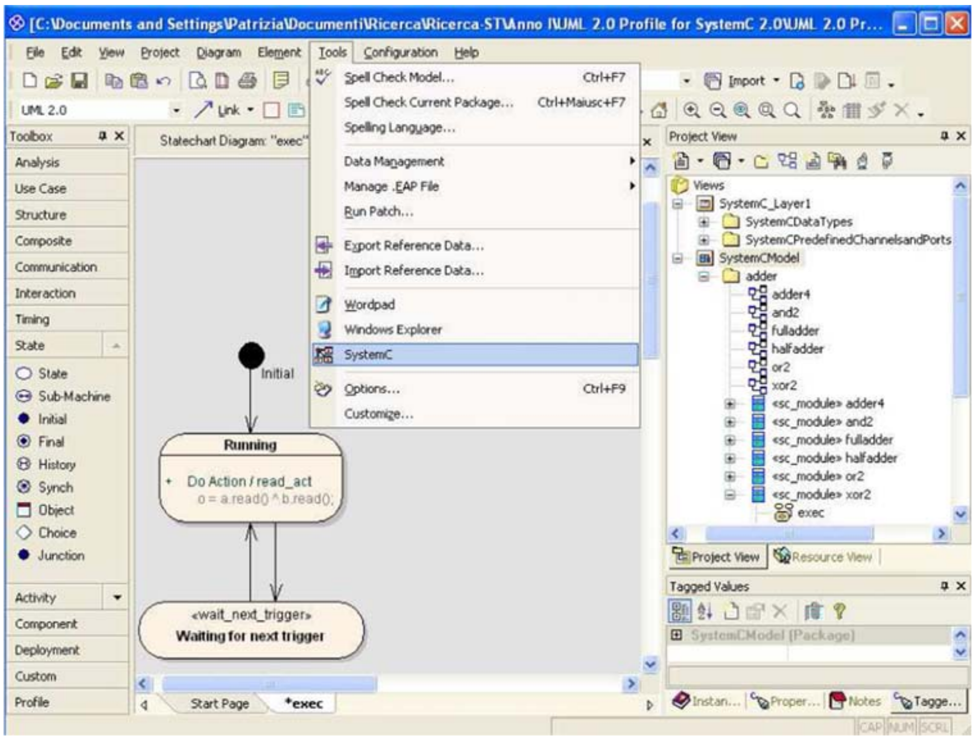


Fig. 12 Generate SystemC code from EA

given pattern, a UML system design may automatically and correctly evolve to an abstract/refined model reflecting the abstraction/refinement rules defined for the applied pattern. Below, we provide an example of communication refinement from functional level to RTL level based on model transformations.

6.1.1 Communication refinement

In general, in the refinement process we do not only refine the model's internal structure, its timing, or the data types being used; we also need to think about how components communicate with their environment. *Communication refinement* refers to mapping an abstract communication protocol into an actual implementation related to a given target architecture. To give an idea on how to perform refinement at UML level by model-to-model transformations, we show here how to apply communication refinement patterns to a functional timed model leading to a RTL model.

We clarify the process of communication refinement in a general way. To this purpose, we assume to have two high-level modules M1 and M2 communicating over a channel C via some abstract protocol. As suggested in [40], one possible approach to refining this basic communication scenario toward an implementation consists of the following steps:

1. Select an appropriate communication scheme to implement
2. Replace the abstract communication channel C with a refined one $\mathsf{C}_{\text{Refined}}$ realizing the selected communication protocol
3. Enable the communication of the modules M1 and M2 over $\mathsf{C}_{\text{Refined}}$ by either:
   (a) wrapping $\mathsf{C}_{\text{Refined}}$ in a way that the resulting channel $\mathsf{C}_{\text{Wrapped}}$ provides the interfaces required by M1 and M2 (*wrapping*) or
   (b) refining M1 into $\mathsf{M1}_{\text{Refined}}$ and M2 into $\mathsf{M2}_{\text{Refined}}$ in order their required interfaces match the ones provided by $\mathsf{C}_{\text{Refined}}$ (*adapter-merging*)

