

Chapter 2

Extending UML for Electronic Systems Design: A Code Generation Perspective

Yves Vanderperren, Wolfgang Mueller, Da He, Fabian Mischkalla,
and Wim Dehaene

1 Introduction

Larger scale designs, increased mask and design costs, ‘first time right’ requirements and shorter product development cycles motivate the application of innovative ‘System on a Chip’ (SoC) methodologies which tackle complex system design issues.¹ There is a noticeable need for design flows towards implementation starting from higher level modeling. The application of the Unified Modeling Language (UML) in the context of electronic systems has attracted growing interest in the recent years [16, 35], and several experiences from industrial and academic users have been reported [34, 58].

Following its introduction in 1995, UML has been widely accepted in software engineering and supported by a considerable number of Computer Aided Software Engineering (CASE) tools. Although UML has its roots in the software domain, the Object Management Group (OMG), the organization driving the UML standardization effort [44, 45], has turned the UML notation into a general-purpose modeling language which can be used for various application domains, ranging from business process to engineering modeling, mainly for documentation purposes. Besides the language complexity, the main drawback of such a broad target is the lack of

¹While the term ‘SoC’ is commonly understood as the packaging of all the necessary electronic circuits and parts for a system on a *single* chip, we consider the term in larger sense here, and cover electronic systems irrespective of the underlying implementation technology. These systems, which might be multi-chip, involve several disciplines including specification, architecture exploration, analog and digital hardware design, the development of embedded software which may be running on top of a real-time operating system (RTOS), verification, etc.

Y. Vanderperren (✉) · W. Dehaene
ESAT-MICAS, Katholieke Universiteit Leuven, Leuven, Belgium
e-mail: yves.vanderperren@ieee.org

W. Mueller · D. He · F. Mischkalla
C-LAB, Paderborn University, Paderborn, Germany

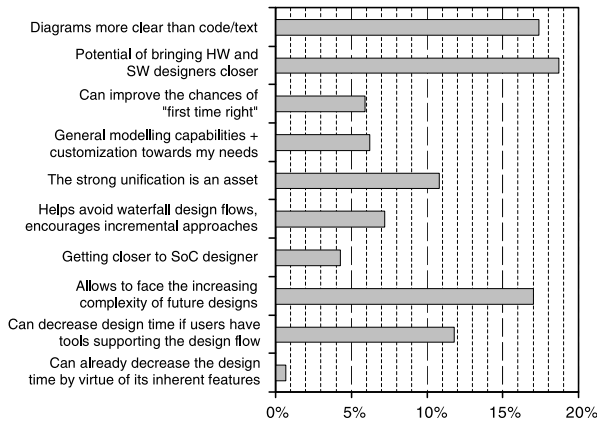


Fig. 2.1 Positive aspects of UML [62]

sufficient semantics, which constitutes the main obstacle for real engineering application. Therefore, application specific customizations of UML (UML profiles), such as the System Modeling Language (SysML) [38] and the UML Profile for SoC [42], are of increasing importance. The addition of precise semantics allows for the automatic generation of code skeleton, typically C++ or Java, from UML models.

In the domain of embedded systems, the complexity of embedded software doubled every 10 months in the last decades. Automotive software, for instance, may exceed several GBytes [22]. In this domain, complexity is now successfully managed by model-based development and testing methodologies based on MATLAB/Simulink with highly efficient C code generation. Unfortunately, the situation is more complex in electronic systems design than in the embedded software domain, as designers face a combination of various disciplines, the coexistence of multiple design languages, and several abstraction levels. Furthermore, multiprocessor architectures have become commonplace and require languages and tool support for parallel programming. In this multi-disciplinary context,

UML has great potential to unify hardware and software design flows. The possibility to bring designers of both domains closer and to improve the communication between them was recognized as a major advantage, as reported by surveys conducted during the UML-SoC Workshops at the Design Automation Conference (DAC) in 2006 (Fig. 2.1) and 2007 [63]. Additionally, UML is also perceived as a means to manage the increasing complexity of future designs and improve their specification. UML diagrams are expected to provide a clearer overview compared to text.

Significant issues remain, however, such as the perceived lack of maturity of tool support, the possible difficulty of acceptance by designers due to lack of knowledge, and the existence of different UML extensions applicable to SoC design but which are not necessarily compatible [62].

A detailed presentation of the UML is beyond the scope of this chapter and we assume that the reader has a basic knowledge of the language. The focus of this

chapter is the concrete application of UML to SoC design, and follows the following structure. The next section introduces basic concepts of the UML extension mechanism, how to define a UML profile. Thereafter, we present some UML profiles relevant for UML for SoC and embedded systems design before we introduce one application in the context of SystemC/C++ co-simulation and -synthesis [27].

2 Extending UML

A stereotype is an extensibility mechanism of UML which allows users to define modeling elements derived from existing UML classifiers, such as classes and associations, and to customize these towards individual application domains [45]. Graphically, a stereotype is rendered as a name enclosed by `«...»`. The readability and interpretation of models can be highly improved by using a limited number of well defined stereotypes. Additionally, stereotypes can add precise meanings to individual elements, enabling automatic code generation.

For instance, stereotypes corresponding to SystemC constructs can be defined, such as `«sc_module»`, `«sc_clock»`, `«sc_thread»`, `«sc_method»` etc. The individual elements of a UML model can then be annotated with these stereotypes to indicate which SystemC construct they correspond to. The resulting UML model constitutes a first specification of a SystemC model, which can then be automatically generated. The stereotypes give to the UML elements the precise semantics from the target language (SystemC in this case).

As an example, Fig. 2.2 represents a Class Diagram with SystemC-oriented stereotypes. It corresponds to the simple bus example delivered with SystemC, with master, slave, and arbiter classes stereotyped as `«sc_module»` and connected to a bus. Modules are connected by a directed association with stereotype `«connect»`. We introduce this stereotype as an abstraction for a port with associated interface where the flow points into the direction of the interface. An alternative and more detailed representation of the bus connection is provided by the explicit definition of the interface via a separate element with stereotype `«sc_interface»` (Fig. 2.3). Such examples illustrate how stereotypes add necessary interpretations to UML diagrams. A clear definition and structure of stereotypes is of utmost importance before applying UML for effective documentation and efficient code generation.

UML is defined on the basis of a metamodel, i.e., the UML language is itself described based on a model. This approach makes the language extremely flexible, since an application specific customization can be easily defined by the extension of that metamodel through the definition of stereotypes. In theory, the principle of an application specific customization of UML through a so-called *UML profile* through stereotypes is simple. Considering a specific application domain, all unnecessary parts of the UML metamodel are stripped in a first step. In a second step, the resulting metamodel is extended. This mainly means the definition of a set of additional stereotypes and tagged values, i.e., stereotype attributes. In further steps, useful graphical icons/symbols, constraints, and semantic outlines are added.

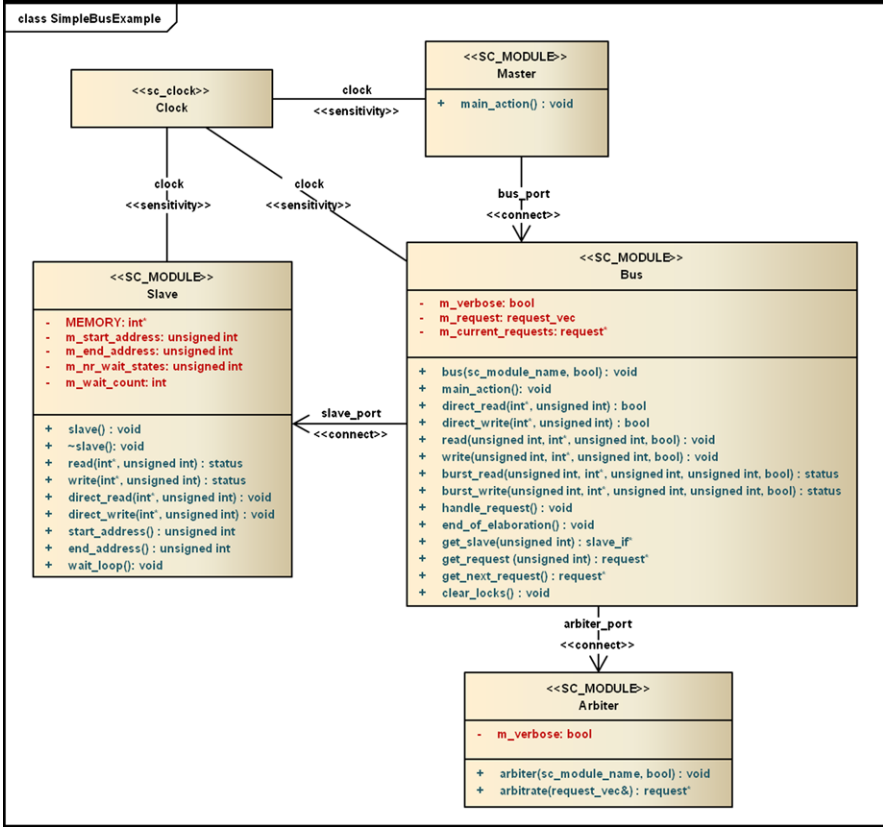


Fig. 2.2 UML simple bus class diagram

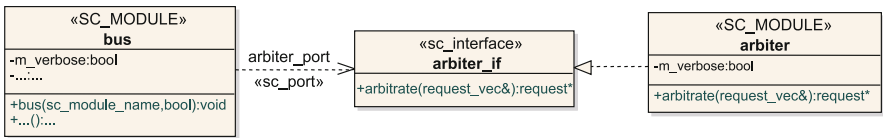


Fig. 2.3 UML arbiter interface

In practice, the first step is often skipped and the additional semantics weak, leaving room for several interpretations.

