SystemC/C-Based Model-Driven Design for Embedded Systems

ELVINIA RICCOBENE
Università degli Studi di Milano
PATRIZIA SCANDURRA
Università degli Studi di Bergamo
SARA BOCCHIO and ALBERTO ROSTI
STMicroelectronics
LUIGI LAVAZZA
Università dell'Insubria and CEFRIEL
and
LUIGI MANTELLINI
Dial Face Industry

This article summarizes our effort, since 2004 up to the present time, for improving the current industrial Systems-on-Chip and Embedded Systems design by joining the capabilities of the unified modeling language (UML) and SystemC/C programming languages to operate at system-level. The proposed approach exploits the OMG model-driven architecture—a framework for Model-driven Engineering—capabilities of reducing abstract, coarse-grained and platform-independent system models to fine-grained and platform-specific models. We first defined a design methodology and a development flow for the hardware, based on a SystemC UML profile and encompassing different levels of abstraction. We then included a multithread C UML profile for modelling software applications. Both SystemC/C profiles are consistent sets of modelling constructs designed to lift the programming features (both structural and behavioral) of the two coding languages to the UML modeling level. The new codesign flow is supported by an environment, which allows system modeling at higher abstraction levels (from a functional executable level to a register transfer level) and supports automatic code-generation/back-annotation from/to UML models.

Author's addresses: E. Riccobene, Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Crema (CR), Italy; email: elvinia.riccobene@unimi.it; P. Scandurra, Dip. di Ingegneria dell'Informazione e Metodi Matematici, Università degli Studi di Bergamo, Dalmine (BG), Italy; email: patrizia.scandurra@unibg.it; S. Bocchio and A. Rosti, STMicroelectronics Lab R&I, Agrate Brianza, Italy; emails: {sara.bocchio, alberto.rosti}@st.com; L. Lavazza, Dip. di Informatica e Comunicazione, Università dell'Insubria, Varese and CEFRIEL, Milan, Italy; email: luigi.lavazza@uninsubria.it; L. Mantellini, Industrie Dial Face S.p.A. Milan, Italy; email: luigi.mantellini@gmail.com

DOI 10.1145/1550987.1550993 http://doi.acm.org/10.1145/1550987.1550993

Categories and Subject Descriptors: I.6.0 [**Simulation and Modeling**]: General; I.6.5 [**Simulation and Modeling**]: Model Development; D.2.3 [**Software Engineering**]: Design Tools and Techniques; B.0 [**Hardware**]: General

General Terms: Model-driven engineering, embedded systems and SoC design, SystemC UML profile, multithread C UML profile, model refinement

Additional Key Words and Phrases: MDE, ES, SoC, UML, SystemC, C

ACM Reference Format:

Riccobene, E., Scandurra, P., Bocchio, S., Rosti, A., Lavazza, L., and Mantellini, L. 2009. SystemC/C-based model-driven design for embedded systems. ACM Trans. Embedd. Comput. Syst. 8, 4, Article 30 (July 2009), 37 pages.
DOI = 10.1145/1550987.1550993 http://doi.acm.org/10.1145/1550987.1550993

1. INTRODUCTION

Embedded system (ES) and system-on-chip (SoC) designs has to cope with short product cycles and a steadily increasing system complexity. A SoC may involve the mixing on a single integrated circuit of one or more microcontrollers, microprocessor or digital signal processor cores, memory blocks, timing sources, peripherals, buses for the interconnection, and so on, but also of the software that controls all these hardware components. Moreover, in the design of SoCs, the availability of intellectual property blocks (IPs) shifts the focus of the design practice from the synthesis of the functionality, where the single hardware block is synthesized, to the design of the platform where the final application is going to be implemented.

A platform is a hardware artifact where the design emphasis is put on its compositional architecture, rather than on deriving the whole functionality directly in hardware. In a platform-based design [Vincentelli 2002; Keutzer et al. 2000], the functionality of the application is implemented mostly via software that is then mapped on the platform. The platform-based design is essentially a meet-in-the-middle process where successive refinements of specifications meet with abstractions of potential implementations. Therefore, during the development process, parts that are in the software domain can move to be part of the platform as hardware coprocessors and vice versa in order to refine the platform template to the customers' goal.

This novel development approach, based on the reuse of hardware and software IPs, is driving fundamental changes in the design process. It is no more possible to tackle the design of systems at a low level of abstraction: The hardware behavioral and the register transfer level (RTL) are inadequate because they deal only with hardware. The amazing rise in complexity of actual systems requires a modular, component-based approach to both hardware and software design. This leads to development techniques working at higher levels of abstraction and in a hardware-software codesign environment handling both the hardware architecture and the application software.

A new way is needed to describe an entire system, including embedded software, and to formalize a set of constraints and requirements. What is needed is a "lightweight environment" where the application software is mainly described in an algorithmic way, better by modeling than by coding, and the interactions between the software and the underlying platform can also be described in the same environment. In this way, the description of the platform also has to take care of the effects that the hardware resources induce on the application software performance.

In Riccobene et al. [2005a, 2006a, 2007b], we tried to improve the current industrial SoC and embedded system design by defining a model-driven co-design methodology in accordance with the Object Management Group (OMG) model-driven architecture (MDA) [MDA 2003]—a framework for model-driven engineering (MDE) [Bézivin 2005; 2004], according to which models are primary driving assets of system development, including system design, platform, language definition, and automated mapping of models to implementations. The proposed codesign flow supports the platform-based design principles in Vincentelli [2002], Lavagno et al. [2003], and Keutzer et al. [2000]. It involves the OMG standard unified modeling language (UML) 2.1.2 [UML 2008], UML profiles[1] for the multithread C and the SystemC languages [Gröetker et al. 2002], and some other UML profiles related to the high-level modeling of the system.

The UML profiles for SystemC and multithread-C are the key points of this design methodology. The UML capability of unifying the specification and the design of the hardware and software parts of systems allows the settlement of the desired lightweight modeling environment, specifically tailored to early stages of design as front-ends for consolidated lower level hardware-software codesign tools.

In this article, we give a complete and detailed view of the proposed codesign flow for embedded systems based on the use of modeling languages and model transformations to define and reduce abstract, coarse-grained and platform-independent system models to fine-grained and platform-specific models. The improvement of the current industrial SoC and embedded system design methodology—obtained by joining the capabilities of the UML and the SystemC/C programming languages to operate at system-level—has been demonstrated through the application to industrial case studies of different complexity. This is the first time that our work is described encompassing all the involved aspects, namely methodology, design flow, modeling notations, and tools, in a revised and integrated perspective.

The remainder of this article is organized as follows. In Section 2, we quote some related work. In Section 3, we describe our model-based design flow. In Sections 4 and 5, we present the two UML profiles for the hardware and the application software description, respectively. Section 6 illustrates the notion of model refinement (carried out at UML level) focusing, in particular, on the communication refinement aspect. In Section 7, we present the architecture of a design environment that we developed to assist the designer across the UML modeling activity. In Section 8, we discuss some industrial case studies. Finally, in Section 9, we conclude the article by sketching some future directions of our contribution.

[1]Application-specific UML customizations.

2. RELATED WORK

The OMG standard UML [UML 2008] has received wide acceptance in software engineering during recent years. Meanwhile, significant investigations on how to apply UML for real-time systems [Lavagno et al. 2003] and professional development environments with UML support became available in that area.

Recently, the UML2 [UML 2008] and its extension mechanism are receiving significant interest in the embedded system community, since it allows UML customizations toward the definition of a family of languages (UML profiles) targeted to specific application domains (telecommunications, aerospace, real-time computing, automotive, SoC, etc.). This is confirmed by current standardization activities controlled by the OMG such as the Schedulability, Performance, and Timing Analysis (SPT) profile [SPT 2003]; the recent UML for SoC Forum (USoC) [USoC 2006] founded by Fujitsu, IBM/Rational, and CATS to define a set of UML extensions to be used for SoC design; the SysML proposal [SysML 2007], which extends UML toward the systems engineering domain; and the recent modeling and analysis of real-time embedded systems (MARTE) profile initiative [MARTE 2008]. Moreover, several reported experiences and contributions to the theme UML for embedded systems exist in literature (see UML-SoC Workshops [2008], Martin and Mueller [2005], and ECSI UML Workshop [2006]) where different UML diagrams (to be intended as visual formalisms) and their variations found their application in requirements specification, test benches, architectural descriptions, and behavioral modeling.

The use of the UML 1.x for system design [Martin 1999] started since 1999, but the general opinion at that time was that the UML was not mature enough as system design language. Nevertheless, significant industrial experiences using the UML in a system design process soon started leading to the first results in design methodology, such as the one in Moore et al. [2002] where the UML was applied for the development of an OFDM Wireless LAN chipset. In this project, SystemC was used to provide executable models.

Later, more integrated design methodologies were developed. In Zhu et al. [2004], the authors 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 requirements capture and validation 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 towards the use of the UML for SoC design is the hardware and software objects on chip (HASoC) [Edwards and Green 2003] methodology. It is based on the UML-RT profile [Selic 2000] and on the RUP process [Kruchten 1999]. The design process starts with an uncommitted model; then a committed model is derived by partitioning the system into software and hardware; finally, it is 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 earlier could greatly benefit from the use of new constructs available in the UML 2.x (or UML2).

SysML [SysML 2007] is a conservative extension of the UML2 for a domain-neutral modeling 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. Since SysML preserves the basic semantics of UML2 diagrams (state formalism, e.g., are unchanged), our SystemC UML profile can be also intended (and effectively made) a customization of the SysML language rather than the UML. Similar considerations also apply to the MARTE proposal [MARTE 2008]. The standardization proposal [USoC 2006] by Fujitsu, in collaboration with IBM and NEC, has evident similarities with our SystemC UML profile, like the choice of SystemC as target implementation language. However, their profile deals only with structural modeling, and provides neither constructs/formalisms for behavior modeling nor does it adopt a time model.

Some other proposals already exist about extensions of UML toward C/C++/SystemC. All have in common the use of UML stereotypes for SystemC constructs, but they do not rely on a UML profile definition. In this sense, it is appreciable the work in Bruschi and Sciuto [2002] attempting to define a UML profile for SystemC; however, as all the other proposals, it is based on the obsolete version 1.4. of UML, making difficult and little scalable modeling systems without the enhancements for structure/architecture modeling provided by the next versions 2.x. Moreover, in all these works, no code generation, except in Nguyen et al. [2005], from behavioral diagrams is supported. Some recent works look at generating SystemC code from UML diagrams. In Mura et al. [2007], the authors propose to derive RTL-SystemC code from UML statecharts. In Kreku et al. [2007], a technique is presented for generating a work-load model in SystemC from a UML model of the system for high-level performance analysis. In Raslam and Sameh [2007], a mapping from SysML to SystemC is proposed. All these approaches aim at obtaining a SystemC code that resembles the behavior of the UML model, whereas we extend the UML accordingly to the SystemC execution semantics.

In Zimmermann et al. [2008], a platform-based approach is proposed to model an embedded distributed system in a holistic way. Starting from a component description of microelectronic processor and interconnection resources in a standard IP-XACT format [IP-XACT 2007], the UML is adopted (similarly to our approach) as an underlying common data model for the system modeling process. By mapping the system behavior onto instances of the component library, an abstract system model can be generated and refined to an executable model in SystemC.

For the UML adoption in the context of multithread software applications few references exist in the literature. Moreover, multithread object-oriented programming languages, like Java [Leroux et al. 2003] or CORBA [Chen and Cui 2004], which are commonly used in this context, are not so common in the embedded software area. Therefore, these languages are not integrated in a comprehensive hardware-software codesign methodology for embedded systems with the goal of providing a high-level view of the application software only. The MARTE initiative [MARTE 2008] is working toward this direction, but it covers OSEK/VDX OS rather than the Posix specification. The Posix library is considered, instead, in another OMG standard, the UML Profile for Software Radio [UML Profile for SWRadio 2007]. However, this profile is too domain-specific, it deals with software radio applications only and is available only in form of specification.

3. THE MODEL-DRIVEN CODESIGN FLOW

The proposed design methodology, originally sketched in Riccobene et al. [2005a, 2007a] for the hardware components but here explained in the light of a global hardware-software codesign view, aims at applying the MDA approach to get for the embedded system development the same benefits as in software development. A MDA-based system development process relies on the notion of modeling and automated mapping of models to implementations. The basic MDA pattern involves the definition of a platform-independent model (PIM) and its automated mapping to one or more platform-specific models (PSMs).