The two cases (not the only ones) are similar. Both include an intermediate step to build two further modules—i.e., two hierarchical channels called *adapters*—to map one interface to another: one, let us say A1, between M1 and $\mathsf{C}_{\text{Refined}}$, and one, let us say A2, between $\mathsf{C}_{\text{Refined}}$ and M2. In case (a), the resulting channel $\mathsf{C}_{\text{Wrapped}}$ encloses $\mathsf{C}_{\text{Refined}}$ and the two adapters, while in case (b) these adapters are merged to the calling modules M1 and M2, resulting in the refined modules $\mathsf{M1}_{\text{Refined}}$ and $\mathsf{M2}_{\text{Refined}}$. Deciding whether to use wrapping or merging depends on the methodology and chosen target architecture.

Example We show here how to apply the described communication refinement strategy to a simple abstract (functional timed) producer/consumer system taken from [40].

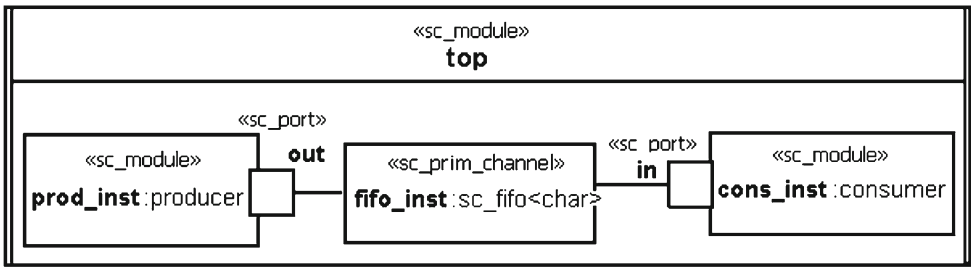


Fig. 13 A producer/consumer design

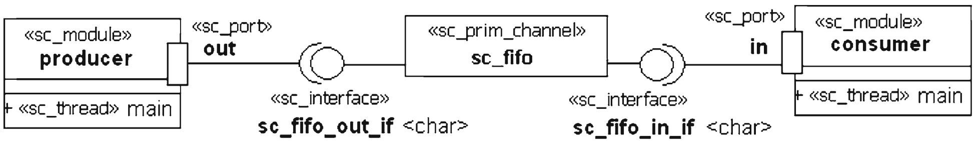


Fig. 14 Producer/consumer modules communicating via a primitive FIFO

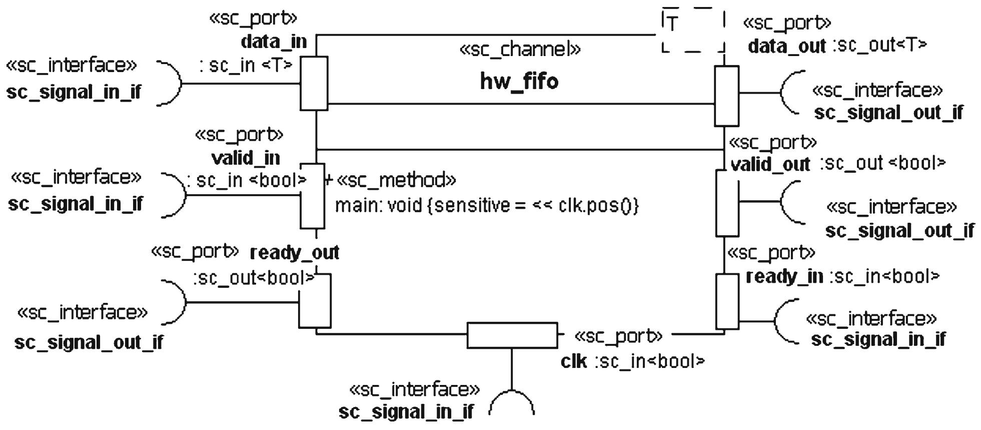


Fig. 15 Clocked RTL HW FIFO

The design of a producer/consumer module that writes and reads characters to/from a FIFO channel is shown by the UML composite structure diagram in Fig. 13. The top composite module is defined to contain one instance of the consumer module, one instance of the producer module, and one FIFO channel instance. The FIFO channel permits to store characters by means of blocking read and write interfaces, such that characters are always reliably delivered. Two processes, the producer and the consumer (see the thread processes main within the producer and the consumer modules in Fig. 14), respectively, feed and read the FIFO. The producer module