Definitions of stereotypes are often given in the form of a table. In most cases, an additional set of Class Diagrams is given, as depicted for example in Fig. 2.4, which shows an excerpt from the UML profile for SoC [42], which will be further discussed in Sect. 3.1. The extended UML metaclass *Port* is indicated by the keyword `<<metaclass>>`. For its definition, the stereotype *SoCPort* is specified with the keyword `<<stereotype>>` and linked with an extension relationship (solid line link with

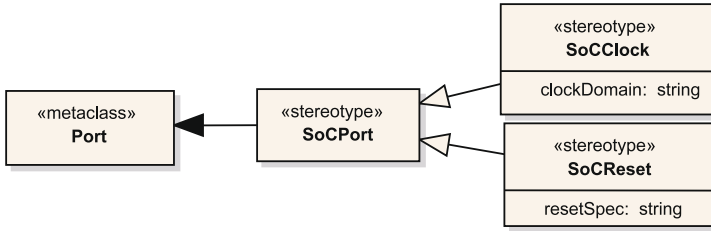


Fig. 2.4 UML stereotype definition

black head). The two other extensions, *SoCClock* and *SoCReset*, are simply specified with their tagged values as generalizations of *SoCPort*. After having defined those extensions, the stereotypes **SoCPort**, **SoCClock**, and **SoCReset** can be applied in Class Diagrams.

Several UML profiles are available as OMG standards and applicable to electronic and embedded systems modeling, such as the UML Testing Profile [39], the UML Profile for Modeling Quality of Service (QoS) and Fault Tolerance Characteristics and Mechanisms [40], the UML Profile for Schedulability, Performance and Time (SPT) [41], the UML Profile for Systems Engineering (which defines the SysML language) [38], the UML Profile for SoC [42], and MARTE (Modeling and Analysis of Real-Time Embedded Systems) [43].

The following sections will focus on the most important ones in the context of SoC design.

3 UML Extensions Applicable to SoC Design

3.1 UML Profile for SoC

The UML profile for SoC was initiated by CATS, Rational (now part of IBM), and Fujitsu in 2002. It is available as an OMG standard since August 2006 [42]. It targets mainly Transaction Level Modeling (TLM) SoC design and defines modeling concepts close to SystemC. Table 2.1 gives a summary of several stereotypes introduced in the profile and the UML metaclasses they extend.

The SoC profile introduces *Structure Diagrams* with special symbols for hierarchical modules, ports, and interfaces. The icons for ports and interfaces are similar to those introduced in [23]. Annex A and B of the profile provide more information on the equivalence between these constructs and SystemC concepts. Automatic SystemC code generation from UML models based on the SoC Profile is supported by tools from CATS [12] and the UML tool vendor ArtisanSW [19].

Table 2.1 Examples of stereotypes defined in the UML profile for SoC

SoC model element	Stereotype	UML metaclass
Module	SoCModule	Class
Process	SoCProcess	Operation
Data	Data	Class
Controller	Controller	Class
Protocol Interface	SoCInterface	Interface
Channel	SoCChannel	Class
Port	SoCPort	Port/Class
Connector	SoCConnector	Connector
Clock Port	SoCClock	Port
Reset Port	SoCReset	Port
Data Type	SoCDataType	Dependency

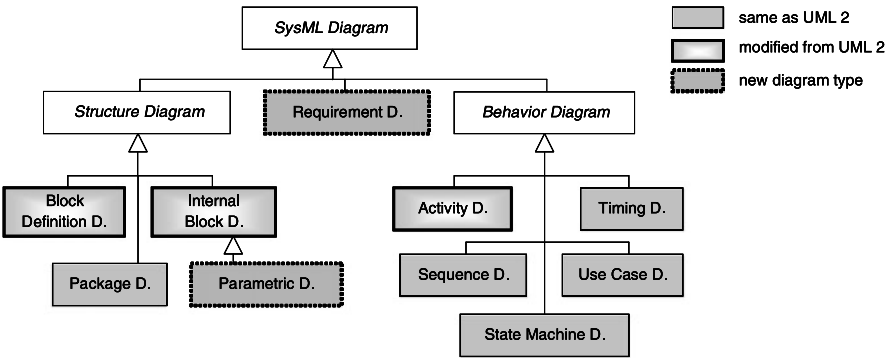


Fig. 2.5 Architecture of SysML

3.2 SysML

SysML is a UML profile which allows modeling systems from a domain neutral and Systems Engineering (SE) perspective [38]. It is the result of a joint initiative of OMG and the International Council on Systems Engineering (INCOSE). The focus of SE is the efficient design of complex systems which include a broad range of heterogeneous domains, including hardware and software. SysML provides opportunities to improve UML-based SoC development processes with the successful experiences from the SE discipline [59]. Strong similarities exist indeed between the methods used in the area of SE and complex SoC design, such as the need for precise requirements management, heterogeneous system specification and simulation, system validation and verification. The architecture of SysML is represented on Fig. 2.5. The main differences are summarized hereafter:

- **Structure:** SysML simplifies the UML diagrams used to represent the structural aspects of a system. It introduces the concept of *block*, a stereotyped class which

describes a system as a structure of interconnected parts. A block provides a domain neutral modeling element that can be used to represent the structure of any kind of system, regardless of the nature of its components. In the context of a SoC, these components can be hardware or software based as well as analog or digital.

- **Behavior:** SysML provides several enhancements to Activity Diagrams. In particular, the control of execution is extended such that running actions can be disabled. In UML, the control is limited to the determination of the moment when actions start. In SysML a behavior may not stop itself. Instead it can run until it is terminated externally. For this purpose SysML introduces *control operators*, i.e., behaviors which produce an output controlling the execution of other actions.
- **Requirements:** One of the major improvements SysML brings to UML is the support for representing requirements and relating them to the models of a system, the actual design and the test procedures. UML does not address how to trace the requirements of a system from informal specifications down to the individual design elements and test cases. Requirements are often only traced to UML use cases but not to the design. Adding design rationale information which captures the reasons for design decisions made during the creation of development artifacts, and linking these to the requirements help analyze the consequences of a requirement change. SysML introduces for this purpose the *Requirement Diagram*, and defines several kinds of relationships improving the requirement traceability. The aim is not to replace existing requirements management tools, but to provide a standard way of linking the requirements to the design and the test suite within UML and a unified design environment.
- **Allocations:** The concept of *allocation* in SysML is a more abstract form of deployment than in UML. It is a relationship established during the design phase between model elements. An allocation provides the generalized capability to a source model element to a target model element. For example, it can be used to link requirements and design elements, to map a behavior into the structure implementing it, or to associate a piece of software with the hardware deploying it.

SysML presents clear advantages. It simplifies UML in several aspects, as it actually removes more diagrams than it introduces. Furthermore, SysML can support the application of Systems Engineering approaches to SoC design. This feature is particularly important, since the successful construction of complex SoC systems requires a cross-functional team with system design knowledge combined with experienced SoC design groups from hardware and software domains which are backed by an integrated tool chain. By encouraging a Systems Engineering perspective and by providing a common notation for different disciplines, SysML allows facing the growing complexity of electronic systems and improving communication among the project members.

However, SysML remains a semi-formal language, like UML. Although SysML contributes to the applicability of UML to non-software systems, it remains a semi-formal language since it lacks associated semantics. For instance, SysML blocks allow unifying the representation of the structure of heterogeneous systems but

have weak semantics, in particular in terms of behavior. As another example, the specification of timing aspects is considered out of scope of SysML and must be provided by another profile. The consequence is a risk of discrepancies between profiles which have been developed separately. SysML can be customized to model domain specific applications, and in particular support code generation towards SoC languages. First signs of interest in this direction are already visible [21, 33, 49, 59, 64].

SysML allows integrating heterogeneous domains in a unified model at a high abstraction level. In the context of SoC design, the ability to navigate through the system architecture both horizontally (inside the system at a given abstraction level) and vertically (through the abstraction levels) is of major importance. The semantic integrity of the model of a heterogeneous SoC could be ensured if tools supporting SysML take advantage of the allocation concept in SysML and provide facilities to navigate through the different abstraction layers into the underlying structure and functionality of the system. Unfortunately, such tool support is not yet available at the time of this writing.