Our model-driven codesign flow involves lightweight modeling notations based on the UML to be applied as high-level system modeling languages and operating in synergy with lower-level system languages for producing PIMs and PSMs. The methodology handles both the hardware architecture with the hardware-dependent software (HdS) components and the (hardware-independent) application software. To foster the methodology in a systematic and seamless way and combine all involved notations together, in Riccobene et al. [2007b] a design process, called unified process for embedded systems (UPES), is concisely presented as extension of the conventional unified process (UP)[2] [Arlow and Neustadt 2002]. UPES includes a subprocess, called unified process for SoC (UPSoC), for the hardware refinement flow.

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 [Vincentelli 2002; Lavagno et al. 2003; Keutzer et al. 2000]. Figure 1 summarizes the most significative activities of the UPES, which is a Y-shaped development process. On one side, the conventional UP process for software development is followed by adopting the UML—or the SysML or the MARTE proposal—to provide a ground executable PIM of the system (the application model) suitable for high-level functional validation and, possibly, performance analysis. On the other side, a generic hardware platform, modeled as PIM (the platform model) with an appropriate language (e.g., a particular UML extension for SoC design), of the final physical architecture is chosen among the available ones. Note that, since the architecture template may be implemented by assembling reusable hardware

[2]The UP is an open software engineering process from the authors of UML. The RUP (Rational Unified Process) is the most widely used commercial variant of UP.

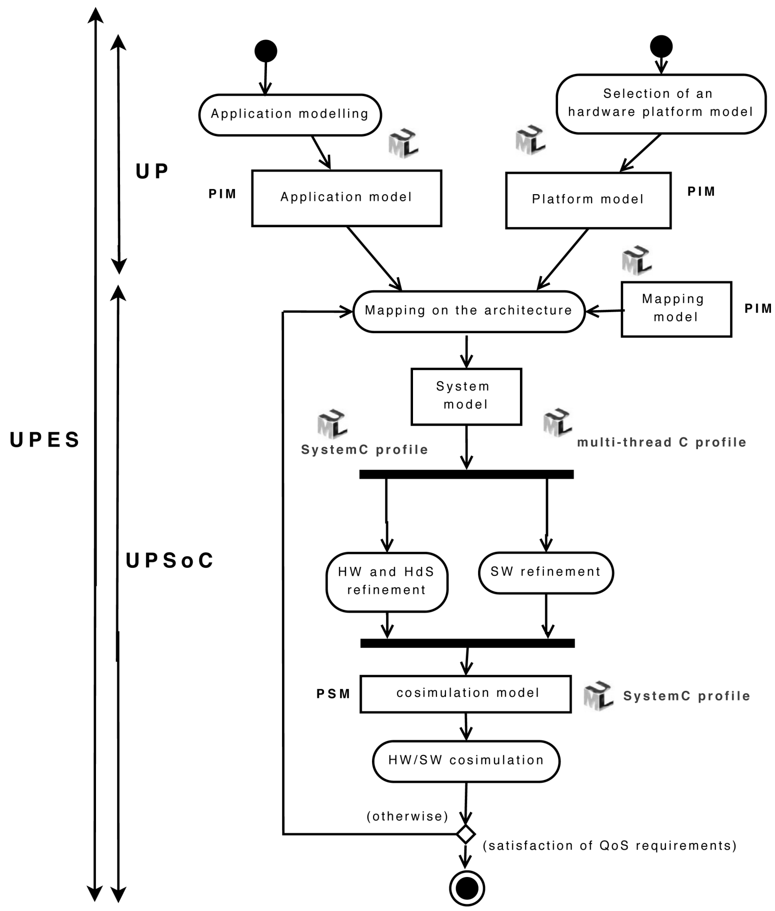


Fig. 1. UPES design flow.

and software IP components (possibly provided by third-party companies), the platform selection process may imply also an IP integration step comprising a set of tasks that are needed to assemble predesigned components in order to fulfill the desired SoC requirements.

The intersection, which is intended as a model weaving operation in the model-driven context, is the mapping of the application model on the given platform model to establish semantic links between the two models at specific joint points. As input, this task requires also a reference model of the mapping (the mapping model or weaving model) to try, which is specified—dictated by the sense and experience of expert engineers—in terms of UML component and deployment diagrams to denote and annotate the partitioning of the original system in hardware and software components. This mapping model establishes the relationships (joint points) of the platform resources and services with the application-level functional components. This mapping phase is a meet-in-the-middle process and is carried out through an iterative activity: The platform model is iteratively configured according to a given hardware-software partitioning and executed to check whether the QoS requirements (delays, power consumption, hardware resources, real-time and embedding constraints) imposed by the system requirements are satisfied.

This high-level system model, obtained as a result of the mapping phase, has a two-fold goal: (i) it can be used as proof of concept, so serving as reference model for the implementation by providing some guidelines; or (ii) it can be the starting point of a further model refinement phase down to the RTL and the software object code. This decision may drive the level of abstraction, the interface descriptions, and also the level of model granularity. In the UPES design flow, the system model is used in the second way. Indeed, after the mapping, two different refinement design flows start for the software and hardware parts, respectively. The software parts are described with the UML profile for multithread C. In this phase,[3] the software designer decides the thread partition according to the inner application parallelism. The functionality is therefore divided in threads running on a host machine—host functional simulation. For the hardware parts, as better described below, the UPSoC subprocess can be followed to refine the hardware and HdS components of the platform PIM into a sequence of PSMs through different levels of abstraction by using the modeling constructs of the UML profile for SystemC. This sequence of PSMs goes from a high-level functional untimed/timed model of the system down to a transactional model, to a behavioral model, to a bus-cycle accurate (BCA) model, to a final RTL model suitable for the synthesis of an end-product integrated into a chip.

These levels of abstraction should allow validating the correctness of the application software on a high-level description of the hardware architecture focusing on the communication and on its performances. This cosimulation activity is performed by executing an appropriate cosimulation model within the SystemC simulation and debugging environment. The cosimulation model (from which the SystemC code is generated) is the UML PSM model of the hardware combined with the UML PSM model of the application software at a fixed level of abstraction. Essentially, the cosimulation can be carried out at transactional or at instruction level, and depending on the level of abstraction that is used for the model, synchronization and communication are handled in a different way. At transactional level, the application software model works as a library encapsulated in a SystemC UML module element; processes are associated to the software functional description to sustain its concurrent activity within the system, whereas communication is implemented by transactions that model the interactions with the hardware architecture—transactional cosimulation. At instruction level, instruction set simulators (ISSs) of the target processing units are integrated within the SystemC environment to execute the object code of the application software together with the SystemC coding of the hardware—cycle-accurate cosimulation. In this last case, the application software can be intended as organized in different layers on top of the architecture platform. The lower layer provides the driver and the architecture controller: This layer

[3]This phase does not necessarily imply a software refinement path, but just a mapping on more detailed platforms to better analyze the performances of the overall application.

is the application programming interface (API) platform and it's typically provided by the platform designer. The upper layer are the OS and the application software developed by the software designer.

Currently, we support only the cycle-accurate cosimulation. To support the transactional cosimulation, a (bidirectional) transformation bridge between the software and the hardware perspectives is needed. This implies modeling of complex concurrency aspects over threads and tacking model composition of SystemC threads and POSIX C threads through automated hardware-software interfaces (that essentially incapsulate C components in C++ components), which are still challenging for us. The issues to cope are more at conceptual level than at implementation level. At the implementation level, once identified, the hardware-software communication patterns are realized in terms of composition/transformation of the two PSM UML models involved (one conforming to the SystemC UML profile and the other one conforming to the multithread C UML profile).

There is always a trade-off, whether it is better to extend a language or to couple different languages with different abstraction levels. The described approach is in the middle. The involved UML profiles, in fact, must be understood as consistent sets of modeling constructs that lift both the structural and behavioral features of the SystemC/C programming languages to the UML modeling level, while providing "unification" in the overall UML modeling activity. The UML profile for SystemC allows using UML at PSM level, provides unification between PIM and PSMs, and allows automatic encoding of PSMs into final SystemC code. Similarly, for the software components, the UML profile for multithread C provides modeling constructs, which combine the UML capabilities to represent the ideas and the design concepts with the objects exported by the Posix Library to represent the real software concurrent environment. It gives a new graphical dimension to C, enforcing the designer to describe systems at "modeling level" rather than designing at a lower level by means of "code."

Using SystemC to link the system level SoC design flow to the consolidated VLSI design flow is a well-known issue. What is innovative is the idea to rely on low-cost customized (by a standard profiling technique) UML visual modeling tools as front-ends of lower-level hardware-software codesign frameworks (the last would be used therefore for the final exploration and synthesis only).

Model-Driven SoC Design Flow

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 hardware partition have been mapped directly onto the hardware resources of the selected platform. It is possible to create a pure functional model (see Figure 2), or to add timing information in a functional-timed model. A transactional model describes abstract communication by transactions, which are protocols exchanging complex data with associated timing information. Details of the implementation platform can be modeled by a behavioral model (that is pin and functionally accurate), by a bus cycle accurate model (that is

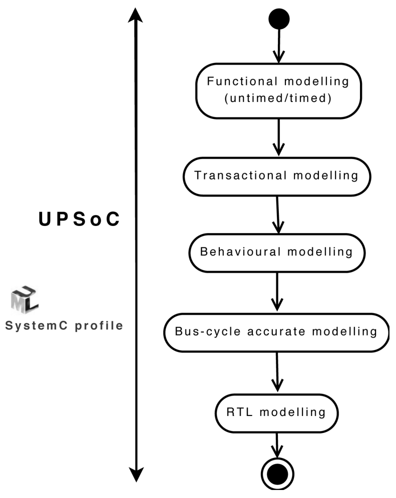


Fig. 2. SoC refinement flow.

also cycle accurate at its boundaries), or by a RTL model (the model is described as transfer of information between registers).

The UPSoC can be also adopted in a stand-alone way by platform providers to deliver off-line models of platforms[4] by designing from scratch an abstract hardware platform—from the analysis of informal requirements about the architecture elements (hardware resources and processing elements)—or by assembling reusable hardware and software IP components (possibly delivered by third-party companies). The model of this generic platform can be then delivered at a desired level of abstraction and then reused and targeted accordingly to implement a specific hardware architecture. Note that, as part of the UPSoC refinement process, the designer may need to adapt each component's functionality (a step commonly called IP derivation) and synthesize hardware and software wrappers to interconnect them (see Section 6). The generation of hardware and software wrappers is usually known as interface or communication synthesis. Besides this wrapper generation, application software may also need to be retargeted to the processors and OS of the chosen architecture.

The UPSoC process is based on model-to-code and model-to-model transformations from abstract models toward C/C++/SystemC code models and/or refined (but still abstract) models. 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 C/C++/SystemC implementation languages. Model-to-model transformations allow model refinement along different (not necessarily successive) abstraction levels (see Figure 2).

[4]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 API layer and the microarchitecture layer, respectively.

These model 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 (namely the UML metamodel extended with the stereotypes and constraints of the given profile). Each model is therefore a PSM model with a fixed action semantics.

The SystemC UML profile provides in a graphical way all modeling elements for the design of the hardware platform with the HdS components, and exactly resembles the modeling primitives of SystemC. However, system-level design driven by models using the SystemC UML profile is more flexible and manageable than code-oriented system-level design using SystemC. Indeed, well-established abstraction/refinement patterns coming from hardware/system engineering and until now only used at code level with great difficulty, can be easily managed—possibly automatically by appropriate model transformation engines—as model-to-model transformation steps and, therefore, applied at 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. Furthermore, proving correctness of the refinement process and system properties is a very hard, and we can say impressible, activity at code level, whereas it is feasible at the model level.

Currently, we are working [Gargantini et al. 2008b; Carioni et al. 2008] on complementing our model-driven design methodology with a formal analysis process for high-level system validation and verification. We have developed a method for simulation-based [Gargantini et al. 2009] and scenario-based [Carioni et al. 2009] validation of embedded system designs provided in terms of UML models. This approach is based on automatic model transformations from SystemC UML graphical models into abstract state machine (ASM) [Börger and Stärk 2003] formal models, and exploits ASM model validation by simulation and scenario construction. A validation tool integrated into the existing model-driven codesign environment [Riccobene et al. 2006b] to support the proposed validation flow is also available. It is based on the ASMETA toolset [Gargantini et al.2008a; ASMETA 2009]—a set of tools around ASMs.