writes data through its out port into the FIFO by a sc_fifo_out_if interface, and the consumer module reads data from the FIFO through its in port by the sc_fifo_in_if interface. These two interfaces are implemented by the FIFO channel (see the sc_fifo channel in Fig. 14). Because of the blocking nature of the sc_fifo read/write operations, all data are reliably delivered despite the varying rates of production and consumption.

Now, let us assume that the (abstract) FIFO instance above is replaced with a (refined) model of a clocked RTL hardware FIFO named hw_fifo<T> for an hardware implementation. The new hardware FIFO uses a signal-level ready/valid handshake protocol for both the FIFO input and output (Fig. 15). It should be noted that we cannot use hw_fifo directly in place of sc_fifo, since the former does not provide any interfaces at all, but has ports that connect to signals, i.e., has ports that use the sc_signal_in_if and sc_signal_out_if interfaces.

Following the wrapper-based approach (3b) described above, we can define a hierarchical channel hw_fifo_wrapper<T> (C_{Wrapped}) implementing the interfaces sc_fifo_out_if and sc_fifo_in_if and containing an instance of hw_fifo<T> (C_{Refined}). In addition, it contains sc_signal instances to interface with hw_fifo<T> and a clock port (since hw_fifo<T> has also a clock port) to feed in the clock signal to the hw_fifo<T> instance (Fig. 16). Finally, we need to add a hardware clock instance in the top-level design to drive the additional clock port that is now on the hw_fifo_wrapper<T> instance (Fig. 17). The hw_fifo_wrapper<T> implements the required signal-level ready/valid handshake protocol whenever a read or write operation occurs; this

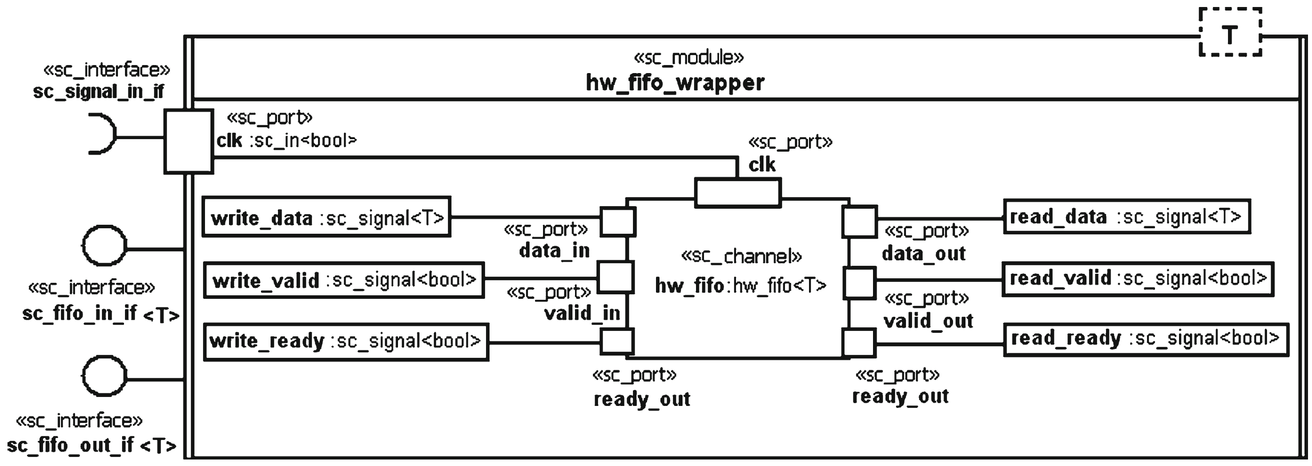
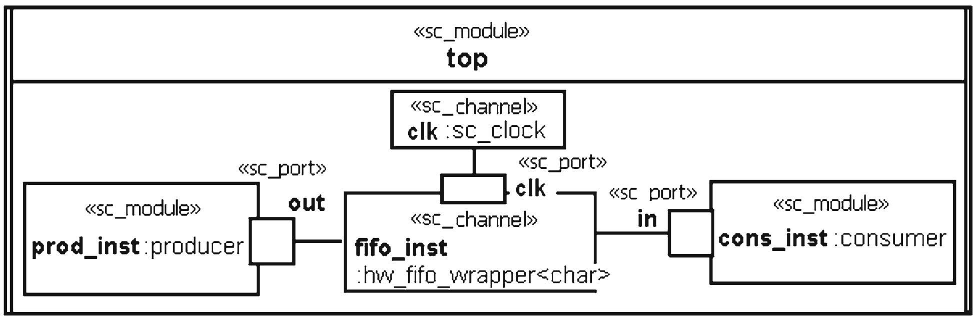