3.3 UML Profile for MARTE

The development of the UML profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems) was initiated by the ProMARTE partners in 2005. The specification was adopted by OMG in 2007 and has been finalized in 2009 [43]. The general purpose of MARTE is to define foundations for the modeling and analysis of real-time embedded systems (RTES) including hardware aspects. MARTE is meant to replace the UML profile for Schedulability, Performance and Time (SPT) and to be compatible with the QoS and SysML profile, as conceptual overlaps may exist.

MARTE is a complex profile with various packages in the areas of core elements, design, and analysis with a strong focus on generic hardware/software component models and schedulability and performance analysis (Fig. 2.6). The profile is structured around two directions: the modeling of features of real-time and embedded systems, and the annotation of the application models in order to support the analysis of the system properties.

The types introduced to model hardware resources are more relevant for multi-chip board level designs rather than for chip development. The application of MARTE to SystemC models is not investigated, so that MARTE is complementary to the UML profile for SoC. MARTE is a broad profile and its relationship to the RTES domain is similar to the one between UML and the system and software domain: MARTE paves the way for a family of specification formalisms.

3.4 UML Profile for IP-XACT

IP-XACT was created by the SPIRIT Consortium as an XML-based standard data format for describing and handling intellectual property that enables automated con-

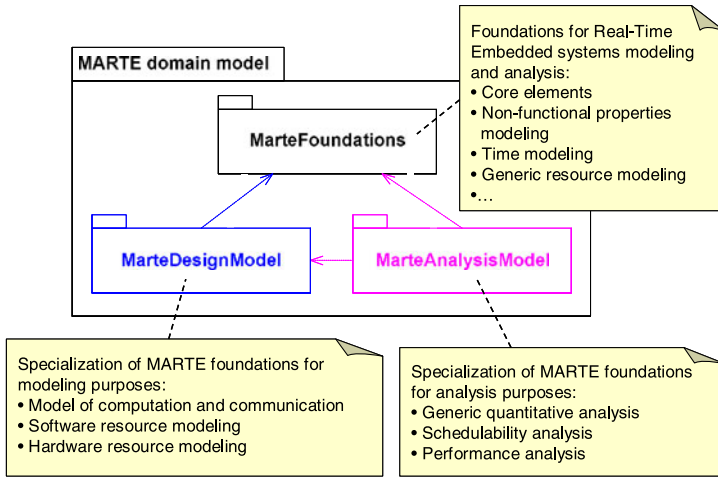


Fig. 2.6 Organization of the MARTE profile

figuration and integration. As such, IP-XACT defines and describes electronic components and their designs [46]. In the context of the SPRINT project an IP-XACT UML profile was developed to enable the consistent application of the UML and IP-XACT so that UML models provide the same information as their corresponding IP-XACT description [54]. For this, all IP-XACT concepts are mapped to corresponding UML concepts as far as possible. The resulting UML-based IP description approach enables the comprehensible visual modeling of IP-XACT components and designs.

4 Automatic SoC Code Generation from UML Models

The relationship between UML models and text code can be considered from a historical perspective as an evolution towards model-centric approaches. Originally (Fig. 2.7.a), designers were writing code having in mind their own representation of its structure and behavior. Such approach did not scale with large systems and prevented efficient communication of design intent, and the next step was code visualization through a graphical notation such as UML (Fig. 2.7.b). Round trip capability between the code and the UML model, where UML models and code remain continuously synchronized in a one-to-one relationship (Fig. 2.7.c), is supported today for software languages by several UML tools. Though technically possible [19], less tool support is available for code generation towards SoC languages. The final step in this evolution is a model-centric approach where code generation is possible from the UML model of the system towards several target languages of choice (Fig. 2.7.d) via one-to-many translation rules. Such flexible generation is still in an infancy stage. The need to unify the different semantics of the target languages and hardware/software application domains with UML constitutes here a

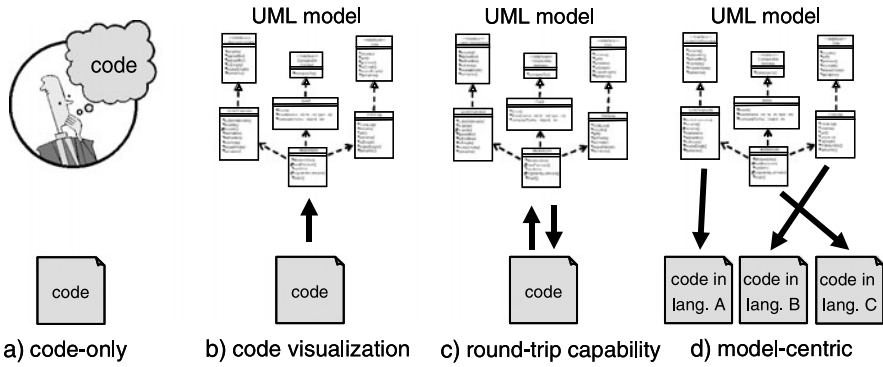


Fig. 2.7 Relationship between UML models and code

major challenge. Furthermore, the models from which code is supposed to be generated must have fully precise semantics, which is not the case with UML. Outside of the UML domain, interestingly, tools such as MATLAB/Simulink now support automatic code generation towards hardware (VHDL/Verilog) *and* software (C/C++) languages from the same model [56]. The quality of the generated code is increasing with the tool maturity, and such achievement proves the technical feasibility of model-centric development. This result has been achieved by narrowing the application domain to signal processing intensive systems, and by starting from models with well defined semantics.

In the following sections, we will investigate various combinations of UML models and SoC languages, and the associated support for code generation.

4.1 One-to-One Code Generation

A language can only be executed if its syntax and semantics are both clearly defined. UML can have its semantics clarified by customizing it towards an unambiguous executable language, i.e., modeling constructs of the target language are defined within UML, which inherits the execution semantics of that language. This procedure is typically done via the extension mechanisms of UML (stereotypes, constraints, tagged values) defined by the user or available in a profile. This one-to-one mapping between code and UML, used here as a notation complementing code, allows for reverse engineering, i.e., generation of UML diagrams from existing code (Fig. 2.7.b), as well as the automatic generation of code frames from a UML model. The developer can add code directly to the UML model or in separate files linked to the output generated from the models. The UML model no longer reflects the code if the generated output is changed by hand. Such disconnect is solved by round-trip capability supported by common UML tools (Fig. 2.7.c). This approach is typically used in the software domain for the generation of C, C++ or Java code. In the SoC

context, UML has been associated with register-transfer level (RTL) as well as electronic system level (ESL) languages. The abstraction level which can be reached in the UML models is essentially limited by the capabilities of the target language.

In Sect. 5, we introduce in more detail the application of a one-to-one code generation from the SATURN project [53]. The application is based on the extension and integration of commercial tools for SystemC/C++/Simulink co-modeling, co-simulation, and co-synthesis.

UML and RTL Languages Initial efforts concentrated on generating behavioral VHDL code from a specification expressed with UML models in order to allow early analysis of embedded systems by means of executable models [36]. However, the main focus was always to generate synthesizable VHDL from StateCharts [24] and later from UML State Machines [2, 8, 13, 14]. In the context of UML, the Class and State Machine Diagrams were the main diagrams used due to their importance in the first versions of UML. UML classes can be mapped onto VHDL entities, and associations between classes onto signals. By defining such transformation rules, VHDL code can be generated from UML models, which inherit the semantics from VHDL. Similarly, the association between UML and Verilog has also been explored.

UML and C/C++ Based ESL Languages In the late 90s, several SoC design languages based on C/C++ (e.g., SpecC, Handel-C, ImpulseC, SystemC) were developed in order to reach higher abstraction levels than RTL and bridge the gap between hardware and software design by bringing both domains into the same language base. These system level languages extend C/C++ by introducing a scheduler, which supports concurrent execution of threads and includes a notion of time. Besides these dialects, it is also possible to develop an untimed model in plain C/C++, and let a behavioral synthesis tool introduce hardware related aspects. Mentor Graphics CatapultC, Cadence C-to-Silicon Compiler and NEC CyberWorkBench are examples of such tools starting from C/C++. In all these cases, users develop a model of the system using a language coming actually from the software field. As the roots of UML lie historically in this domain, it is natural to associate UML with C/C++ based ESL languages. Although the first generation of behavioral synthesis tools in the 1990s was not a commercial success, a second generation has appeared in the recent years and is increasingly used by leading semiconductor companies for data-flow driven applications with good quality of results. In addition to the advantage of having a unified notation and a graphical representation complementary to the ESL code, it is now easier to bridge the gap between a high level specification and a concrete implementation. It is indeed possible to express the former as high level UML models, refine these and the understanding of the system until the moment where ESL code can be generated, verify the architecture and the behavior by executing the model, and eventually synthesize it. Such design flow is essentially limited by the capabilities of the chosen behavioral synthesis tool.