4. THE UML PROFILE FOR THE HARDWARE

Our model-driven codesign methodology involves a UML2 profile for SystemC [Riccobene et al. 2005b, 2005c] as a modeling language for designing and refining hardware components. More precisely, our methodology complies with the latest version 2.1.2 of the UML [UML Superstructure 2007]; however, for simplicity in the rest of the article, we always refer to UML2.

A UML profile is a set of stereotypes, each defining how the syntax and the semantics of a specific UML metaclass (a class in the UML metamodel) is extended for a target application domain. Therefore, the UML2 profile for SystemC is a high-level visual language consisting of a set of modeling constructs designed to lift both structural and behavioral features of the SystemC language (including events and time features) to UML level. Furthermore, the profile allows us to model systems at different levels of abstraction (from a functional executable level to RTL).

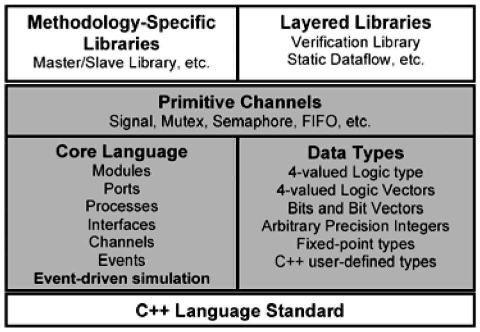


Fig. 3. SystemC language architecture.

The choice of lifting SystemC at UML level was intentional and due to the fact that SystemC [OSCI group 2008] is an open industry standard for the system-level hardware description, controlled by the major companies in the electronic design automation (EDA) industry. The main target of this profile is to provide a means for hardware design with UML at system-level and a direct encoding of this design in environment completely based on the C++ programming language.

Based on the UML2 specification and on the SystemC 2.1 specification, the basic SystemC profile is logically structured to reflect the core layer (or layer 0) of SystemC[5] according as the layered architecture of the language as shown in Figure 3 [Gröetker et al. 2002].

A structure and communication part defines stereotypes for the SystemC building constructs (modules, interfaces, ports, and channels) to be used in various UML structural diagrams, such as UML class diagrams and composite structure diagrams, to represent hierarchical structures and communication blocks.

UML class diagrams are used to define modules, interfaces, and channels. The internal hierarchical structure of composite modules, especially the one of the topmost level module (representing the structure of the overall system), is captured by UML composite structure diagrams; then, from these diagrams several UML object diagrams can be created to describe different configuration scenarios. This separation allows the specification (also partial) of different hardware platforms as instances of the same parametric model (i.e., the composite structure diagram).

A behavior and synchronization part defines state and action stereotypes, which lead to a variation of the UML state machine diagram, the SC process state machines. This formalism has been included in the profile definition to

[5]The *Core Language* and *Data Types* are the so-called core layer (or layer 0) of the standard SystemC; it consists of the event-based and discrete-timed SystemC simulation kernel, the core design primitives, and data types. The primitive channels represent, 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 fifo channels. Finally, the external libraries layer on top of the layer 1 are not considered as part of the standard SystemC language.

Table I. UML Diagrams in the SystemC Profile

Diagram	Purpose
Class diagram	To define modules, channels, interfaces, and port types
Composite structure diagram	To describe how parts of modules and channels are connected to each other to form the internal structure of a container module
State machine diagram	Used as *method* state machine to describe the reactive behavior of processes/operations of modules and channels
Object diagram	Derived from a composite structure diagram by instantiating parts and connectors, to describe a configuration (also partial) of the system

model the control flow and the reactive behavior of SystemC processes (methods and threads) within modules.

A data types part defines UML classes for representing the set of SystemC data types as UML class library.

In addition, also the predefined channels, ports, and interfaces of the layer 1 of SystemC are considered. These concepts 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—which specializes the basic profile. Table I summarizes the UML diagrams used in the profile.

The original UML profile definition for SystemC is in Riccobene et al. [2005b]. In the following sections, we briefly describe the most significant modeling elements (i.e., the available stereotypes) of the basic SystemC profile giving the semantics in an informal way and leaving out the OCL constraints, which usually complete a stereotype definition to add semantic constraints to its base UML metaelements.

A modeling example—the `Counter` system—is provided in the Appendix A to illustrate the notation, even if an idea of the modeling constructs is given through the producer/consumer system presented in Section 6 as model refinement example. To these purposes, we assume the reader is familiar with UML2.

4.1 Structure and Communication

Figure 4 shows the stereotypes definition using the standard notation of UML profiles. A stereotype is depicted as a class with the keyword ≪stereotype≫. The extension relationship between a stereotype and its UML metaclass is depicted by an arrow with a solid black triangle pointing toward the metaclass. Tags may be added to a stereotype to state additional metaclass properties.

When applied to an element in a model, a stereotype is shown as a keyword consisting in the name of the stereotype within a pair of guillemets, near the symbol of the element or with the special icon defined for it (if one was defined for it) in place of the conventional symbol for the element. Tagged values (if any) are displayed inside or close as name-value pairs. Figure 5 shows some examples of stereotypes application at model level.

A system design is essentially broken down into a containment hierarchy of modules. A module is a container class that provides the ability to encapsulate structure and functionality of hardware/software blocks. Each module may

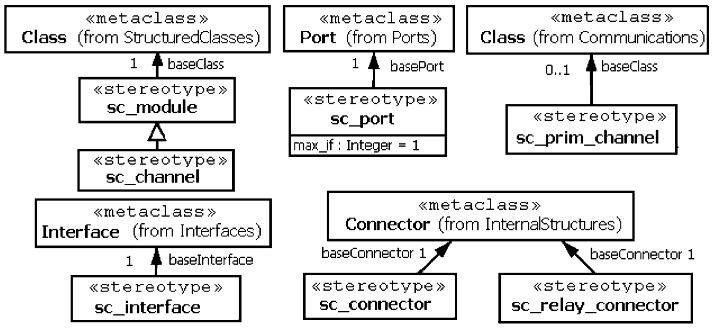

«metaclass» Class (from StructuredClasses)
1 baseClass
«stereotype» sc_module
«stereotype» sc_channel
«metaclass» Port (from Ports)
1 basePort
«stereotype» sc_port
max_if : Integer = 1
«metaclass» Class (from Communications)
0..1 baseClass
«stereotype» sc_prim_channel
«metaclass» Interface (from Interfaces)
1 baseInterface
«stereotype» sc_interface
«metaclass» Connector (from InternalStructures)
baseConnector 1
baseConnector 1
«stereotype» sc_connector
«stereotype» sc_relay_connector

Fig. 4. Structure stereotypes definition.

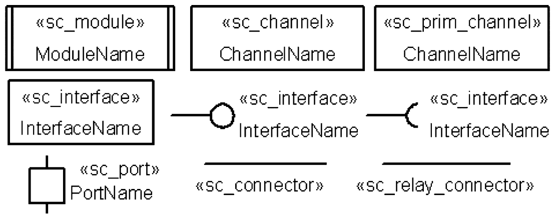

«sc_module» ModuleName
«sc_channel» ChannelName
«sc_prim_channel» ChannelName
«sc_interface» InterfaceName
«sc_interface» InterfaceName
«sc_interface» InterfaceName
«sc_port» PortName
«sc_connector»
«sc_relay_connector»

Fig. 5. Structure stereotypes notation.

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. Processes run concurrently in the design and may be sensitive to events, which are notified by other processes.

The sc_module stereotype defines a SystemC module as extension of a UML structured class. As structured class, a composite structure (diagram) can be further associated to a module to represent its internal structure (if any) made of channel and other module *parts*.[6] Furthermore, since modules contain reactive processes, modules are considered *active* classes.

In SystemC, an interface defines the set of access functions (methods) for a channel. The sc_interface stereotype defines a SystemC interface as a UML interface, and uses its (longhand/shorthand) notation.

A port of a module is a proxy object through which a process accesses to a channel interface. The sc_port stereotype maps the notion of SystemC port directly to the notion of UML port, plus some constraints to capture the concepts of simple, multi and behavior port. The tag max_if defined for the sc_port stereotype specifies the maximum number of channel interfaces that may be attached to the port. The type of a port, namely its required interface, is shown with the socket icon attached to the port. Figure 6 shows an example of a SystemC module having a multiport, an array port, and a simple port, together with the port type and interface definitions of the simple port.

[6]A property (or part) denotes a set of instances (in case of a property multiplicity greater than 1) that are owned by the structured module. These instances are instances (just a subset of the total set of instances) of the classifier (a module or a channel) typing the property.

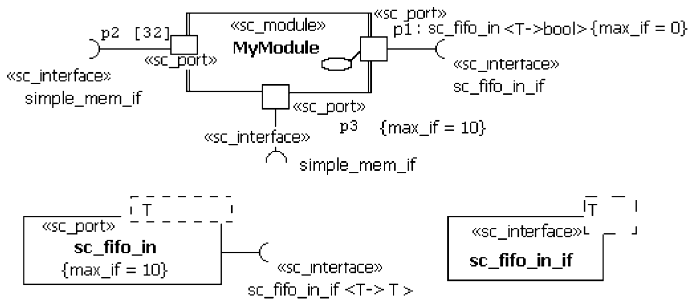


Fig. 6. Examples of a module, ports and interfaces.

Since UML ports are linked by connectors, a SystemC connector (binding of ports to channels) is provided as extension of the UML connector, by the `sc_connector` stereotype. A relay connector—the `sc_relay_connector` stereotype—is also defined to represent the port-to-port binding between parent and child ports.[7]

In SystemC, a channel provides the implementation of interface functions and serves 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, that is, it can have structure, it can contain processes, and it can directly access other channels. The `sc_prim_channel` and `sc_channel` stereotypes define, respectively, a SystemC primitive and hierarchical channel as extension of a simple UML class that implements a certain number of interfaces (i.e., the provided interfaces of the channel). A hierarchical channel can further have structure (including ports), it can directly access other channels, and it can contain processes.

A composite structure diagram can be further associated to a module (or to a hierarchical channel) to represent its internal structure (if any). In such a diagram, channel instances and module instances can appear as parts of the structured module. When an instance of the containing module is created, a set of instances corresponding to its parts are created. An example of composite structure diagram is given in Section A, Figure 30.

4.2 Behavior and Synchronization

Figure 7 shows the stereotypes definition for the SystemC constructs used to model the behavioral aspects of a system. Processes are the basic mechanism in SystemC for representing concurrent behavior. Two kinds of processes are available: method and thread, both behaving like an operation with no arguments and no return type. Clocked threads are a specialization of threads. Each kind of process has a slight different behavior, but basically all processes: run concurrently, are sequential, and are activated (if terminated or simply suspended) on the base of their own sensitivity, which consists of an initial list of

[7]A port-to-port connection is the binding of a module port (parent port) to a lower-level module port (child port).

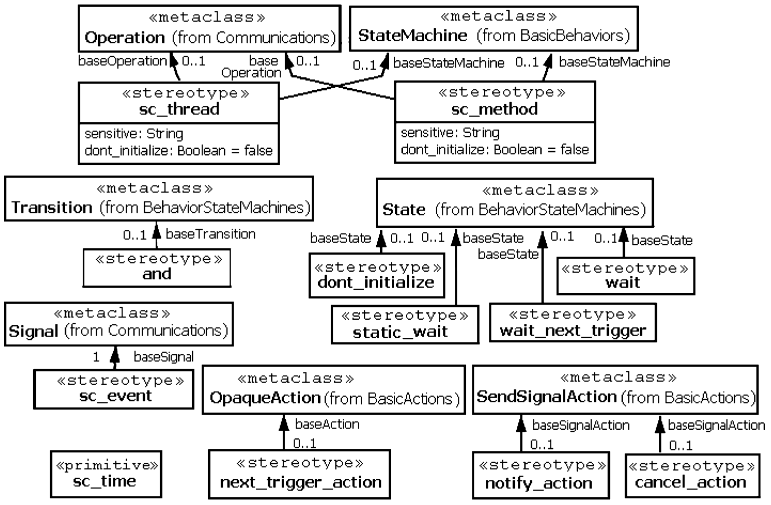

«metaclass» Operation (from Communications)
«metaclass» StateMachine (from BasicBehaviors)
baseOperation 0..1
base Operation 0..1
0..1 baseStateMachine
0..1 baseStateMachine
«stereotype» sc_thread
sensitive: String
dont_initialize: Boolean = false
«stereotype» sc_method
sensitive: String
dont_initialize: Boolean = false
«metaclass» Transition (from BehaviorStateMachines)
0..1 baseTransition
«stereotype» and
«metaclass» State (from BehaviorStateMachines)
baseState 0..1
0..1 baseState
baseState 0..1
0..1 baseState
«stereotype» dont_initialize
«stereotype» wait
«stereotype» static_wait
«stereotype» wait_next_trigger
«metaclass» Signal (from Communications)
1 baseSignal
«stereotype» sc_event
«metaclass» OpaqueAction (from BasicActions)
baseAction 0..1
«metaclass» SendSignalAction (from BasicActions)
baseSignalAction 0..1
baseSignalAction 0..1
«primitive» sc_time
«stereotype» next_trigger_action
«stereotype» notify_action
«stereotype» cancel_action

Fig. 7. Behavior stereotypes definition.

zero, one or more events (the static sensitivity of a process), and can dynamically change at runtime realizing the so-called dynamic sensitivity mechanism.