Fig. 16 The hw_fifo_wrapper hierarchical channel

Fig. 17 A clocked producer/consumer design



protocol will properly suspend read or write transactions if hw_fifo<T> is not ready to complete the operation. Details on the SystemC code can be found in [40].

6.2 Model-to-code transformations

Model-to-code transformations are primarily aimed at providing executable models at a fixed abstraction level. They must be intended as *horizontal* transformations creating program text in the SystemC implementation language.

In the model-to-code category, we distinguish between *visitor*- and *template*-based approaches. Visitor-based simply means that the model is navigated and code is generated for each element visited according to specific rules describing for each metaconcept what code to generate and which elements to visit next. Template-based approaches use target text that includes place holders that are to be replaced with concrete data in concrete template instances. Rules in such template-based approaches usually access information in the source model using the left-hand side of a rule and give a template as right-hand side that incorporates this data in a piece of target code.

We adopted a visitor-based approach for the EA-based environment. The EA supports forward/reverse engineering to/from C++. We developed an EA add-in (based on the EA automation and scripting interface and on the Windows OLE Automation ActiveX technology) which exploits the added semantics in the SystemC UML profile to generate complete SystemC code (*full code generation*) from input models (for both the structural and behavioral views) developed with the EA-based UML modeler.

The EA-based code generator traverses all class diagrams and for every encountered class it produces a C++ header file (.h); this happens for both the definition of SystemC classes (modules, channels, and interfaces) and simple C++ classes, allowing a mixed design style. Classes contain fields and methods. For each method it is possible to describe its behavior either as an inline code description or as a SystemC process state machine diagram. Each process state machine, therefore, contributes to the generation and enrichment of a body file (.cpp) containing the implementation code of all methods of a class or module or channel.

Moreover, by an analysis of the composite structure diagram associated to a module to describe its internal structure (especially the one of the topmost-level module, which represents the structure of the overall system), it is possible to determine how internal parts are connected to each other in the module constructors in the header file. Finally, object

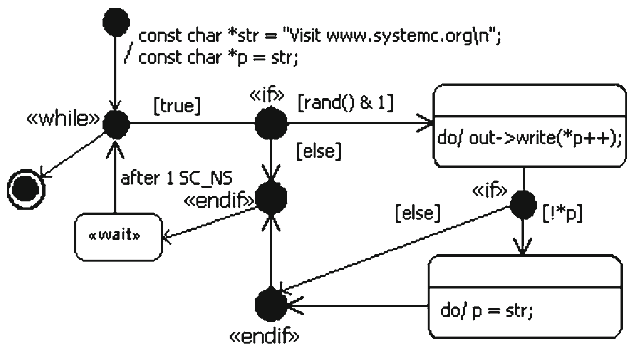


Fig. 18 Producer's main thread state machine and its SystemC code

diagrams may be seen as instances of the composite structure diagrams, useful to model particular initializations of the system components for the construction of appropriate test benches.