In the last decade, SystemC emerged as one of the most prominent ESL languages. Tailoring UML towards SystemC in a 1-to-1 correspondence was first investigated in [5, 20, 47]. Several benefits were reported when UML/SysML is asso-

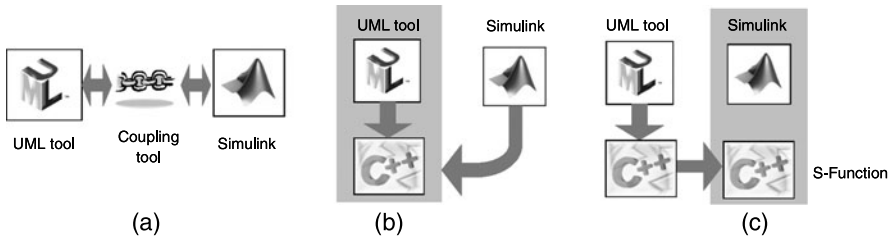


Fig. 2.8 UML and MATLAB/Simulink

ciated with SystemC, including a common and structured environment for the documentation of the system specification, the structure of the SystemC model and the system's behavior [47]. These initial efforts paved the way for many subsequent developments, whereas the introduction of several software-oriented constructs (e.g., Interface Method Calls) in SystemC 2.0 and the availability of UML 2.x contributed to ease the association between UML and SystemC. For example, efforts at Fujitsu [20] have been a driving factor for the development of the UML profile for SoC (Sect. 3.1), and STMicroelectronics developed a proprietary UML/SystemC profile [51]. Additionally, NXP and the UML tool vendor Artisan collaborated in order to extend the C++ code generator of Artisan so that it can generate SystemC code from UML models [19, 48]. This work was the starting point for further investigations which are presented in Sect. 5. It is furthermore possible to rely on a code generator which is independent of the UML tool and takes as input the XML Metadata Interchange (XMI) file format for UML models, which is text based [10, 37, 67]. The aim of all these works is to obtain quickly a SystemC executable model from UML, in order to verify as soon as possible the system's behavior and performance. UML can also be customized to represent SpecC [29, 32] or ImpulseC [65] constructs, which allows a seamless path towards further synthesis of the system. Other efforts to obtain synthesizable SystemC code from UML have also been reported [55, 67].

UML and MATLAB/Simulink Two main approaches allow coupling the execution of UML and MATLAB/Simulink models: co-simulation, and integration based on a common underlying executable language (typically C++) [60].

In the case of co-simulation (Fig. 2.8.a), Simulink and the UML tool communicate with each other via a coupling tool. Ensuring a consistent notion of time is crucial to guarantee proper synchronization between the UML tool and Simulink. Both simulations exchange signals and run concurrently in the case of duplex synchronization, while they run alternatively if they are sequentially synchronized. The former solution increases the simulation speed, whereas the time precision of the exchanged signals is higher in the latter case. As an example, the co-simulation approach is implemented in Exite ACE from Extessy [17], which allows, e.g., coupling a Simulink model with Artisan Studio [57] or IBM Rational Rhapsody [26]. Exite ACE will be further introduced in the application example given in Sect. 5. A similar simulation platform is proposed in [25] for IBM Rational Rose RealTime.

The alternative approach is to resort to a common execution language. In absence of tool support for code generation from UML, the classical solution is to generate C/C++ code from MATLAB/Simulink, using MATLAB Compiler or Real-Time Workshop, and link it to a C++ implementation of the UML model. The integration can be done from within the UML tool (Fig. 2.8.b) or inside the Simulink model (Fig. 2.8.c). This solution was formerly adopted, for instance, in the Constellation framework from Real-Time Innovation, in the GeneralStore integration platform [50], or in IBM's Telelogic Rhapsody and Artisan Software Studio. Constellation and GeneralStore provide a unified representation of the system at model level on top of code level. The Simulink subsystem appeared in Constellation as a component, which can be opened in MATLAB, whereas a UML representation of the Simulink subsystem is available in GeneralStore, based on precise bidirectional transformation rules.

The co-simulation approach requires special attention to the synchronization aspect, but allows better support for the most recent advances in UML 2.0, the UML profile for SoC and SysML, by relying on the latest commercial UML tools. On the other hand, development frameworks which rely on the creation of a C++ executable model from UML and MATLAB/Simulink give faster simulations.

One of the advantages of combining UML with Simulink compared to a classical Simulink/Stateflow solution is that UML offers numerous diagrams which help tie the specification, architecture, design, and verification aspects in a unified perspective. Furthermore, SysML can benefit from Simulink by inheriting its simulation semantics in a SysML/Simulink association. UML tool vendors are working in this direction and it will be possible to plug a block representing a SysML model into Simulink. Requirements traceability and documentation generation constitute other aspects for potential integration between SysML and Simulink, as several UML tool vendors and Simulink share similar features and 3rd party technology.

4.2 One-to-Many Code Generation

Some UML tools, such as Mentor Graphics Bridgepoint [11] or Kennedy Carter iUML [30], support the execution of UML models with the help of a high-level action language whose semantics is defined by OMG, but not its syntax. As a next step, code in a language of choice can be generated from the UML models by a *model compiler* (Fig. 2.7.d). In contrast to the one-to-one relationship described in previous section, there is not necessarily a correspondence between the structure of the model and the structure of the generated code, except that the behavior defined by the model must be preserved. Such an approach, often called *executable* (xUML) or *executable and translatable UML* (xtUML), is based upon a subset of UML which usually consists of Class and State Machine Diagrams. The underlying principle is to reduce the complexity of the UML to a minimum by limiting it to a semantically well-defined subset, which is independent of any implementation language. This solution allows reaching the highest abstraction level and degree of independence

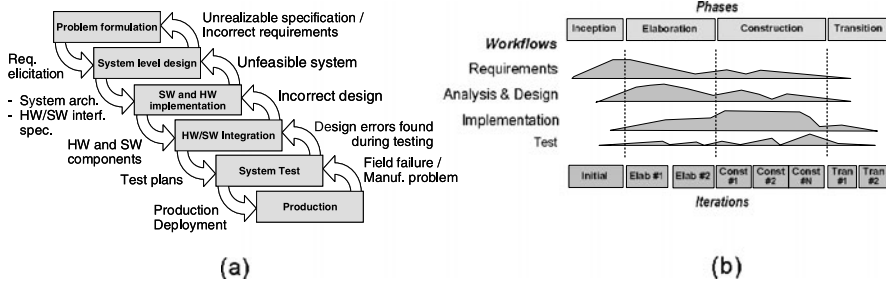


Fig. 2.9 Waterfall vs. iterative development processes (adapted from [31])

with respect to implementation details. However, this advantage comes at the cost of the limited choice of modeling constructs appropriate to SoC design and target languages available at the time of writing (C++, Ada, for example). Still, recent efforts such as [56] confirm that approaches based on a one-to-many mapping may gain maturity in the future and pave the road towards a unified design flow from specification to implementation. In particular, a behavioral synthesis technology from UML models towards both RTL languages and SystemC has become available recently [3]. Provided that synthesis tools taking as input C or C++ based SoC languages gain more popularity, xtUML tools could as well support in theory flexible generation of software and hardware implementations, where the software part of the system is produced by a model compiler optimizing the generated code for an embedded processor, while the hardware part is generated targeting a behavioral synthesis tool.

4.3 Methodological Impact of Code Generation

UML is often and wrongly considered as a methodology. UML is essentially a rich and complex notation that can address complex systems and help improve cross-disciplinary communication. Its application should be guided by a development process that stipulates which activities should be performed by which roles during which part of the product development. The absence of a sound methodology and poor understanding of the purposes of using UML lead inevitably to failures and unrealistic expectations [7].

Nevertheless, the possibility of generating code from UML models has a methodological impact, by enabling an iterative design flow instead of a sequential one. Modern development processes for software [31], embedded software [15], and systems engineering [4] follow iterative frameworks such as Boehm's spiral model [9]. In disciplines such as automotive and aerospace software development, however, we can still find processes relying on sequential models like the waterfall [52] and the V-model [18], due to their support of safety standards such as IEC 61508, DIN V VDE 0801, and DO 178-B. A traditional waterfall process (Fig. 2.9.a) assumes

a clear separation of concerns between the tasks which are executed sequentially. Such a process is guaranteed to fail when applied to high risk projects that use innovative technology, since developers cannot foresee all upcoming issues and pitfalls. Bad design decisions made far upstream and bugs introduced during requirements elicitation become extremely costly to fix downstream. On the contrary, an iterative process is structured around a number of iterations or microcycles, as illustrated on Fig. 2.9.b with the example of the Rational Unified Process [31]. Each of these involves several disciplines of system development running in parallel, such as requirements elicitation, analysis, implementation, and test. The effort spent in each of these parallel tasks depends on the particular iteration and the risks to be mitigated by that iteration. Large-scale systems are incrementally constructed as a series of smaller deliverables of increasing completeness, which are evaluated in order to produce inputs to the next iteration. The underlying motivation is that the whole system does not need to be built before valuable feedback can be obtained from stakeholders inside (e.g., other team members) or outside (e.g., customers) the project.