The SystemC UML profile defines two processes stereotype `sc_method` and `sc_thread` (see Figure 7); both extend the `Operation` and the `StateMachine` UML metaclasses. This double extension allows us to associate an operation to its behavior specified in terms of a (method) state machine.

Special state and action stereotypes are added to support the behavioral features mentioned earlier. Our profile provides stereotype definitions (`wait`, `static_wait`, `and`, `wait_next_trigger`, `next_trigger_action`, `dont_initialize`) to model the static and dynamic sensitivity mechanism of a process behavior (a thread or a method).

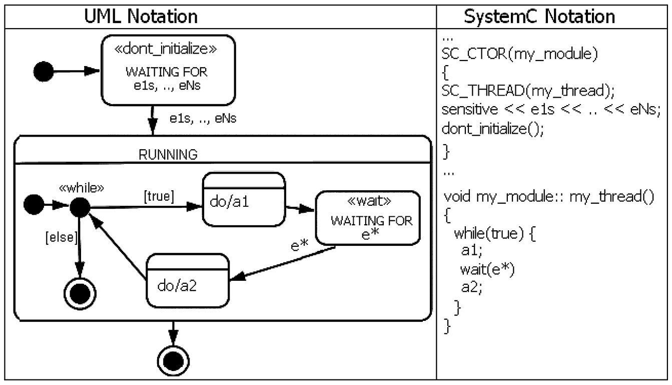
The `sc_event` stereotype models a SystemC *event* in terms of a UML signal (instance of the class `SignalEvent`).

The `notify_action` stereotype can be applied to an UML action to model the SystemC function `notify` used to notify events. The `cancel_action` stereotype can be applied to an UML action to model the SystemC function `cancel` used to eliminate a pending notification of an event.

The `sc_time` type is used to specify concrete time values used for setting clocks objects, or for the specification of UML time triggers.

As part of the UML profile for SystemC, the control structures `while`, `if-then-else`, and the like are represented in terms of stereotyped junction or choice pseudostates.