The header file `producer.h`, e.g., for the `producer` module (Fig. 14) would contain the declaration of the `producer` class together with the declaration of the `main` thread process, as reported in Listing 1.

Listing 1 producer.h

```
#include "read_write_if.h"
class producer: public sc_module {
 public:
  sc_port<sc_fifo_out_if<char> > out;
    void main();
    SC_HAS_PROCESS(producer);
    producer(sc_module_name mn): sc_module(mn)
    {
      SC_THREAD(main);
    }
};
```

The body section (file `producer.cpp` in Fig. 18) is determined by the thread state machine `main` (Fig. 18), where the C++ notation is put on top of the UML action semantics and the control structures are directly derived from the stereotyped pseudostates `while`, `if-then`, etc., of the state machine diagram.

We have been applying similar code engineering techniques to the Papyrus-based environment, but following a template-based approach by relying on JET and Acceleo technologies. Compared with visitor-based approaches, template-based approaches have the advantage of reusing pieces of code and thus template-based tools are less error-prone than visitor-based ones.

7 Related work

In this paper, we provide guidance on how to integrate a UP style of software development into the overall process for embedded systems. The UPES reflects current industry best practices and follows the *platform-based design* principles [15,45]. For the UPES definition, we have been taking in consideration the work experiences in [11,15,44,46] as first contributions to the definition of a development process for the embedded systems domain based on abstract and executable models and on the system-on-chip (SoC) paradigm for software–hardware convergence.

The possibility to use UML 1.x for system design [16] has emerged since 1999, but general opinion at that time was that UML was not mature enough as a system design language. Nevertheless significant industrial experience using UML in a system design process soon started to lead to the first results in design methodology, such as the one in [41] that was applied to an internal project for the development of an orthogonal frequency-division multiplexing (OFDM) wireless LAN chipset. In this project SystemC was used to provide executable models.

More integrated design methodologies were later developed. The authors of [28] propose a methodology using the UML for the specification and validation of SoC design. They define a flow, parallel to the implementation flow, which is focused on high-level specs capture and validation. In [15], a UML profile for a *platform-based* approach to embedded software development is presented. It includes stereotypes to represent platform services and resources that can be assembled together. The authors also present a design methodology supported by a design environment, called Metropolis, where a set of UML diagrams (use cases, classes, state machines, activity, and sequence diagrams) can be used to capture the functionality and then refine it by adding models of computation.

Another approach to the unification of UML and SoC design is the hardware and software objects on chip (HASoC) [11] methodology. It is based on the UML-RT profile [36] and on the Rational Unified Process (RUP) [14]. The design process starts with an *uncommitted model*, after which a *committed model* is derived by partitioning the system into software and hardware, and then mapped onto a system platform. From these models a SystemC skeleton code can be also generated, but to provide a finer degree of behavioral validation, detailed C++ code must be added by hand to the skeleton code. All the works mentioned above could greatly benefit from the use of UML2.

In [9], the authors present a model-driven framework called model-driven design of embedded systems (ModES) made of metamodels definition and APIs to integrate, by model transformations, several model-based design tools. However, this framework is more related to the design space exploration at a high abstraction level than to model refinement, model validation, and automatic code generation from models, which are, instead, our main concerns.

SysML [39] is a conservative extension of UML 2.0 for a domain-neutral representation (i.e., a PIM model as in MDA [17]) of *system engineering* applications. It can be involved at the beginning of the design process, in place of the UML, for the requirements, analysis, and functional design workflows. So it is in agreement with our UML profile for SystemC, which can be thought of (and effectively made) as customization of SysML rather than UML. Unluckily, when we started, the SysML specification was not yet finalized and there were no tools yet supporting it. Similar considerations apply also to the recent modeling and analysis of real-time embedded systems (MARTE) profile initiative [37].

The standardization proposal [42] by Fujitsu, in collaboration with IBM and NEC, has evident similarities with our SystemC UML profile, such as the choice of SystemC as a target implementation language. However, their profile does not provide building blocks for behavior modeling and any time model.