Iterative processes are not restricted to the software domain or to UML: as an example, model-centric design flows based on Simulink [56], where models with increasing levels of details are at the center of the specification, design, verification, and implementation tasks, belong to the same family of design flows. The possibility to generate C/C++ and VHDL/Verilog code from Simulink models share similarities with the code generation capability of UML tools. In the context of SoC design, executable models based on UML and ESL languages provide a means to support iterative development process customized towards SoC design, as proposed in [47]. Automatic code generation from UML models enables rapid exploration of design alternatives by reducing the coding effort. Further gain in design time is possible if UML tools support code generation towards both hardware and software implementation languages, and if the generated code can be further synthesized or cross-compiled. Further examples of SoC design flows based on UML can be found in [6, 28, 61, 66].

5 Application Design Example

In the remainder of this chapter, we will present a complete application example illustrating the configuration of a UML editor for the purpose of SystemC-based modeling, automatic one-to-one code generation, simulation and synthesis. The approach has been developed in the ICT project SATURN (FP7-216807) [53] to close the gap between UML based modeling and simulation/synthesis of embedded systems. More precisely, SATURN extends the SysML editor ARTiSAN Studio for the co-modeling of synthesizable SystemC, C/C++, and Matlab/Simulink; the generated code implements a SystemC/C/C++/Simulink co-simulation based on EXITE ACE from EXTESSY.

Before we go into technical details, we first present the SATURN design flow and introduce the different SATURN UML profiles.

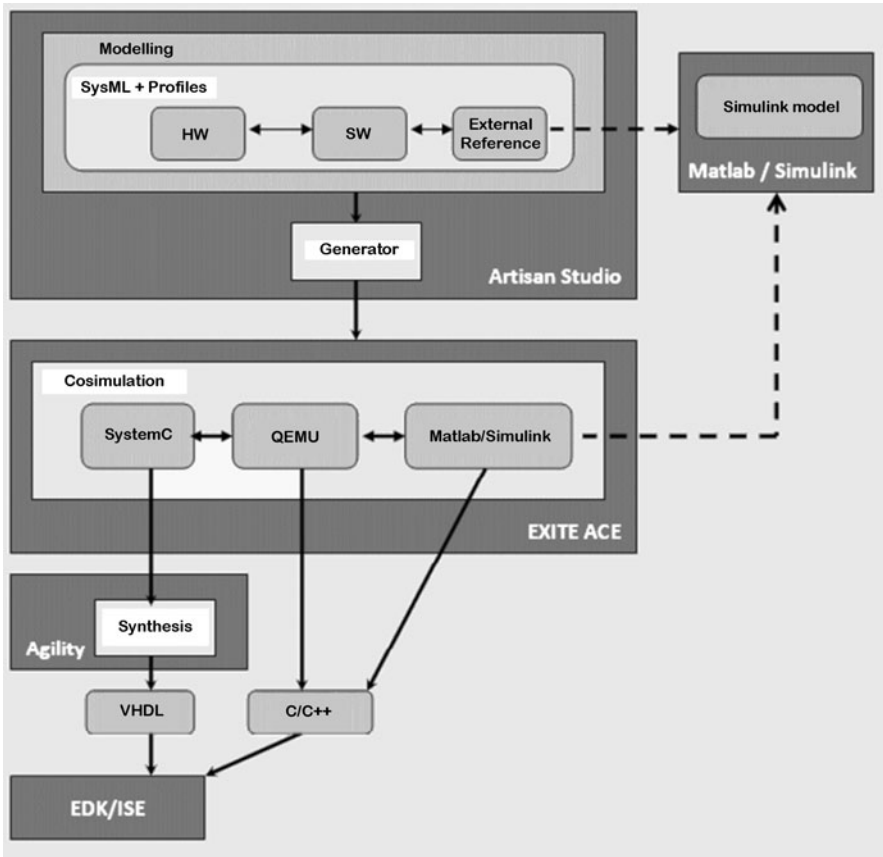


Fig. 2.10 The SATURN design flow

5.1 Methodology

The SATURN design flow, shown in Fig. 2.10, is defined as a front-end flow for industrial designs based on FPGAs with integrated microcontrollers such as the Xilinx Virtex-II-Pro or Virtex-5 FXT, which integrate PowerPC 405 and PowerPC 440 microcontrollers. The flow starts with the SysML editor Artisan Studio, which was customized by additional UML profiles for synthesizable SystemC, C/C++ and MATLAB/Simulink. As such, the developer takes early advantage of UML/SysML to capture the system requirements and proceeds to hardware/software partitioning and performance estimation without changing the UML-based tool environment. To support IP integration, the reference to different external sources is supported, i.e., MATLAB/Simulink models and C/C++ executables running on different CPUs and operating systems. Though the design flow is directed towards the SystemC subset synthesizable by the Agility SystemC compiler [1] extended by the special features of the synthesis tool, the general principles are not limited to synthesizable Sys-

temC and Agility. Other back-ends tools and synthesizable subsets, such as Mentor Graphic's CatapultC and FORTE's Cynthesizer, could be supported as well through additional UML profiles. After creating the model, a one-to-one code generation is carried out by the ACS/TDK code generation framework. The code generator is implemented by the Template Development Kit (TDK). The Automated Code Synchronization (ACS) automatically synchronizes the generated code with the model.

In a first step, code generation is applied for simulation purposes. ACS generates SystemC models for simulation as well as interface software for full system mode co-simulation with the QEMU software emulator. The additionally generated makefiles and scripts implement the design flow automation such as the compilation of the C/C++ files to an executable and the OS image generation for QEMU. This tool flow also covers C code generated from MATLAB/Simulink models, e.g., by Mathworks RealTime Workshop or dSPACE TargetLink which can be compiled for the target architecture and executed by QEMU and co-simulated with SystemC.

QEMU is a software emulator based on binary code translation which is applied in replacement to an Instruction Set Simulator. It supports several instruction set architectures like x86, PPC, ARM, MIPS, and SPARC. There is typically no additional effort to port the native binaries from QEMU to the final platform. The simulation is currently based on the semantics of a TLM 1.0 blocking communication. The integration with QEMU applies shared memory communication with QEMU in a separate process. The co-simulation with other simulators like Simulink is supported by means of the EXITE ACE co-simulation environment, e.g., for test-bench simulation.

After successful simulation, the synthesizable SystemC code can be further passed to Agility for VHDL synthesis. The design flow follows then conventional lines, i.e., the Xilinx EDK/ISE tools takes the VHDL code as input and generates a bitstream file which is finally loaded with the OS image to the FPGA.

The next section will outline more details of the SATURN UML profiles, before we describe a modeling example and provide further details on code generation.

5.2 The SATURN Profiles

The SATURN profile is based on SysML and consists of a set of UML profiles:

- UML profile for synthesizable SystemC
- UML profile for Agility
- UML profile for C/C++ and external models

UML Profile for Synthesizable SystemC The UML Profile for synthesizable SystemC is introduced as a pragmatic approach with a focus on structural SystemC models. Graphical symbols for some stereotypes like interfaces and ports are inherited from the SystemC drawing conventions. The stereotypes of the profile provide a semantics oriented towards SystemC to SysML constructs in SysML Internal Block

Table 2.2 UML profile for synthesizable SystemC

SystemC concept	UML stereotypes	Base class
<i>sc_main</i>	« <i>sc_main</i> »	<i>Class</i>
<i>sc_module</i>	« <i>sc_module</i> »	<i>Class</i>
<i>sc_interface</i>	« <i>sc_interface</i> »	<i>Interface</i>
<i>sc_port</i>	« <i>sc_port</i> »	<i>Port</i>
<i>sc_in</i>	« <i>sc_in</i> »	<i>Port</i>
<i>sc_out</i>	« <i>sc_out</i> »	<i>Port</i>
<i>sc_out</i>	« <i>sc_out</i> »	<i>Port</i>
<i>sc_signal</i>	« <i>sc_signal</i> »	<i>Property, Connector</i>
<i>sc_fifo</i>	« <i>sc_fifo</i> »	<i>Property, Connector</i>
<i>sc_clock</i>	« <i>sc_clock</i> »	<i>Class</i>
<i>sc_method</i>	« <i>sc_method</i> »	<i>Action</i>
<i>sc_trace</i>	« <i>sc_trace</i> »	<i>Property</i>

Diagrams, such as blocks, parts, and flowports. Table 2.2 gives an overview of all stereotypes for synthesizable SystemC.

A stereotype «*sc_main*» defines the top-level module containing the main simulation loop with all of its parameters as attributes. The top level module may be composed of a set of *sc_modules* as the fundamental building blocks of SystemC. For this purpose, the «*sc_module*» stereotype is defined and applied to a SysML block. The debugging of tracing signals and variables is supported through the application of the «*sc_trace*» stereotype.

In order to connect modules, dedicated stereotypes for in, out, and inout ports are provided. Those stereotypes allow refining a SysML flowport as a SystemC primitive port. In SystemC, the *sc_in*, *sc_out*, and *sc_out* ports indicate specialized ports using the interface template like *sc_signal_in_if*(*T*). The «*sc_port*» stereotype is applied to a SysML standard port through which SystemC modules can access a channel interface.

Ports connect to channels or other ports, optionally via interfaces. Regarding channels, the profile supports signals and complex channels like fifos. The «*sc_clock*» stereotype is applied to declare clocks in the SystemC model. Although clocks are not synthesizable, they are required for simulation purposes.