All these stereotypes and their associated OCL constraints lead to a variation of the UML state machine formalism called SystemC process state machines [Riccobene and Scandurra 2004]. This formalism allows modeling the control flow and the reactive behavior of processes (methods and threads) within modules, dealing with concurrency, synchronization and timing aspects, and allows the generation of efficient and compact executable SystemC code.



```
...
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. 8. A thread process pattern.

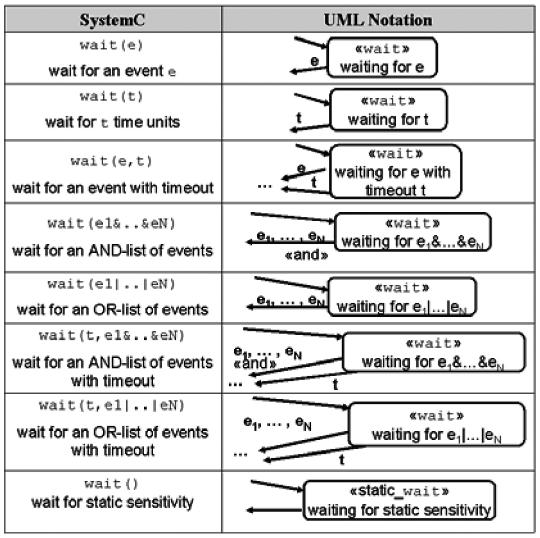
In addition to the stereotypes presented earlier, a finite number of abstract behavior patterns of state machines have been identified and can be used for modeling. Figure 8 depicts, for example, one of these behavior patterns together with the corresponding SystemC pseudocode for a thread that: (i) 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 state machine pattern given in Figure 8 is a shortcut to represent a generic `wait(e*)` call where the event `e*` matches several cases. Figure 9 shows how a `wait(e*)` call is modeled in UML for all possible forms of the condition `e*`: a single timed event, a single signal event, a single event with timeout, an AND-list of signal events, an OR-list of signal events, an AND-list of signal events with timeout, an OR-list of signal events with timeout, and the static sensitivity list. The stereotype `and` labeling the outgoing transition of a `wait` state denotes an AND-semantics for the list of events labeling the transition: The process leaves the state when all events in the list have been triggered (not necessarily at the same time).

We adopted the state machines rather than other UML behavioral diagrams (as the activity diagrams) because this kind of diagram provides a behavioral pattern appropriate for modeling the reactive and hierarchical behavior of SystemC processes, which are essentially activated by triggering external synchronization events. Moreover, according to the UML specification, state machines are sequential as far as their internal behavior is concerned, but any state machine is concurrent with respect to the other state machines of the system. Indeed, UML state machines can be used for modeling simple function calls that execute under the control of SystemC processes, and it is also possible to capture the SystemC synchronization mechanism for suspending/resuming a process in terms of stereotyped wait states and events.

4.3 Profile Extension

In 2006, SystemC received a major revision and became IEEE Standard [SystemC 2006]. This last revision includes new structural and behavioral



SystemC	UML Notation
`wait(e)` wait for an event `e`	e «wait» waiting for e
`wait(t)` wait for `t` time units	t «wait» waiting for t
`wait(e,t)` wait for an event with timeout	... e t «wait» waiting for e with timeout t
`wait(e1&..&eN)` wait for an AND-list of events	$e_1, \dots, e_N$ «and» «wait» waiting for $e_1\&\dots\&e_N$
`wait(e1|..|eN)` wait for an OR-list of events	$e_1, \dots, e_N$ «wait» waiting for $e_1\|\dots\|e_N$
`wait(t,e1&..&eN)` wait for an AND-list of events with timeout	$e_1, \dots, e_N$ «and» ... t «wait» waiting for $e_1\&\dots\&e_N$
`wait(t,e1|..|eN)` wait for an OR-list of events with timeout	$e_1, \dots, e_N$ ... t «wait» waiting for $e_1\|\dots\|e_N$
`wait()` wait for static sensitivity	«static_wait» waiting for static sensitivity

Fig. 9. Dynamic sensitivity of a thread.

features required for modeling at the transaction-level toward hardware-software implementation according to the OSCI (Open SystemC Initiative) TLM 2.0 standard. An extension of the SystemC UML profile was, therefore, necessary to align the profile definition with the standard IEEE.

Details on the structural enhancements (i.e., export and event queue) can be found in Bocchio et al. [2008]. Later in the text, some enhancements concerning the behavioral features are presented. In particular, we model dynamic processes and fork/join synchronization mechanism, capturing the semantics as specified in the IEEE 1666 SystemC standard and implemented in the SystemC 2.2 execution engine. Note that such enhancements are necessary to model the OSCI SystemC TLM 2.0 library, which is, however, out of scope of this article and has been planned as future work (see Section 9).

4.3.1 *Enhanced Behavioral Features.* We extended the SystemC process state machines by adding specialized submachine states and orthogonal regions within a state machine to model the notion of process hierarchy expressed in SystemC in terms of dynamic processes. A dynamic process is a process created at run-time during execution, as child process of a method process or a thread process, or a clocked thread process. A dynamic process can, in turn, create other processes dynamically. The SystemC 2.2 release supports the notion of dynamic process by introducing the concept of spawned process, that is, a process (a child process) created by another process (the parent process) by invoking the predefined function `sc_spawn`. In the SystemC UML profile, the dynamic creation of such a process—a dynamic spawned process—is denoted in the state machine diagram associated to the parent process by means of a

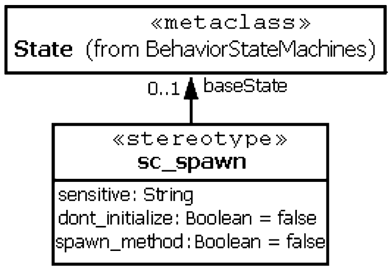

«metaclass»
State (from BehaviorStateMachines)
0..1
baseState
«stereotype»
sc_spawn
sensitive: String
dont_initialize: Boolean = false
spawn_method: Boolean = false

Fig. 10. sc_spawn stereotype.

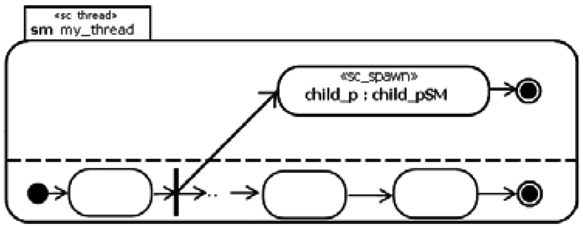


Fig. 11. Dynamic spawn of a process.

submachine state[8] labeled with the stereotype sc_spawn (see Figure 10 for the stereotype definition). The state machine referenced by the submachine state specifies the functionality of that dynamic process.

After the creation of a spawned process, the parent process and the new child process proceed in parallel, unless a specific synchronization schema is explicitly provided by the designer by means of notification of events. This natural asynchronism is reflected at UML level in the state machine diagram of the parent process by the use of orthogonal regions. To be precise, a process state machine, which dynamically creates processes is represented by a state machine with two or more regions (Figure 11). One region contains the behavior specification of the parent process, while the others contain exactly one sc_spawn submachine state each. The overall process creation (i.e., the invocation of the SystemC sc_spawn function) is denoted by a fork vertex in the parent region with two outgoing transitions: one entering in the sc_spawn submachine state of the child process and one entering in some state of the parent region to continue the specification of the parent process behavior after the process creation. Therefore, the submachine state (and, therefore, its reference process state machine) is exclusively entered via a fork vertex departing from the parent region and can be exited either as a result of reaching its final state (normal case) or via a join vertex in the parent region (in the case of a sc_fork/sc_join schema, see below). No entry/exit points can be defined for a sc_spawn submachine state.

The sc_spawn's tagged values are used to specify some spawn options, which determine certain properties of the spawned process instance. In particular, the

[8]In UML, a submachine state specifies the insertion of the specification of a submachine state machine. The state machine that contains the submachine state is the container.

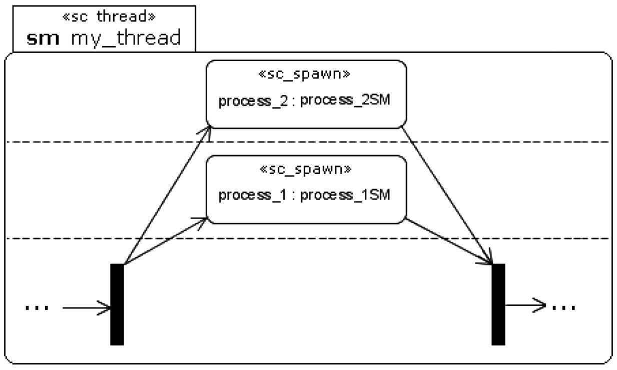


Fig. 12. SC_FORK and SC_JOIN.

boolean tag spawn_method being set to true indicates that the spawned process is a method process, and therefore, the associated process state machine shall be a method state machine. By default, this tag is set to false, that is, by default a spawned process is a thread process.

SystemC 2.2 introduces also the macros sc_fork and sc_join to be used in pairs within a thread process to enclose a set of calls to the function sc_spawn. The parent thread control leaves the fork-join construct when all the spawned processes are terminated; this means that during the execution of the spawned processes, the parent process is not running. We use the UML fork/join pseudostates to model these macros, as shown in Figure 12: a pair of fork/join for two spawned processes; both the fork/join bars are contained within the region of the parent (thread) process. After termination of the two spawned processes, the parent thread continues to execute.

5. THE UML PROFILE FOR THE SOFTWARE

We enhanced our embedded system design framework with the possibility to model pure application software adding a UML profile for the C language, based on the UML2 specification and on the Standard Posix Thread Library, as specified on the OpenGroup's Web site [Open Group 2008]. The profile extends the UML with the thread model and the structural/behavioral components supported by the Posix Thread library to represent resources and communication scenarios of a software multithreading environment for creating and destroying threads, passing messages and data between threads, scheduling thread execution, saving and restoring thread contexts, and so on.

The profile is defined at two main levels: the thread basic profile, a library-neutral modeling of all thread aspects that are independent from the specific library implementation, and the Posix thread profile, which specializes the basic profile by adding the Posix thread specificity. This separation facilitates a possible future extension of the profile to others thread libraries, such as OpenMP or MPI (Message Passing Interface) [OpenMP 2008]. The Posix thread profile has been defined to cover the host functional simulation. The C code for the application can be generated from the profiled UML model and simulated on

Table II. UML Diagrams in the Multithread C Profile

Diagram	Purpose
Class diagram	To define pt_modules, synchronization objects, and interfaces
Composite structure diagram	To describe how parts (modules and synchronization objects) are connected to each other
State machine diagram	Used as *method* state machine implementing the behavior of a thread or of a member function of modules and synchronization objects
Object diagran	Derived from a composite structure diagram to provide the specification (also partial) of values of structural features of entities

a host platform on a common OS (Linux) that provides the needed support for multitasking.

The thread basic profile includes the minimum set of stereotypes that describe the structural and behavioral part of the application model. It is structured as follows. The structure and communication part defines stereotypes for the basic and abstract thread model; that is, building blocks for describing threads, thread descriptors, and abstract synchronization objects. These elements are used to derive implementation specific thread models. These constructs (and the ones derived when necessary) are used in various UML structural diagrams, like class and composite structure diagrams. The behavior and synchronization part defines stereotypes that enhance the UML state machines to allow high-level representation of threads behavior including thread life management (create, join, and exit). A control structures part describes the standard C control structures (if-else, switch, loops) by means of special UML pseudostates.

Similarly, the Posix thread profile consists of: a structure and communication part, which introduces specific stereotypes supporting Posix thread descriptors, synchronization mechanisms (e.g., mutex, lockers, etc.), ports (like the mutexport stereotype, etc.); a behavior and synchronization part, which adds a few stereotypes to represent some locking characteristics.

In addition, a UML class library, called UML Posix Thread Library, has been defined to provide ready to use classes for structural elements (`pthread_t`, specific synchronization objects like `mutex`, `rwlock`, etc.) and primitive data types. The Posix Thread Profile and the UML Posix Thread Library depend on each other, and are both built on the top of the thread basic profile. Table II summarizes the UML diagrams used in the profile.

The complete profile definition can be found in Bocchio et al. [2007]. An overview of the basic profile follows. A modeling example is also provided later in the text to better illustrate the concepts.

5.1 The Thread Basic Profile

5.1.1 *Structure and Communication.* The thread execution model is represented by the stereotype `pt_module` (see Figure 13). A `pt_module` collects threads represented by the stereotypes `pt_thread_fun_t` and `pt_thread_descriptor`. These last describe the static behavior (operation whose behavior can be described by a state machine) and the actual state of the thread, respectively. The concurrency level in a `pt_module` is indicated by the tagged value `concurrency`.

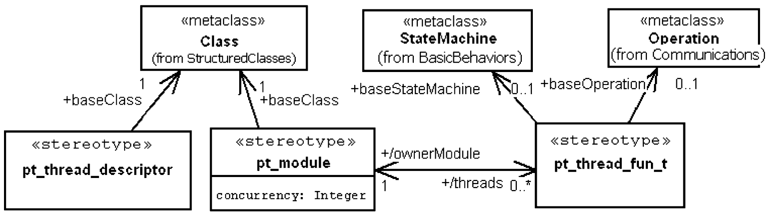


Fig. 13. Basic stereotypes of the thread basic profile.

A 0 value indicates an unlimited number of simultaneous active concurrent threads, while a positive value indicates an upper bound to the number of simultaneous active concurrent threads. A pt_module can be seen as a C scope. Moreover, a pt_module may represent the basic software unit mapped on a single computing unit of the hardware platform.

Synchronization objects are represented by the stereotype pt_sync_object that provides a common behavior to access synchronization objects; each specialized synchronization object (mutex, read-write locker, etc.) must implement this stereotype. The communication between threads and synchronization objects are modeled by ports: a pt_module will have ports that require services from defined interfaces, and these services will be provided by synchronization objects.

5.1.2 *Behavior and Synchronization.* The behavior of threads relating to management issues (like create, exit, and join primitives) and synchronization (locking and unlocking primitives) is described by state machine diagrams. Particular states that handle thread life cycles are stereotyped: pt_create, pt_join, and pt_exit state stereotypes are used in the profile to define, respectively, a thread creation, the synchronization between two threads, and the explicit termination of a thread. In particular, the pt_create state requires a pt_thread_descriptor object to act the creation of a new execution context, which is placed into the parent execution context. The created child thread is executed in the new context owned by the descriptor. The new context is represented by a thread descriptor and a new region of the state machine. After the new context is created, the threadable function identified by the submachine contained in the region is called.

Synchronization-related states are also stereotyped: lock, trylock, timedlock, and unlock are state stereotypes used to enforce and formalize the calls to the generic synchronization objects. However, these states do not formalize how the operating system manages the suspension and resuming of threads, or which messages or events cause the transitions out these states.

5.2 A Modeling Example: The Adaptive Unsharper Image Filter

The Adaptive Unsharper Image Filter is a special filter that enhances the contrast of an input color image, as described in Thomas et al. [2007]. This algorithm uses an adaptive filter that controls the contribution of the sharpening path in such a way that contrast enhancement occurs in high detail areas and little image sharpening occurs in smooth areas. The filter is essentially composed of the following functional units:

—read_im reads the images from the input stream;
—split_im divides the single frame in the three separate color components (Red, Green, and Blue);
—unsharp_im is a battery of three unsharper filters, one for each color component;
—union_im reassembles the three components into a single-color image frame;
—display_im adapts the internal image representation to the external displaying engine.

The chain of modules implements a multilevel pipeline: The synchronization between successive elaboration levels is important for functional correctness. For this reason, every module is connected to the next one by a synchronized channel based on an image buffer and a ready-acknowledge protocol. The producer stores the result in the output buffer, then sends a ready signal to communicate the availability of the image data, and waits for an acknowledge signal from the consumer module. On the other side, the consumer module waits for a ready signal from the producer, then processes the image, and, upon completion, sends an acknowledge signal to the producer. This simple buffer access protocol avoids that uncontrolled multiple accesses corrupt the data.

In the proposed system, the read_im module is a producer component that reads from a file the input stream; the split_im, unsharp_im, and union_im modules are both consumer and producer components that read the input stream from a preceding module(s) and produce an output stream for the next module(s); the display_im module is a consumer component that reads the stream from the preceding module and stores the image frames on the disk or in a display device (not represented here). All modules are stereotyped with pt_struct, a specialization of the stereotype pt_module, that allows creating C structs instead of files. For example, Figure 14 shows the UML class for the unsharp_im module and Listing 1 reports the corresponding C code.

The master-slave protocol between adjacent modules is managed by two semaphores (ready and acknowledge)[9] to grant the synchronism on the elaboration pipeline. In general, each module implements a set of support methods and attributes. The behavior of a module is represented by means of a threadable function (in this example named mainthread) stereotyped with the pt_thread_fun_t. Figure 15 shows a generic mainthread() implementation. It is essentially an infinite loop where the thread waits for a new input, locking one or more input_ready semaphores; waits that the consumer is ready to accept new information, locking one or more output_ack semaphores; performs the operation to transform the input information into the output information; signals to the consumer that a new information is ready, unlocking one or more

[9]It was decided to use semaphores because they allow a quite straightforward implementation of the required master-slave communication. Other choices, like mutex or read-write-lock, would require the support of variables that emulate semaphores.

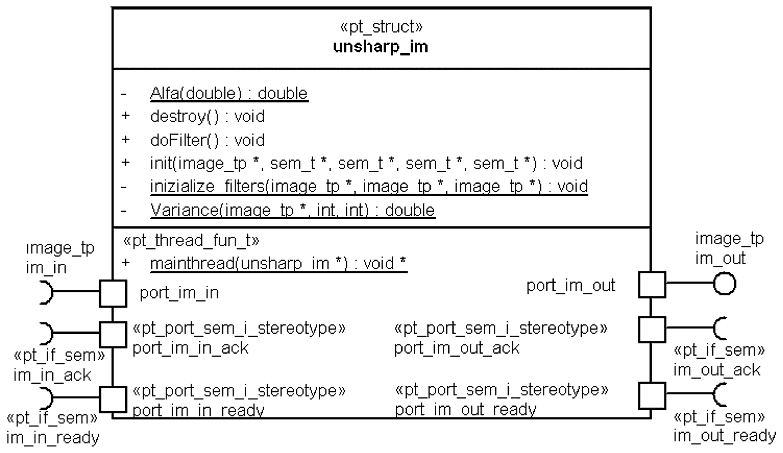

«pt_struct»
unsharp_im
- Alfa(double) : double
+ destroy() : void
+ doFilter() : void
+ init(image_tp *, sem_t *, sem_t *, sem_t *, sem_t *) : void
- inizialize_filters(image_tp *, image_tp *, image_tp *) : void
- Variance(image_tp *, int, int) : double
«pt_thread_fun_t»
+ mainthread(unsharp_im *) : void *
image_tp
im_in
port_im_in
«pt_port_sem_i_stereotype»
port_im_in_ack
«pt_if_sem»
im_in_ack
«pt_port_sem_i_stereotype»
port_im_in_ready
«pt_if_sem»
im_in_ready
image_tp
im_out
port_im_out
«pt_port_sem_i_stereotype»
port_im_out_ack
«pt_if_sem»
im_out_ack
«pt_port_sem_i_stereotype»
port_im_out_ready
«pt_if_sem»
im_out_ready

Fig. 14. The unsharp_im module.

```
#ifndef __UNSHARP_H__
#define __UNSHARP_H__
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "image_tp.h"
typedef struct _unsharp_im {
    image_tp *im_in;
    sem_t *im_in_ready;
    sem_t *im_in_ack;
    image_tp im_out;
    sem_t *im_out_ready;
    sem_t *im_out_ack;
} unsharp_im;
extern void unsharp_im_init(unsharp_im *us,
    image_tp *_im_in, sem_t *_im_in_ready,
    sem_t *_im_in_ack,image_tp *_im_out,
    sem_t *_im_out_ready, sem_t *_im_out_ack);
extern void unsharp_im_destroy(unsharp_im *us);
extern void unsharp_im_doFilter();
extern void *unsharp_im_mainthread(void *param);
#endif
```

Listing 1. struct_unsharp_im

output_ready semaphores; signals to the producer that the module is ready for new input, unlocking one or more input_ack semaphores.

The connections between consecutive objects can be implemented by unidirectional point-to-point channels. Each channel consists of a buffer (the image to be transferred) and two standard semaphores: the first represents a ready signal (the buffer contains valid data), the second represents the acknowledge signal (the buffer has been read, thus its contents can be over written). Figure 16 shows the two semaphores together with their corresponding C code. By naming convention, we identify by means of a _ack suffix the acknowledge signals, and by means of a _ready suffix the ready signal. Note that at start-up time every buffer is ready to receive data, thus all _ack semaphores are initialized to 1 (contents can be overwritten), while no buffers are ready to be read (no new content present), so all _ready semaphores are initialized to 0. The tagged value

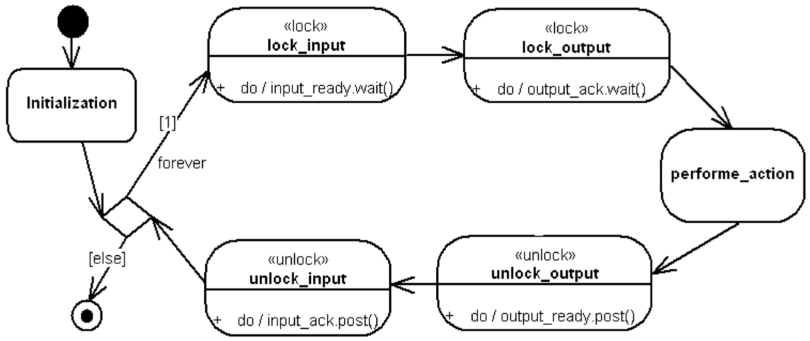
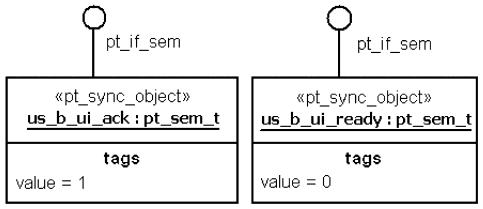


Fig. 15. The mainthread().