Some other proposals already exist for extensions of UML towards C/C++/SystemC. All have in common the use of UML stereotypes for SystemC constructs, but do not rely on a UML profile definition. In this sense, the work in [5] attempting to define a UML profile for SystemC is appreciable; but, as with all the other proposals, it is based on the previous version of UML (UML 1.4). Moreover, in all the proposals we have seen, except in [19], no code generation from behavioral diagrams is considered.

Refinement, or in general model-to-model transformation, is another key concept in MBD. However, compared with the refinement techniques available for formal methods such as Z, B, and Abstract State Machines (ASMs) [4], little work has been carried out for modeling languages such as UML. Some proposals that we are considering in our process can be found in [7,22,24–26].

8 Conclusions and future directions

This paper describes aspects of reusable model-to-model and model-to-code transformations in the context of the UPES/UPSoC processes for embedded system development. This aspect is vital for integrating, synchronizing or transforming models for our design methodology based on the UPES/UPSoC processes.

Today, model transformations are mainly written from scratch. This is in many cases a very time-consuming and difficult task. According to the MBD vision, there must be large libraries of reusable model transformations available.

In the future, we aim to identify characteristics of reusable transformations and ways of achieving reuse by collecting in a library precise abstraction/refinement transformation patterns according to the levels of abstraction: functional, transactional, behavioral, BCA, and RTL. In particular, we are focusing on the TLM to model the communication aspects at a certain number of TLM sublevels according to the OSCI TLM 1.0 library [21]. We believe that the use of fine-grained transformations that are being composed (chaining) would be beneficial for increasing both the productivity and quality of the developed systems.

References

1. SystemC Language Reference Manual (2006). IEEE Std 1666-2005, 31 March 2006
2. Arlow J, Neustadt I (2002) UML and the unified process. Addison-Wesley, Reading
3. Bocchio S, Riccobene E, Rosti A, Scandurra P (2008) An enhanced systemc uml profile for modeling at transaction-level. In: Villar E (ed) Embedded systems specification and design languages
4. Börger E, Stärk R (2003) Abstract state machines: a method for high-level system design and analysis. Springer, Berlin
5. Bruschi F, Sciuto D (2002) SystemC based design flow starting from UML model. In: Proceedings of European SystemC users group meeting
6. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of CODES+ISSS, Newport Beach, California, USA
7. The Catalysis Process (1998). http://www.catalysis.org
8. Czarnecki K, Helsen S (2003) Classification of model transformation approaches. In: Proceedings of 2nd OOPSLA workshop on generative techniques in the context of model-driven architecture
9. do Nascimento FAM, Oliveira MFS, Wagner FR (2007) ModES: embedded systems design methodology and tools based on MDE. In: Fourth International workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES'07). IEEE Press (2007)
10. The Enterprise Architect Tool (2008). http://www.sparxsystems.com.au/
11. Edwards M, Green P (2003) UML for hardware and software object modeling. UML for real design of embedded real-time systems, pp 127–147
12. Eclipse Modeling Framework (2008). http://www.eclipse.org/emf/
13. Jouault F, Allilaire F, Bézivin J, Kurtev I, Valduriez P (2006) ATL: a QVT-like transformation language. In: OOPSLA '06: companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. ACM, pp 719–720
14. Kruchten P (1999) The rational unified process. Addison-Wesley, Reading
15. Lavagno L, Martin G, Sangiovanni Vincentelli A, Rabaey J, Chen R, Sgroi M (2003) UML and platform based design. UML for real design of embedded real-time systems
16. Martin G (1999) UML and VCC. White paper, Cadence Design Systems, Inc, Dec.
17. OMG (2003) The Model Driven Architecture (MDA). MDA Guide V1.0.1, http://www.omg.org/mda/
18. Mens T, Wermelinger M, Ducasse S, Demeyer S, Hirschfeld R, Jazayeri M (2005) Challenges in software evolution. In: Proceedings of the international workshop on software evolution. IEEE
19. Nguyen KD, Sun Z, Thiagarajan PS, Wong WF (2005) Model-driven SoC design: the UML-SystemC bridge. UML for SOC Design
20. OMG. UML 2.0 OCL Specification, ptc/03-10-14
21. The Open SystemC Initiative (2008). http://www.systemc.org
22. Paige RF, Kolovos DS, Polack FAC (2005) Refinement via consistency checking in MDA. In: Proceedings Refinement Workshop, ENTCS, Surrey, UK, April
23. Papyrus UML Web Site http://www.papyrusuml.org (2008)