In order to model the system behavior, SystemC provides *sc_threads*, *sc_cthreads*, and *sc_methods*. The SystemC profile currently only supports *sc_methods* in its first version. As *sc_methods* do neither include wait statements nor explicit events, this limitation makes designs less error-prone and simplifies the task of code generation.

UML Profile for Agility SATURN currently applies the Agility SystemC compiler [1] to transform TLM-based SystemC models to RTL or EDIF netlists, which can be further processed by Xilinx ISE. The tool specific properties of Agility has been defined by a separate UML profile. Alternative synthesis tools like CatapultC

Table 2.3 UML profile for synthesizable SystemC

Agility concept	UML stereotypes	Base class
<i>ag_main</i>	«ag_main»	<i>Class</i>
<i>ag_global_reset_is</i>	«ag_global_reset_is»	<i>Port</i>
<i>ag_ram_as_blackbox</i>	«ag_black_box»	<i>Property</i>
<i>ag_add_ram_port</i>	«ag_add_ram_port»	<i>Property</i>
<i>ag_constrain_port</i>	«ag_constrain_port»	<i>Port</i>
<i>ag_constrain_ram</i>	«ag_constrain_ram»	<i>Property</i>

Table 2.4 UML profile for C/C++ extensions and external models

UML stereotypes	Base class
«cpu»	<i>Class</i>
«executable»	<i>Action</i>
«external»	<i>Class</i>

can be integrated by the definition of alternative UML profiles along the lines of the following approach. In order to allow the understanding of the code by Agility, a few extensions to the SystemC profile have to be defined and are summarized in Table 2.3. The designer is indeed able to insert some statements with *ag_* prefix into the SystemC code, which will be processed by the Agility compiler. These statements are basically pragmas which are transparent for simulation and only activated by the Agility compiler during synthesis. These pragmas provide additional synthesis information for SystemC entities. As a result, Agility stereotypes can only be assigned to object which have already a stereotype from the SystemC profile.

Agility identifies *ag_main* as a top level module for synthesis. An asynchronous global reset of internal registers and signals is defined by *ag_global_reset_is*. Additionally, Agility supports references to native implementations of RAMs through *ag_ram_as_blackbox*. In VHDL, for instance, this generates a component instantiation with appropriate memory control/data ports. The internal behavior could then, for instance, be linked with a netlist of a platform specific RAM. Through *ag_add_ram_port*, an array can be declared as a single or dual port RAM or ROM. *ag_constrain_port* allows assigning manually a specific VHDL type to a port, different to the standard port types which are *std_logic* for single bit ports and *numeric_std.unsigned* otherwise. By default *ag_constrain_ram* declares an array as a single-port RAM with one read-write port. However, most RAMs such as the BlockRAM of the Xilinx Virtex series are also configurable as multi-port memories. Through the corresponding stereotype, ROMs as well as RAMs with true dual port capabilities can be implemented.

UML Profile for C/C++ and External Models Additional basic extensions to the SystemC profile have to be defined for the purpose of hardware/software co-modeling. They are listed in Table 2.4.

A basic feature for software integration to TLM-based SystemC models is supported by `«cpu»` which indicates a SysML block as a Central Processing Unit characterized by (i) its architecture (Register, RISC, CISC, etc.) and (ii) the Operation System (OS) running on top of its hardware.

For a CPU the `«executable»` stereotype is used to define an instance of a C/C++ application (process) which is cross-compiled for the specific hardware under the selected operating system. In order to support software reuse `«executable»` simply refers to existing C/C++ source code directories managed by makefiles. Though currently not supported the stereotype can be easily extended for full UML software modeling by means of activity or state machine diagrams. Finally, the `«external»` stereotype is introduced to interface the design with arbitrary native models which are supported by the underlying simulation and synthesis framework. Currently, the additional focus is on MATLAB/Simulink models as their integration is covered by the EXITE ACE co-simulation environment.

5.3 Co-modeling

Modeling starts in ARTiSAN Studio by loading the individual UML profiles and libraries which are hooked on to the integrated SysML profile.

Thereafter, as a first step, the design starts with the specification of a SysML Block Definition Diagram (BDD), which is based on the concepts of structured UML classes. In a BDD, users specify modules and clocks as blocks as well as their attributes and operations. A relationship between different blocks indicates the hierarchical composition. Figure 2.11 shows the BDD of a simple example, consisting of a design with a top level block, a PPC405 CPU, and a SystemC model for an FPGA, which has several subcomponents like clock, PLB bus and some transactors.

For the definition of the architecture of a design expressed in SystemC, C or Simulink, SysML Internal Block Diagrams (IBDs) are applied in a second step. Hereby, IBD blocks and parts are defined as instances of the BDD. Each SystemC block is defined by a simple activity diagram with one action for each method. Each method is defined as plain sequential ASCII code. This approach is motivated by several studies which have shown that it is more efficient to code SystemC at that level as textual code rather than by activity or state machine diagrams. Additional studies have shown that it is not very effective to represent 1-dimensional sequential code through 2-dimensional diagrams. Non-trivial models may easily exceed the size of one A4 pages which is hard to manage as a diagram.

In order to map software executables to processor components, i.e., blocks stereotyped with `«cpu»`, the SATURN profile applies SysML allocations. Figure 2.12 shows the principles mapping a SysML block stereotyped with `«executable»` to a processor instance. In IBDs such an association is indicated by the name of the allocated software executable in the *allocatedFrom* compartment. Additionally, the *allocatedTo* compartment of the software block lists the deployment on the processor platform. As it is shown in the properties of a software block, each executable

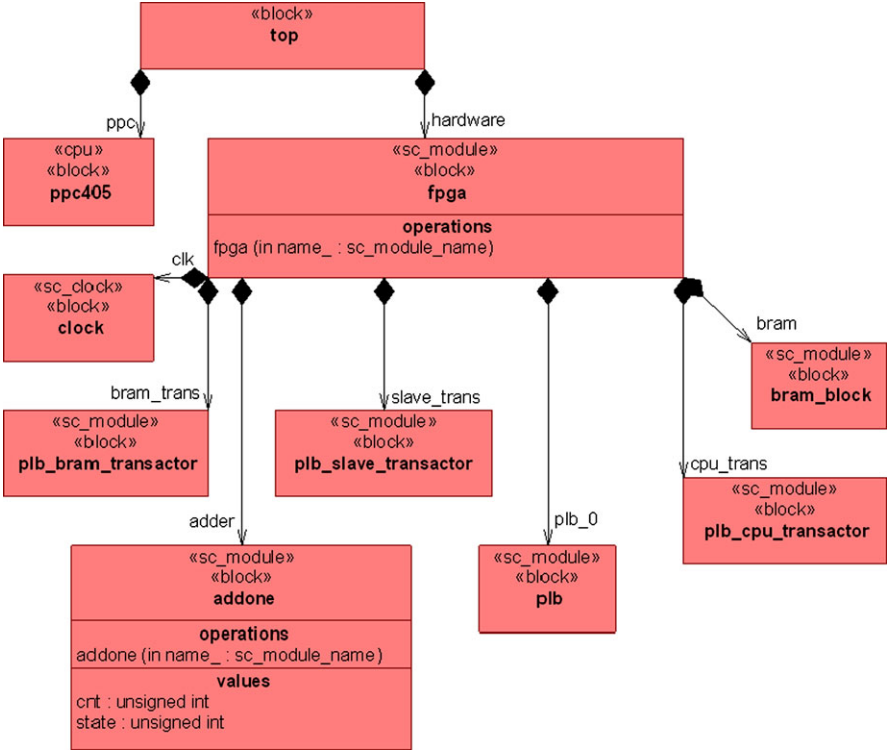


Fig. 2.11 Block definition diagram example

Fig. 2.12 Software allocation example

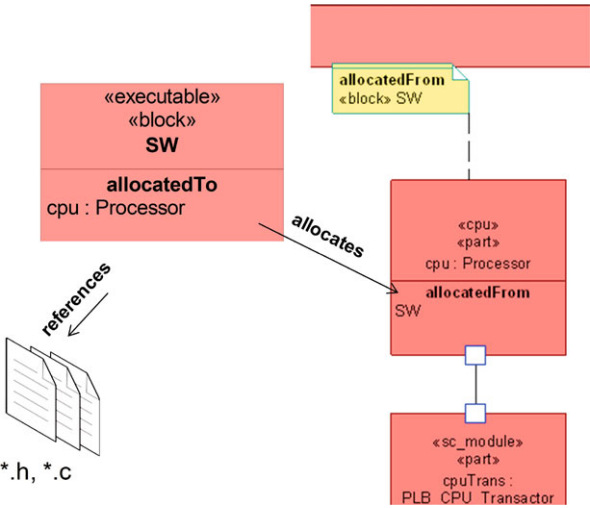
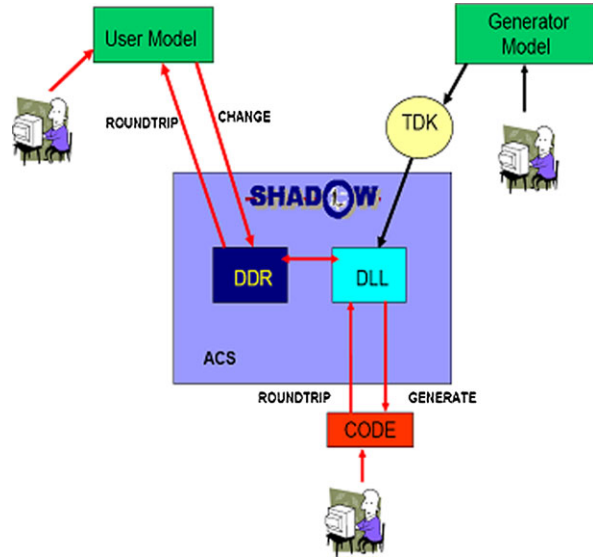


Fig. 2.13 ARTiSAN studio code generation



has the tagged value *directory* linked to the stereotype *«executable»* that refers to the directory of the source code. This provides a flexible interface to integrate arbitrary source code which also could be generated by any other software environment or any UML software component based on the Artisan Studio C profile.