```
#include <semaphore.h>
...
sem_t si_us_g_ack;
sem_t si_us_g_ready;
...
sem_init(&si_us_g_ack, 0, 1);
sem_init(&si_us_g_ready, 0, 0);
...
```

Fig. 16. The acknowledge and ready semaphores and their C code.

`value` is used to denote the initialization values to pass to the constructors. The C code generator will use the initial values to create the correct initialization code.

The `adaptive_unsharper` module (see Figure 17) defines the `main()` function, that is, the entry point of the program. This function, after the creation of all objects, creates a thread for each module, then waits (indefinitely) for their conclusions, as shown in Figure 18; the created threads run in parallel. The `adaptive_unsharper` module is a composite class that defines the connection of the different modules, connected by synchronization objects. In Figure 19, a fragment of its composite structure is shown, representing the connection between the unsharper filter for the red channel and the union image. For the complete structure, see Figure 31 in Appendix B.

In conclusion, by means of the Posix thread profile, it was possible to model the Adaptive Unsharper Image Filter at a relatively high-level of abstraction, using the instruments provided by the library (for instance, the semaphores), without the burden of dealing with code-level details. The structure of the system was modeled in a very straightforward way (see the final resulting composite structure diagram reported in Appendix B in Figure 31). The behavior of the system was also described very easily: the state machine in Figure 15 gives a high-level intuitive definition of the behavior of a stage of the filter. This behavior is reused by all the filter stages. The inherent parallelism of the filter is also described in a rather straightforward way in Figure 18, again without the need to write any code.

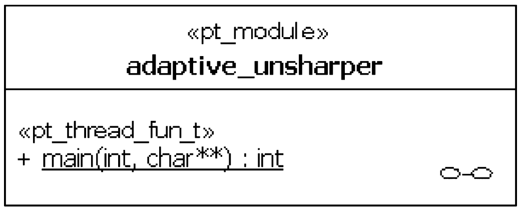


Fig. 17. The adaptive_unsharper module.

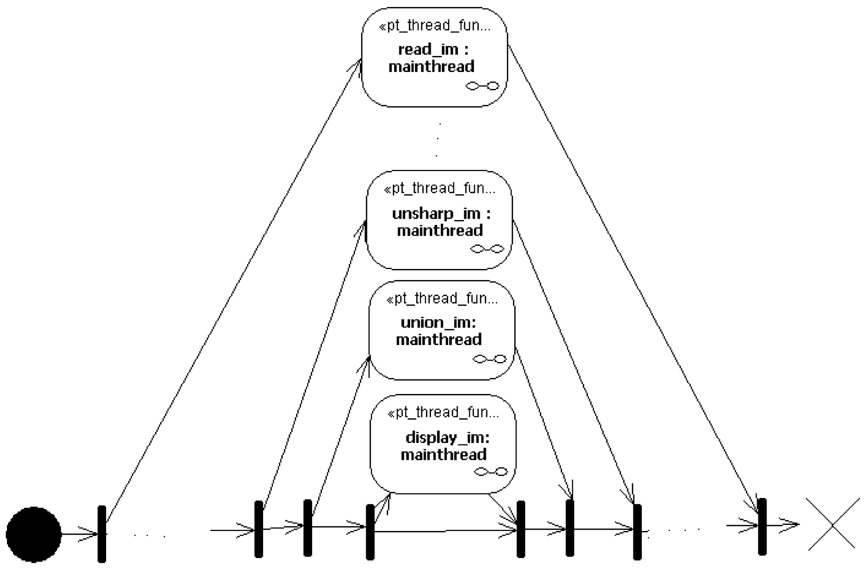


Fig. 18. The main() function.

6. MODEL REFINEMENT

Incremental abstraction/refinement patterns can be specified in terms of model-to-model transformations and handled by model transformation engines. These last can be integrated within the UML visual modeling tool to provide model transformation services. The idea is to collect and reuse precise abstraction/refinement transformation patterns coming from industry best practices. By applying the model transformations defined for a 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. The automation of such refinement patterns at UML level brings more control and flexibility on the model evolution process than operating directly at code level.

To clarify the concepts, we present some basic techniques adopted in particular in the communication refinement process. These refinement patterns can be applied to different communication scenarios (usually between two modules) of the design: hardware-hardware, software-software, hardware-software. For example, we illustrate the application at UML level of these communication refinement patterns to a hardware-hardware scenario.

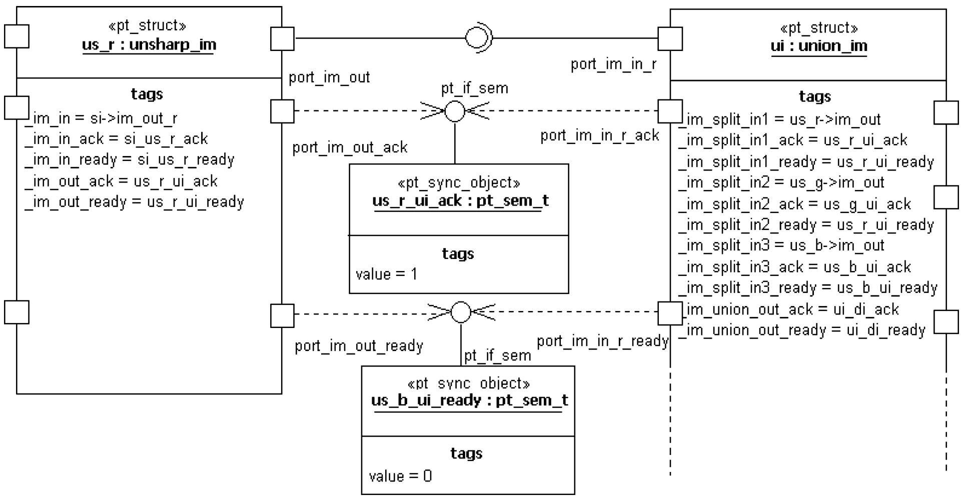

«pt_struct»
us_r : unsharp_im
tags
_im_in = si->im_out_r
_im_in_ack = si_us_r_ack
_im_in_ready = si_us_r_ready
_im_out_ack = us_r_ui_ack
_im_out_ready = us_r_ui_ready
port_im_out
port_im_out_ack
port_im_out_ready
pt_if_sem
«pt_sync_object»
us_r_ui_ack : pt_sem_t
tags
value = 1
pt_if_sem
«pt_sync_object»
us_b_ui_ready : pt_sem_t
tags
value = 0
port_im_in_r
port_im_in_r_ack
port_im_in_r_ready
«pt_struct»
ui : union_im
tags
_im_split_in1 = us_r->im_out
_im_split_in1_ack = us_r_ui_ack
_im_split_in1_ready = us_r_ui_ready
_im_split_in2 = us_g->im_out
_im_split_in2_ack = us_g_ui_ack
_im_split_in2_ready = us_r_ui_ready
_im_split_in3 = us_b->im_out
_im_split_in3_ack = us_b_ui_ack
_im_split_in3_ready = us_b_ui_ready
_im_union_out_ack = ui_di_ack
_im_union_out_ready = ui_di_ready

Fig. 19. A small fragment of the `adaptive_unsharper` structure.

6.1 Basic Patterns for Communication Refinement

Typically, the model of the system is a set of concurrently executing components (i.e., modules with inner processes) communicating with each other through abstract communication channels. The communication between modules occurs, therefore, through channels (possibly available as reusable components of a protocol library) that implement a point-to-point communication scheme. Access is given by interfaces, which can be accessed through ports defined in the corresponding module. The advantage of this modeling method is the separation between communication and computation [Jerraya and Wolf 2004]. That allows a separated refinement of communication and computation and thus an independent development. With this separation, we are able to refine the communication structure easily by removing the current channels and replacing them by more detailed channels with the designated communication schema implemented.

We assume to have two high-level modules M1 and M2 communicating over a channel C via some abstract protocol. The basic approach to refining this basic communication scenario consists of the following steps:

(1) Select an appropriate communication scheme to implement;
(2) Replace the abstract communication channel C with a refined one $C_{Refined}$, which realizes the selected communication protocol;
(3) Enable the communication of the modules M1 and M2 over $C_{Refined}$ by either:
(a) wrapping $C_{Refined}$ in a way that the resulting channel $C_{Wrapped}$ provides the interfaces required by M1 and M2 (***wrapping***), or
(b) refining M1 and M2 into $M1_{Refined}$ and $M2_{Refined}$, respectively, so that their required interfaces match the ones provided by $C_{Refined}$ (***merging***).

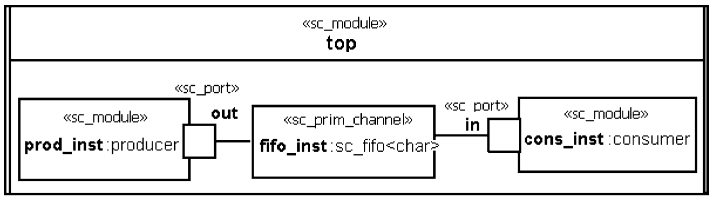


Fig. 20. Producer/consumer modules communicating via a primitive FIFO.

The two techniques are similar. Both include an intermediate step to build two further modules (i.e., two SystemC hierarchical channels), called adapters,[10] to map one interface to another: one, say A1, between M1 and $C_{Refined}$, and one, say A2, between $C_{Refined}$ and M2. In the case (a), the resulting channel $C_{Wrapped}$ encloses $C_{Refined}$ and the two adapters; while, in the case (b) these adapters are merged to the calling modules M1 and M2 resulting in the refined modules $M1_{Refined}$ and $M2_{Refined}$. Deciding whether to use wrapping or merging depends on the methodology and on the chosen target architecture. The wrapping technique is helpful when targeting software implementations, while the merging approach is typically required when refining an abstract communication protocol (denoted by an abstract channel) toward a hardware implementation based on a given target architecture (e.g., the refinement of a transaction-level model to a pin-level model).

6.2 Example of Hardware-Hardware Communication Refinement

Here, we present a hardware-hardware communication refinement scenario of a simple producer/consumer system taken from Gröetker et al. [2002]. We look at the basic tasks involved in the refinement process moving from a functional model to a RTL model. The modeling notation adopted to this purpose is the SystemC UML profile.

The UML composite structure diagram in Figure 20 shows the overall design of the producer/consumer system. A top composite module contains two functional modules, a producer prod_inst and a consumer cons_inst, communicating via an abstract FIFO channel fifo_inst by writing and reading characters to/from it. The FIFO channel is an instance of the primitive channel sc_fifo, which permits to store characters by means of blocking read and write interfaces. Two processes, the producer and the consumer (see the thread processes main within the producer and the consumer modules in Figure 21), 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, 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 Figure 21). 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.

[10]Adapters are components that can be easily reused. Typically, a finite number of different interfaces are used, and, therefore, a small library of adapters is usually sufficient to cope with the needs of even large designs.

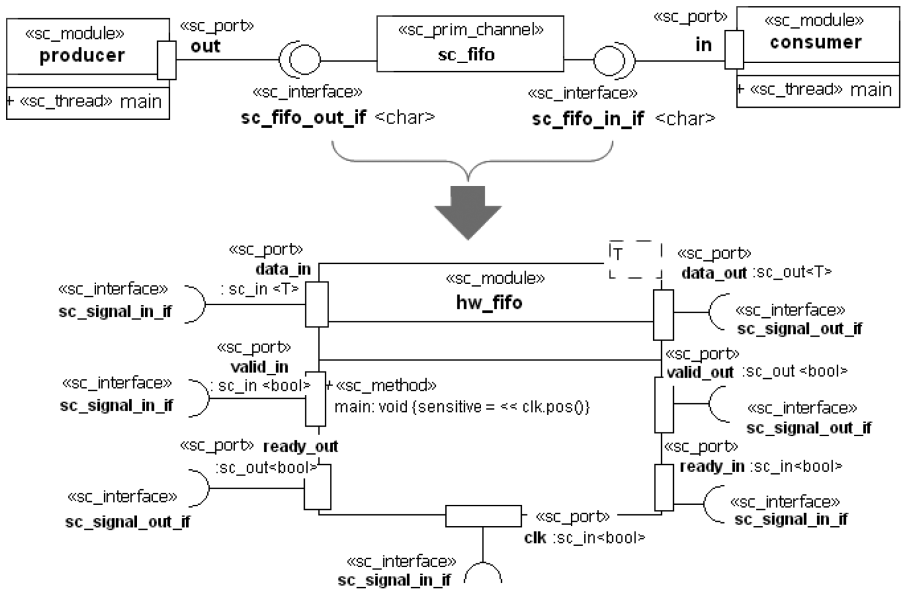


Fig. 21. Replacing a primitive FIFO with a clocked hardware FIFO.

Now, we aim at replacing the (abstract) FIFO instance above with a (refined) model of a clocked RTL hardware FIFO named hw_fifo for a hardware implementation (see Figure 21). The new hardware FIFO uses a signal-level ready/valid handshake protocol for both the FIFO input and output. Data is read/written at the rising clock edge if both the valid and ready lines are active.

It should be noted that we cannot use an hw_fifo instance directly in place of the sc_fifo instance, since the former does not provide any interfaces at all, but has ports that connect to signals, that is, has ports that use the sc_signal_in_if and sc_signal_out_if interfaces. In order to connect prod_inst and cons_inst to an hw_fifo instance, we need a pair of adapters that convert the FIFO's input and output interfaces into pin-level accesses.

As explained earlier, the adapters can be integrated in the design by wrapping or by merging. We illustrate in the sequel both the two communication refinement techniques. The by-wrapping approach can easily be automatized by model transformations, while the operations of the by-merging technique require a certain brain effort to be performed. Currently, we have implemented in the EA-based environment only the by-wrapping refinement pattern by adopting an hybrid approach based on both direct-manipulation and structure-driven approach [Czarnecki and Helsen 2003].

6.2.1 *Adapter-Wrapping.* Following the wrapper-based approach 3.b of the refinement procedure described in Section 6.1, we can define a hierarchical channel hw_fifo_wrapper ($C_{Wrapped}$), which implements the sc_fifo_out_if and sc_fifo_in_if interfaces and contains an instance of hw_fifo ($C_{Refined}$). In addition, it contains sc_signal instances to interface with hw_fifo and a clock port (since hw_fifo has also a clock port) to feed in the clock signal to the hw_fifo instance

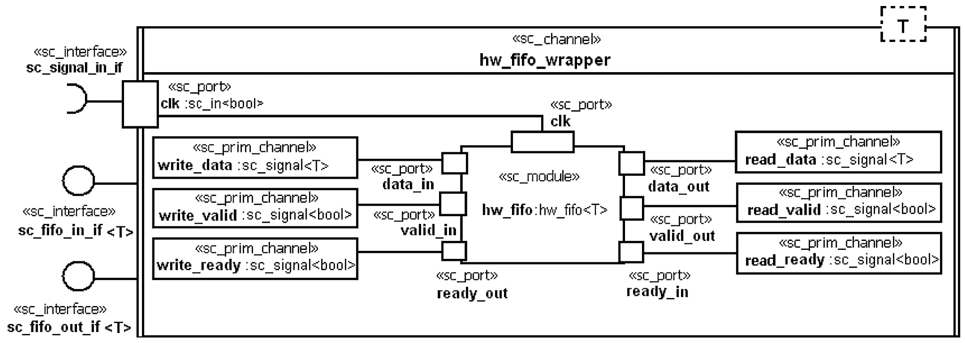


Fig. 22. The hw_fifo_wrapper hierarchical channel.

(Figure 22). 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 instance (Figure 23). Clearly, we had to replace the template argument (T) of the adapter with the actual data-type being used (char).

6.2.2 *Adapter-Merging.* Following the merging-based approach 3.a of the refinement procedure presented in Section 6.1, we need a pair of adapters, fifo_write_hs and fifo_read_hs, to be merged into the calling modules (the producer and the consumer, respectively). These adapters are hierarchical channels implementing the required interfaces. The fifo_write_hs adapter implements the sc_fifo_out_if interface and its implementation of the write operation closely corresponds to that of the hw_fifo_wrapper. The fifo_write_hs adapter has four ports; one for the data, one for the clock, and two for the control lines valid and ready. When the write operation is invoked by the producer instance, the adapter drives the data lines with the new sample and asserts the valid line; it then waits until the consumer has read the sample. This happens on a rising clock edge if both the valid and ready lines are active. The fifo_read_hs adapter can be implemented similarly. Before merging, the adapters instances can be inserted, as they are in the system for a preliminary validation.