24. Pons C, Kutsche R-D (2003) Using UML-B and U2B for formal refinement of digital components. In: Proceedings of forum on specification and design languages, Frankfurt
25. Pons C, Kutsche R-D (2004) Traceability across refinement steps in UML modeling. In: Proceedings of the WiSME@UML workshop
26. Pons C, Garcia D (2006) An OCL-based technique for specifying and verifying refinement-oriented transformations in MDE. In: MoDELS, pp 646–660
27. OMG (2007) MOF Query/Views/Transformations, ptc/07-07-07
28. Zhu Q, Oishi R, Hasegawa T, Nakata T (2004) System-on-chip validation using UML and CWL. In: Proceedings of CODES
29. Riccobene E, Scandurra P (2004) Modelling systemc process behaviour by the UML method state machines. In: Proceedings of RISE'04. Springer, Heidelberg
30. Riccobene E, Scandurra P, Rosti A, Bocchio S (2005) A UML 2.0 profile for SystemC: toward high-level SoC design. In: EMSOFT '05: Proceedings of the 5th ACM international conference on embedded software. ACM Press, pp 138–141
31. Riccobene E, Scandurra P, Rosti A, Bocchio S (2006) A model-driven co-design flow for embedded systems. In: FDL '06: Proceedings of forum on specification and design languages
32. Riccobene E, Scandurra P, Rosti A, Bocchio S (2006) A model-driven design environment for embedded systems. In: DAC '06: Proceedings of the 43rd annual conference on design automation. ACM, New York, pp 915–918
33. Riccobene E, Scandurra P, Rosti A, Bocchio S (2007) Designing a unified process for embedded systems. In: Fourth international workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES'07). IEEE
34. Riccobene E, Scandurra P, Rosti A, Bocchio S (2007) A UML2 profile for SystemC 2.1. STMicroelectronics Technical Report, April
35. Schmidt DC (2006) Guest editor's introduction: model-driven engineering. Computer 39(2):25–31
36. Selic B (2000) A generic framework for modeling resources with UML. In: Proceedings of the 16th symposium on integrated circuits and systems design (SBCCI'03). IEEE Computer Society, vol 33, pp 64–69
37. De Simone R, et al. MARTE: A new profile RFP for the modelling and analysis of real-time embedded systems. In: UML for SoC design workshop at DAC'05
38. OMG (2008) SPEM, formal/08-04-01
39. OMG (2007) SysML, Version 1.0, formal/2007-09-01. http://www.omgsysml.org/
40. Gröetker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer, Dordrecht
41. Moore T, Vanderperren Y, Sonck G, van Oostende P, Pauwels M, Dehaene W (2002) A design methodology for the development of a complex system-on-chip using uml and executable system models. In: Forum on specification and design languages. ECSL
42. Fujitsu Limited, IBM, NEC (2005) A UML extension for SoC. Draft RFC to OMG, 2005-01-01
43. Vanderperren Y, Dehaene W (2005) A model-driven development process for low power soc using UML. UML for SOC design
44. Vanderperren Y, Pauwels M, Dehaene W, Berna A, Ozdemir F (2003) A systemc based system on chip modelling and design methodology. SystemC: methodologies and applications
45. Sangiovanni Vincentelli A (2002) Defining platform-based design. EEDesign, February
46. Zhu Q, Matsuda A, Kuwamura S, Nakata T, Shoji M (2002) An object-oriented design process for system-onchip using UML. In: Proceedings of the 15th international symposium on System Synthesis, Kyoto, Japan