5.4 Code Generation

The previous UML profiles are introduced to give adequate support for SystemC/C++-based code generation. By means of the introduced stereotypes, individual objects receive additional SystemC/C++ specific information. To better understand the complete customization, we briefly outline the concepts of the ARTiSAN Studio's retargetable code generation and synchronization, which is composed of two components: the Template Development Kit (TDK) and the Automated Code Synchronization (ACS).

As outlined in Fig. 2.13, the code generation starts with the user model which has been entered into the SysML editor. After starting ACS, the user model is first transformed into an internal Dynamic Data Repository (DDR), which saves the model in an internal representation of the user model. Each time the user model is modified, Shadow ACS is triggered, the DDR updated, and new code generated by a code generator dll. For reverse-engineering, the ACS can also be triggered by the changes of the generated code finally updating the user model. The code generator itself is defined by a *Generator Model* through TDK. A *Generator Model* is a model of the code generation, which is mainly composed of transformation rules written in a proprietary code generation description language, i.e., the SDL Template Language. This language has various constructs, through which all elements

and properties of the user model can be retrieved and processed. The following is a list of main constructs like conditional statements, for loops and an indicator of the current object. Note that all keywords are identified by %.

- **%if ...%then ... { %elseif ...%then ...} [%else ...] %endif**
Conditional Statement
- **%for (<listexpr>) ... %endfor**
A loop through all objects in <listexpr>.
- **%current**
Variable identifying the current object.

The following example shows an SDL excerpt for generating of SystemC code from SystemC stereotyped objects. The specification first goes through all classes of the model and checks them for individual stereotypes for generating different code segments. The example also sketches how to write the code of an `sc_module` header and the opening and closing brackets into a file.

```
%for "Class"
    %if %hasstereotype "sc_module" %then
        %file %getvar "FileName"
            "class_" %getlocalvar "ClassName"
                "_:\n\tpublic_sc_module\n{"
                ...
            "}"
        %endfile
    %else
        ...
    %endif
%endfor
```

Figure 2.14 gives a more complex example which takes the block name specified in the user model as the class name and generates an `sc_module` inheritance. All declarations of operations and attributes as well as implementations of constructors are exported to the header `.h` file of an individual block. All implementations of operations are written to the `.cpp` source file.

5.5 Co-simulation

The hardware-software co-simulation for the generated code (cf. Fig. 2.15) is implemented by means of EXITE ACE, which is a simulator coupling framework developed by EXTESY [17]. EXITE ACE provides an execution environment allowing for co-simulation of heterogeneous components, which are defined as the smallest function units of a system. Currently components from several tools (MATLAB, Simulink, TargetLink, ASCET, Dymola, Rhapsody in C, etc.) are supported. Each component is composed of an interface specification and a function implementation. The interface specification, which can be taken from the UML/SysML model,

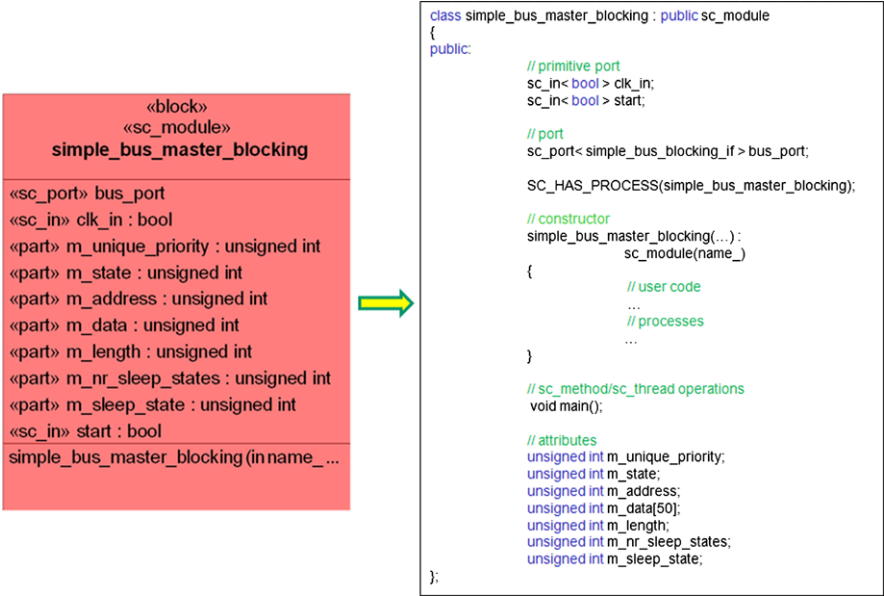


Fig. 2.14 SystemC code generation of a SysML block

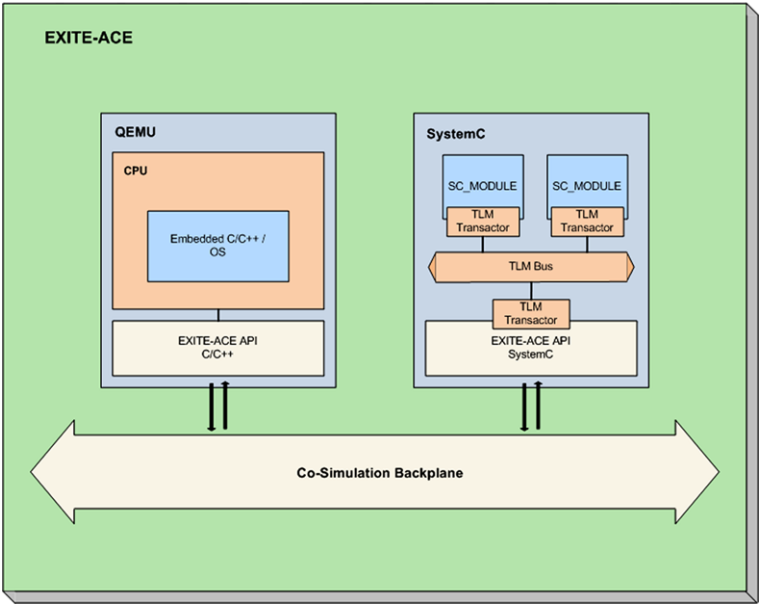


Fig. 2.15 SystemC–C/C++ co-simulation by EXITE ACE

describes the communication interface in form of ports definition. The function implementation usually refers to an executable model (for instance DLL or MDL file) incorporating computation algorithm. In the context of SystemC based verification, EXITE ACE is extended to support SystemC and QEMU components in order to allow for hardware-software co-simulation. Additionally, we extended QEMU for a blocking communication with the backplane and implemented a SystemC transactor to interface with the EXITE ACE. The transactor has to implement the individual communication policy, such as blocking or non-blocking. This architecture also supports the direct communication via shared memory between the SystemC simulator and QEMU in order to avoid the overhead of the simulation backplane.

6 Conclusions

The adoption of UML in an organization provides an opportunity to adopt new design practices and to improve the quality of the final product. Several efforts from the academic and industrial user community as well as UML tool vendors have been carried out in the recent years to investigate how tools could be extended, developed, and associated, in order to ease the use of UML for the design of electronic systems. Although UML still appears as a risky technology in this context, the situation is likely to change with the growing complexity of electronic designs and the need to specify efficiently heterogeneous systems. In addition, the increasing quality of system-level tools from EDA vendors and the expansion of UML tool vendors towards the market of electronic system design give the opportunity to bridge the gaps between the different development phases, and between the application domains. The perspective of having a unified framework for the specification, the design and the verification of heterogeneous electronic systems is gradually becoming reality. The application presented in the last section gave a first impression on the extension and integration of commercial tools into a coherent design flow for SystemC based designs. However, this is just a first step and some issues as traceability and management of synthesized objects through UML front-ends require further investigations and presumably a deeper integration of the tools.

Acknowledgements The work described in this chapter was partly funded by the German Ministry of Education and Research (BMBF) in the context of the ITEA2 project TIMMO (ID 01IS07002), the ICT project SPRINT (IST-2004-027580), and the ICT project SATURN (FP7-216807).