In the process of merging an adapter into the calling module, the adapter's ports, attributes, operations and processes are copied into the original calling module. This last is further refined by removing the original ports used to access the adapter. The result is a refined pin-level module that replaces the original module and adapter instances in the system. For the example under consideration, the producer module is merged with the fifo_write_hs and the result is a new module hw_producer. Similarly, by merging the fifo_read_hs adapter into the consumer modules, we obtain the refined module hw_consumer. Clearly, in both the two refined modules the template argument (T) of the adapters is replaced with the actual data-type being used (char). Figure 24 shows the top-level composite structure of this scenario.

7. THE CODESIGN ENVIRONMENT

We developed a prototype tool working as front-end for consolidated lower level design tools. Figure 25 shows the tool architecture (components inside

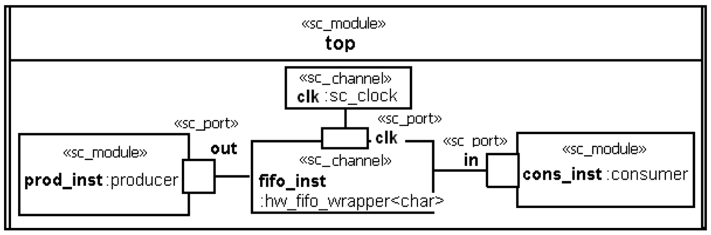


Fig. 23. A clocked producer/consumer design (by adapter-wrapping).

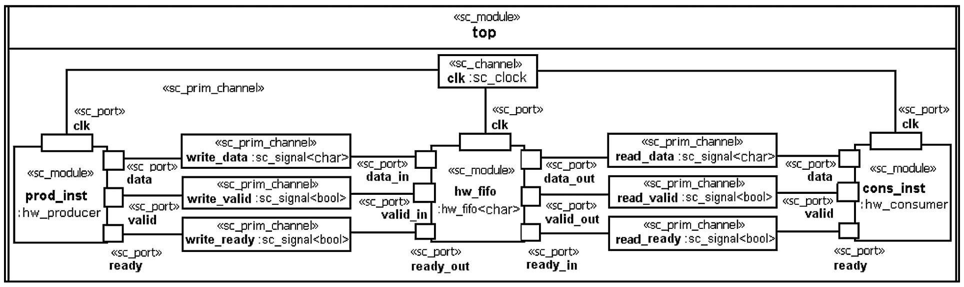


Fig. 24. A clocked producer/consumer design (by adapter-merging).

dashed lines are still under development). The tool consists of two major parts: a development kit (DK) with design and development components, and a run-time environment (RE) represented by the SystemC execution engine. The DK consists of:

—A modeler supporting the UML2 profiles for SystemC and for multithread C. It is based on the Enterprise Architect tool [Enterprise Architect 2008] by SparxSystems.
—Two translators for the forward/reverse engineering to/from C/C++/ SystemC. For code generators, we followed a full generation approach, so C/C++/SystemC are adopted as action languages at PSM level and the source code is fully generated. The reverse engineering component translates SystemC code into a UML model conforming to the SystemC UML profile, C++ code into UML classes, and C code into a UML model conforming to the multithread C UML profile. Currently, the reverse facility is limited to the generation of the design skeleton, that is, the static structure of the system.
—An abstraction/refinement evaluator (under development) to guarantee traceability and correctness along the refinement process from the high-level abstract description to the final implementation. This component supports some model-to-model transformations defined for some predefined refinement patterns and implemented by using the EA MDA transformers.
—A validation & verification toolset (under development), which is built upon the abstract state machine metamodeling (ASMETA) toolset [ASMETA 2008], to be used to guarantee traceability and correctness along the UPSoC design process from a UML high-level abstract description of the system to the final SystemC implementation.

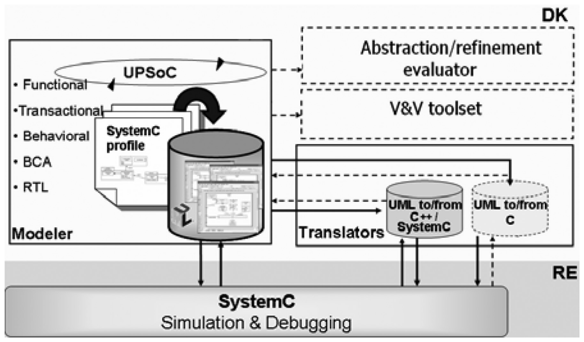


Fig. 25. Tool architecture.

Further details of the developed components of the codesign environment are given in Riccobene et al. [2006b].

8. INDUSTRIAL CASE STUDIES

For the hardware part, we have developed several different case studies, some taken from the SystemC distribution like the simple bus design, and some of industrial interest. The simple bus design is a well-known transactional level example to perform also cycle-accurate simulation. It is made of 1,018 lines of code that implement a high performance, abstract bus model. The complete code is available at OSCI group [2008]. We modeled the simple bus system entirely in a forward engineering manner. The code generator has been tested primarily on this example, and it produced an executable and complete (i.e., including both structural and behavioral parts) SystemC code. In this example, we performed also an evaluation about the benefits of modeling with our UML-based methodology compared to coding. Even if our methodology is oriented to the implementation and the modeling style strictly resembles the structure of the target SystemC language, by analyzing the produced SystemC code, we found out that 652 of the 1,018 lines are automatically inferred from the description of the model, the remaining 366 lines of code are produced from the actions inside the model (that are introduced manually) or are derived from the description in the process state machines diagrams. All in all, we unexpectedly concluded that there is a significant reduction in the effort to produce the final code and there is also a benefit in documentation and reuse. For instance, one benefit compared with SystemC is that all the header files are completely and consistently generated from the UML structural diagrams.

To test the expressive power of the SystemC UML profile in representing a variety of architectural and behavioral aspects, we modeled the On-Chip Communication Network (OCCN) library [OCCN 2005]. The OCCN project focuses on modeling complex on-chip communication networks by providing a highly-parameterized and configurable SystemC library. This library is made of about 14.000 lines of code and implements an abstract communication pattern for connecting multiple processing elements and storage elements on a single chip.

The generic features of the OCCN modeling approach involve multiple abstraction levels, separation of communication and computation, and communication layering. We used this example to test the reverse engineering flow. The OCCN design skeleton has been imported automatically from the C++/SystemC code into the EA-based modeler exploiting the reverse engineering facility, then it was refined in the behavioral parts using the modeling constructs of the SystemC UML profile. In this case study, we tested advantages and indispensability of a round trip engineering—based on a synchronized cooperation between code generation (forward engineering) and reverse engineering—in order to have a complete description of a system design on both the UML abstract level and the code level. Through this example, we also understood that the reverse engineering component is useful in practice as a tool to inspect the structure of a source code graphically. Starting from a bunch of source code header files, it is possible to obtain immediately a graphical representation of the significative design changes introduced at code level including structure, relations, variables, functions, and so on.

In Bocchio et al. [2005], we present an application example related to an 802.11.a physical layer transmitter and receiver system described at instruction level. The hardware platform is composed of a VLIW processor developed in STM, called LX, with a dedicated hardware coprocessor that implements a fast fourier transform (FFT) operation. The processor acts as a master to the hardware module and the memory components, where code and data are stored. The communication is implemented by a system bus: We use the OCCN SystemC model described earlier. The UML model of the application software is a function library encapsulated in a UML class, which provides through ports the I/O interface of the software layer to the hardware system. This class is then translated to C/C++ code and the resulting application code is executed by the LX ISS wrapped in SystemC to allow cycle accurate hardware-software cosimulation. The UML wrapper of the LX ISS is modeled with the SystemC UML profile, in order to generate a SystemC wrapper for the ISS and to allow a hardware-software cosimulation at transactional or cycle-accurate level.

To give a feeling about the effective usage of the proposed design methodology for modeling real cases, we recall that the effort for modeling the simple bus was about one person per week, while the effort for modeling the OCCN and the 802.11.a was about one person per month for each example. Of course, we started the description of these case studies having their SystemC or C description available, so we cannot really perform an evaluation of the design effort of “starting a design in SystemC from scratch” versus “starting the same design with this methodology from scratch”. Our feeling is that the time and effort of the two approaches are comparable. There are, indeed, a few benefits using this UML-based approach: There is a big savings in the number of lines that are written by hand (redundant lines of codes due to the C++ syntax are avoided, see the numbers from the simple bus example), but the greatest benefits consist in the model maintenance and its documentation.

All case studies mentioned above have not required any special refinement strategy. Currently, we are defining a formal refinement methodology with precise abstraction/refinement rules for the transactional-level modeling. To achieve this goal, we started by making a revision of the SystemC UML profile to include the new features provided by the SystemC IEEE Standard 1,666, which are the basis for modeling the the OSCI TLM 2.0 library [OSCI group 2008] and supporting a certain number of TLM sublevels. Moreover, a complete hardware-software codesign case study that illustrates the use of both profiles in a joined coherent design flow is under development. In this article, we presented examples relating to the two separate domains.

9. CONCLUSIONS AND FUTURE DIRECTIONS

In this article, we show how UML and UML profiles can be effectively used within a wider scope of application domains, such as the SoC and the embedded system design. The work presented here is part of our ongoing effort to enact the UPES/UPSoC design flow that starts with system descriptions using UML-notations and automatically produce complete implementations of the software and hardware components as well as their communication interfaces.

We extended the UML2 for SystemC and for multithread C in order to provide a means for software and hardware engineers to improve the current industrial embedded systems design flow joining the capabilities of UML, SystemC, and C to operate at system level. Whereas the UML profile for SystemC allows modeling resources and concurrency from the hardware perspective, the UML profile for multithread C (inspired by the POSIX-pthread library) permits to model the concurrency and the access to the resources as they are seen from the software perspective.

Currently, we have been working on implementing a (bidirectional) transformation bridge between the software and the hardware perspectives in order to facilitate the mapping process. It would be desirable to have a single SystemC-based environment for both architecture exploration and the application software implementation, eliminating the need to create two separate validation contexts. However, more planning and analysis must be conducted. The modeling of complex concurrency aspects over threads, and the model composition of SystemC threads, and POSIX C threads through automated interfaces is still a challenging issue.

In the future, we aim also at identifying characteristics of reusable model transformations and ways of achieving reuse by collecting in a library precise abstraction/refinement transformation patterns according to the levels of abstraction: functional, TLM, behavioral, BCA, and RTL. In particular, we are focusing on the TLM level to model the communication aspects at a certain number of TLM sublevels according to the OSCI TLM standard [OSCI group 2008]. We believe the use of fine-grained transformations that are being composed (chaining) would be beneficial, both increasing the productivity and the quality of the developed systems.

Recently [Gargantini et al. 2008b; Carioni et al. 2008], we are also tackling the problem of formally analyzing UML visual models. We aim at complementing the proposed design methodology with a formal analysis process for high-level system validation and verification (V&V), which involves the abstract state machine (ASM) [Börger and Stärk 2003] formal method. Formal methods and analysis tools have been most often applied to low-level hardware design. However, these techniques are not applicable to embedded system descriptions given in system-level design languages (like SystemC, SpecC, etc.) [Vardi 2007], since such languages are closer to concurrent software than to traditional hardware description. We propose to address the problem of formally analyze high-level UML-like embedded system descriptions by joining UML-like modeling languages with the ASM formal method and its related techniques for formal model analysis. Our overall goal is to provide a design environment where both the application software and the hardware architecture are described together by a multiviews UML model representing the mapping of the functionality (of the application software) onto an architecture. Moreover, through a formal analysis process the system components can be functionally validated and verified early at high-levels of abstraction, and even in a transparent way (i.e., no strong skills and expertise on formal methods are required to the user) by the use of the ASM formal method and supporting analysis tools.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

Arlow, J. and Neustadt, I. 2002. *UML and the Unified Process.* Addison Wesley, Reading, MA.

ASMETA. 2009. The Abstract State Machine mETAmodeling. Web site. http://asmeta.sf.net/.

Bézivin, J. 2005. On the unification power of models. *Softw. Syst. Model. 4*, 2, 171–188.

Bocchio, S., Lavazza, L., Mantellini, L., and Rosti, A. 2007. A UML profile for Posix thread library. STMicroelctronics Tech. rep., AST-AGR-2007-7.

Bocchio, S., Riccobene, E., Rosti, A., and Scandurra, P. 2005. A SoC design flow-based on UML 2.0 and SystemC. In *Proceedings of the International Design Automation Conference.* ACM, New York.

Bocchio, S., Riccobene, E., Rosti, A., and Scandurra, P. 2008. An enhanced SystemC UML profile for modeling at transaction-level. In *Embedded Systems Specification and Design Languages.* Springer, Berlin.

Börger, E. and Stärk, R. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, Berlin.

Bruschi, F. and Sciuto, D. 2002. SystemC based design flow starting from UML model. In *Proceedings of the European SystemC Users Group Meeting.*

Carioni, A., Gargantini, A., Riccobene, E., and Scandurra, P. 2008. Scenario-based validation of embedded systems. In *Proceedings of the Forum on Specification and Design Languages.*

Carioni, A., Gargantini, A. Riccobene, E., and Scandurra, P. 2009. Model-driven system validation by scenarios. In *Languages for Embedded Systems and Their Applications.* Springer-Verlag, Berlin.

Chen, J. and Cui, H. 2004. Translation from adapted UML to Promela for CORBA-based applications. In *Proceedings of the 11th SPIN Workshop.* Springer-Verlag, Berlin, 234–251.

Czarnecki, K. and Helsen, S. 2003. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Achitecture.* ACM, New York.

ECSI UML Workshop. 2006. UML profiles for embedded systems, http://www.ecsi-association.org/ecsi.

EDWARDS, M. AND GREEN, P. 2003. UML for hardware and software object modeling. In *UML for Real: Design of Embedded Real-Time Systems.* Kluwer Academic, Dordrecht, The Netherlands.

ENTERPRISE ARCHITECT. 2008. The Enterprise Architect Tool. http://www.sparxsystems.com.au/.

GARGANTINI, A., RICCOBENE, E., AND SCANDURRA, P. 2008a. A language and a simulation engine for abstract state machines based on meta-modeling. *J. Universal Comput. Sci. 14,* 12, 1949–1983.

GARGANTINI, A., RICCOBENE, E., AND SCANDURRA, P. 2008b. A model-driven validation & verification environment for embedded systems. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES'08).* IEEE, Los Alamitos, CA.

GARGANTINI, A., RICCOBENE, E., AND SCANDURRA, P. 2009. Model-driven design and ASM-based validation of embedded systems. In *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation.* Springer, Dordrecht, The Netherlands.

GRÖETKER, T., LIAO, S., MARTIN, G., AND SWAN, S. 2002. *System Design with SystemC.* Kluwer Academic, Amsterdam.

IP-XACT. 2007. SPIRIT Consortium, IP-XACT schema v1.4. http://www.spiritconsortium.org.

JERRAYA, A. A. AND WOLF, W. 2004. *Multi-Processor Systems-on-Chips.* Elsevier, San Francisco, CA.

KEUTZER, K., NEWTON, A. R., RABAEY, J. M., AND VINCENTELLI, A. S. 2000. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. CAD Integr. Circu. Syst. 19*, 12, 1523–1543.

KREKU, J., HOPPARI, M., AND TIENSYRJA, K. 2007. SystemC workload model generation from UML for performance simulation. In *Proceedings of the Forum on Specification and Design Languages.*

KRUCHTEN, P. 1999. *The Rational Unified Process.* Addison Wesley, Reading, MA.

LAVANGNO, L., MARTIN, G., VINCENTELLI, A. S., RABAEY, J., CHEN, R., AND SGROI, M. 2003. UML and platform-based design. In *UML for Real Design of Embedded Real-Time Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

LEROUX, H., MINGINS, C., AND REQUILE-ROMANCZUK, A. 2003. JACOT: A UML-based tool for the run-time-inspecton of concurrent Java programs. In *Proceedings of the 1st Workshop on Advancing the State-of-the-Art in Run-Time Inspection.* Elsvier, Amsterdam.

MARTE. 2008. OMG, UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE), ptc/08-06-08.

MARTIN, G. 1999. UML and VCC. White paper. Candence Design Systems. Inc.

MARTIN, G. AND MUELLER, W. 2005. *UML for SoC Design.* Springer, Berlin, Germany.

MDA. 2003. OMG, the Model Driven Architecture. Guide V1.0.1. http://www.omg.org/mda/.

MOORE, T., VANDERPERREN, Y., SONCK, G., VAN OOSTENDE, P., PAUWELS, M., AND DEHAENE, W. 2002. A design methodology for the development of a complex system-on-chip using UML and executable system models. In *Proceedings of the Forum on Specification and Design Languages.*

MURA, M. PAOLOIERI, M., NEGRI, L., AND SAMI, M. 2007. StateCharts to SystemC: A high-level hardware simulation approach. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI.* ACM, New York, 505–508.

NGUYEN, K. D., SUN, Z., THIAGARAJAN, P. S., AND WONG, W. F. 2005. Model-driven SoC design: The UML-SystemC bridge. In *UML for SoC Design*, Martin, G. and Mueller, W. Springer, Berlin, Germany.

OCCN. 2005. OCCN Project. http://occn.sourceforge.net/.

OPEN GROUP. 2008. The Open Group Consortium. http://www.opengroup.org.

OPENMP. 2008. OpenMP Application Program Interface. http://www.openmp.org.

OSCI GROUP. 2008. The Open SystemC Initiative. http://www.systemc.org.

RASLAM, W. AND SAMEH, A. 2007. Mapping SysML to SystemC. In *Proceedings of Forum on Specification and Design Languages.*

RICCOBENE, E. AND SCANDURRA, P. 2004. Modelling SystemC process behavior by the UML method state machines. In *Proceedings of the 1st International Workshop on Rapid Integration of Software Engineering Techniques.* Springer, Berlin.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2005a. A SoC design methodology based on a UML 2.0 profile for SystemC. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE'05).* IEEE, Los Alamitos.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2005b. A UML 2.0 profile for SystemC: Toward high-level SoC design. STMicroelectronics Tech. rep., AST-AGR-2005-3.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2005c. A UML 2.0 profile for SystemC: Toward high-level SoC design. In *Proceedings of the 5th ACM International Conference on Embedded Software.* ACM, New York, 138–141.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2006a. A model-driven co-design flow for embedded systems. In *Proceedings of the Forum on Specification and Design Languages.*

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2006b. A model-driven design environment for embedded systems. In *Proceedings of the 43rd Annual Conference on Design Automation.* ACM, New York, 915–918.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2007a. A model-driven co-design flow for embedded systems. In *Advances in Design and Specification Languages for Embedded Systems.* Springer, Berlin, Germany.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2007b. Designing a unified process for embedded systems. In *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software.* IEEE, Los Alamitos, CA.

SELIC, B. 2000. A generic framework for modeling resources with UML. *Computer 33,* 6, 64–69.

SPT. 2003. OMG, UML Profile for Schedulability, Performance, and Time, formal/03-09-01.

SYSML. 2007. OMG, SysML, formal/2007-09-01. http://www.omgsysml.org/.

SystemC. 2006. SystemC Language Reference Manual. IEEE Std 1666.

THOMAS, F., GERARD, S., DELATOUR, J. AND TERRIER, F. 2007. Software real-time resource modeling. In *Proceedings of the Forum on Specification and Design Languages.*

UML. 2008. OMG, the Unified Modeling Language (UML). http://www.uml.org.

UML PROFILE FOR SWRADIO. 2007. OMG, UML Profile for SWRadio. V1.0, formal/07-03-01.

UML-SOC WORKSHOPS. 2008. UML for SoC Design Workshops. http://www.c-lab.de/uml-soc.

USOC. 2006. OMG, UML Profile for SoC Specification, v1.0.1.

VARDI, M. Y. 2007. Formal Techniques for SystemC Verification; Position Paper. In *Proceedings of the 44th annual conference on Design automation*. IEEE, Los Alamitos, CA.

VINCENTELLI, A. S. 2002. Defining platform-based design. *EEDesign of EETimes.*

ZHU, Q., OISHI, R., HASEGAWA, T., AND NAKATA, T. 2004. System-on-chip validation using UML and CWL. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, Los Alamitos, CA, 92–97.

ZIMMERMAN, J. BRINGMANN, O., GERLACH, J., SCHAEFER, F., AND NAGELDINGER, U. 2008. Holistic sytem modeling and refinement of intern-connected micro-electronic systems. In *Proceedings of the Conference on Design, Automation, and Test in Europe.* IEEE, Los Alamitos, CA.

Received April 2008; revised April 2008; accepted September 2008