References

1. Agility: <http://www.mentor.com>
2. Akehurst, D., et al.: Compiling UML state diagrams into VHDL: an experiment in using model driven development. In: Proc. Forum Specification & Design Languages (FDL) (2007)
3. Axilica FalconML: <http://www.axilica.com>

4. Bahill, A., Gissing, B.: Re-evaluating systems engineering concepts using systems thinking. *IEEE Trans. Syst. Man Cybern., Part C, Appl. Rev.* **28**, 516–527 (1998)
5. Baresi, L., et al.: SystemC code generation from UML models. In: *System Specification and Design Languages*. Springer, Berlin (2003). Chap. 13
6. Basu, A.S., et al.: A methodology for bridging the gap between UML & codesign. In: Martin, G., Mueller, W. (eds.) *UML for SoC Design*. Springer, Berlin (2005). Chap. 6
7. Bell, A.: Death by UML fever. *ACM Queue* **2**(1) (2004)
8. Björklund, D., Lilius, J.: From UML behavioral descriptions to efficient synthesizable VHDL. In: *20th IEEE NORCHIP Conf.* (2002)
9. Boehm, B.: A spiral model of software development and enhancement. *Computer* **21**(5), 61–72 (1988)
10. Boudour, R., Kimour, M.: From design specification to SystemC. *J. Comput. Sci.* **2**, 201–204 (2006)
11. Bridgepoint: http://www.mentor.com/products/sm/model_development/bridgepoint
12. CATS XModelink: <http://www.zipc.com/english/product/xmodelink/index.html>
13. Coyle, F., Thornton, M.: From UML to HDL: a model driven architectural approach to hardware–software co-design. In: *Proc. Information Syst.: New Generations Conf. (ISNG)* (2005)
14. Damasevicius, R., Stuikys, V.: Application of UML for hardware design based on design process model. In: *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)* (2004)
15. Douglass, B.: *Real Time UML*. Addison-Wesley, Reading (2004)
16. Electronics Weekly & Celoxica: Survey of System Design Trends. Technical report (2005)
17. Extessy: <http://www.extessy.com>
18. Forsberg, K., Mooz, H.: Application of the “Vee” to incremental and evolutionary development. In: *Proc. 5th Annual Int. Symp. National Council on Systems Engineering* (1995)
19. From UML to SystemC—model driven development for SoC. Webinar, <http://www.artisansw.com>
20. Fujitsu: New SoC design methodology based on UML and C programming languages. *Find* **20**(4), 3–6 (2002)
21. Goering, R.: System-level design language arrives. *EE Times* (August 2006)
22. Grell, D.: Wheel on wire. *C’t* **14**, 170 (2003) (in German)
23. Grötter, T., Liao, S., Martin, G., Swan, S.: *System Design with SystemC*. Springer, Berlin (2002)
24. Harel, D.: Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* **8**(3), 231–274 (1987)
25. Hooman, J., et al.: Coupling Simulink and UML models. In: *Proc. Symp. FORMS/FORMATS* (2004)
26. IBM Rational Rhapsody: <http://www.ibm.com/developerworks/rational/products/rhapsody>
27. IEEE Std 1666–2005 SystemC Language Reference Manual (2006)
28. Kangas, T., et al.: UML-based multiprocessor SoC design framework. *ACM Trans. Embed. Comput. Syst.* **5**(2), 281–320 (2006)
29. Katayama, T.: Extraction of transformation rules from UML diagrams to SpecC. *IEICE Trans. Inf. Syst.* **88**(6), 1126–1133 (2005)
30. Kennedy Carter iUML. <http://www.kc.com>
31. Kruchten, P.: *The Rational Unified Process: An Introduction*. Addison-Wesley, Reading (2003)
32. Kumaraswamy, A., Mulvaney, D.: A novel EDA flow for SoC designs based on specification capture. In: *Proc. ESC Division Mini-conference* (2005)
33. Laemmermann, S., et al.: Automatic generation of verification properties for SoC design from SysML diagrams. In: *Proc. 3rd UML-SoC Workshop at 44th DAC Conf.* (2006)
34. Martin, G., Mueller, W. (eds.): *UML for SoC Design*. Springer, Berlin (2005)
35. McGrath, D.: Unified Modeling Language gaining traction for SoC design. *EE Times* (April 2005)
36. McUmbert, W., Cheng, B.: UML-based analysis of embedded systems using a mapping to VHDL. In: *Proc. 4th IEEE Int. Symp. High-Assurance Systems Engineering* (1999)

37. Nguyen, K., et al.: Model-driven SoC design via executable UML to SystemC (2004)
38. OMG: OMG Systems Modeling Language Specification 1.1
39. OMG: UML 2.0 Testing Profile Specification v2.0 (2004)
40. OMG: UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms (2004)
41. OMG: UML Profile for Schedulability, Performance, and Time (SPT) Specification, v1.1 (2005)
42. OMG: UML Profile for System on a Chip (SoC) Specification, v1.0.1 (2006)
43. OMG: A UML Profile for MARTE (2009)
44. OMG: UML v2.2 Infrastructure Specification (2009)
45. OMG: UML v2.2 Superstructure Specification (2009)
46. Open SoC Design Platform for Reuse and Integration of IPs (SPRINT) Project. <http://www.sprint-project.net>
47. Pauwels, M., et al.: A design methodology for the development of a complex System-on-Chip using UML and executable system models. In: System Specification and Design Languages. Springer, Berlin (2003). Chap. 11
48. Ramanan, M.: SoC, UML and MDA—an investigation. In: Proc. 3rd UML-SoC Workshop at 43rd DAC Conf. (2006)
49. Raslan, W., et al.: Mapping SysML to SystemC. In: Proc. Forum Spec. & Design Lang. (FDL) (2007)
50. Reichmann, C., Gebauer, D., Müller-Glaser, K.: Model level coupling of heterogeneous embedded systems. In: Proc. 2nd RTAS Workshop on Model-Driven Embedded Systems (2004)
51. Riccobene, E., Rosti, A., Scandurra, P.: Improving SoC design flow by means of MDA and UML profiles. In: Proc. 3rd Workshop in Software Model Engineering (2004)
52. Royce, W.: Managing the development of large software systems: concepts and techniques. In: Proc. of IEEE WESCON (1970)
53. SATURN Project: <http://www.saturn-fp7.eu>
54. Schattkowsky, T., Xie, T., Mueller, W.: A UML frontend for IP-XACT-based IP management. In: Proc. Design Automation and Test Conf. in Europe (DATE) (2009)
55. Tan, W., Thiagarajan, P., Wong, W., Zhu, Y.: Synthesizable SystemC code from UML models. In: Proc. 1st UML for SoC workshop at 41st DAC Conf. (2004)
56. The Mathworks: Model-based design for embedded signal processing with Simulink (2007)
57. Thompson, H., et al.: A flexible environment for rapid prototyping and analysis of distributed real-time safety-critical systems. In: Proc. ARTISAN Real-Time Users Conf. (2004)
58. UML-SoC Workshop Website. <http://www.c-lab.de/uml-soc>
59. Vanderperren, Y.: Keynote talk: SysML and systems engineering applied to UML-based SoC design. In: Proc. 2nd UML-SoC Workshop at 42nd DAC Conf. (2005)
60. Vanderperren, Y., Dehaene, W.: From UML/SysML to Matlab/Simulink: current state and future perspectives. In: Proc. Design Automation and Test in Europe (DATE) Conf. (2006)
61. Vanderperren, Y., Pauwels, M., Dehaene, W., Berna, A., Özdemir, F.: A SystemC based System-on-Chip modelling and design methodology. In: SystemC: Methodologies and Applications, pp. 1–27. Springer, Berlin (2003). Chap. 1
62. Vanderperren, Y., Wolfe, J.: UML-SoC Design Survey 2006. Available at <http://www.c-lab.de/uml-soc>
63. Vanderperren, Y., Wolfe, J., Douglass, B.P.: UML-SoC Design Survey 2007. Available at <http://www.c-lab.de/uml-soc>
64. Viehl, A., et al.: Formal performance analysis and simulation of UML/SysML models for ESL design. In: Proc. Design, Automation and Test in Europe (DATE) Conf. (2006)
65. Wu, Y.F., Xu, Y.: Model-driven SoC/SoPC design via UML to impulse C. In: Proc. 4th UML-SoC Design Workshop at 44th DAC Conf. (2007)
66. Zhu, Q., Oishi, R., Hasegawa, T., Nakata, T.: Integrating UML into SoC design process. In: Proc. Design, Automation and Test in Europe (DATE) Conf. (2005)
67. Zhu, Y., et al.: Using UML 2.0 for system level design of real time SoC Platforms for stream processing. In: Proc. IEEE Int. Conf. Embedded Real-Time Comp. Syst. & Appl. (RTCSA) (2005)

Design Technology for Heterogeneous Embedded
Systems

Nicolescu, G.; O'Connor, I.; Piguet, C. (Eds.)

2012, XII, 480 p., Hardcover

ISBN: 978-94-007-1124-2