Des Autom Embed Syst (2008) 12: 261–292
DOI 10.1007/s10617-008-9028-9

UML for electronic systems design: a comprehensive overview

Yves Vanderperren · Wolfgang Mueller · Wim Dehaene

Received: 28 August 2007 / Accepted: 17 July 2008 / Published online: 21 August 2008

Abstract UML has been widely accepted by the software community for several years. As electronic systems design can no longer be seen as an isolated hardware design activity, UML becomes of significant interest as a unification language for systems description combining both HW and SW components. This article provides a comprehensive view of the UML applied to System-on-Chip (SoC) and hardware-related embedded systems design. The modeling concepts in the UML language are first introduced, including major diagrams for the representation of the behavior and the structure of systems. The principles behind application specific UML customizations (UML profiles) are summarized, and several examples relevant for SoC design are given, such as the SysML (System Modeling Language) and the SoC Profile. Thereafter, various approaches associating UML with existing HW/SW design languages are presented. Beyond language aspects, the article addresses the question of UML-based design flows, and shows how UML can be applied concretely to the development of electronic-based systems. The current situation about tool support constitutes the last focus of the article. In particular, we show how UML tools can be combined with well-known simulation environments, such as MATLAB.

Keywords UML · SysML · Model-based design · System specification · Modelling languages

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

Y. Vanderperren (✉) · W. Dehaene
Katholieke Universiteit Leuven (ESAT–MICAS), Leuven, Belgium
e-mail: yves.vanderperren@esat.kuleuven.be

W. Mueller
Paderborn University (C-LAB), Paderborn, Germany

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. Several entry points into the flow can be defined from high level modeling languages and abstraction levels. The application of UML in the context of electronic systems has attracted growing interest in the recent years [22, 41], and several experiences from industrial and academic users have been reported [40, 69].

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, 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 sufficient semantics, which constitutes the main obstacle for real engineering application. Therefore, application specific customizations of UML (UML profiles), such as SysML [46] and the UML Profile for SoC [51], are of increasing importance. The addition of precise semantics allows for the automatic generation of code skeleton, typically C++ or Java, from UML models. Still, the definition of a precise behavioral semantics for verification based design flows is mandatory but remains often unsolved.

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 [28]. In this domain, complexity is now successfully managed by model-based development and testing methodologies based on MATLAB/Simulink with highly efficient and partly certified code generation. Several automotive software modeling, code generation, and testing environments for Model-, Software-, and Hardware-in-the-Loop have become available in the last years, e.g., SystemDesk, AutomationDesk and TargetLink from dSPACE [21]. Additionally, UML and XMI (XML Meta Interchange—the Extensible Markup Language based exchange format for UML), are of increasing importance for the exchange of specification, modeling, and testing data, like in the AUTOSAR (AUTomotive Open System ARchitecture) standard [4].

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, several gaps in a joint unified software and hardware design flow can be identified. Beyond cultural aspects, the lack of commonly agreed standards for the exchange and integration of hardware and software components constitutes a major issue. Though IP-XACT [63] provides a first step for the exchange of Register Transfer Level (RTL) components with extensions for Electronic System Level (ESL) modeling, a huge gap remains towards the exchange of general and hardware-related software components, and its acceptance by the software community.

[1] While 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 in this article, 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. We include therefore in the term 'SoC' other technological solutions, such as System-in-Package (SiP), Multi-Chip Module (MCM), System-on-Package (SoP), etc.

In that context, UML and XMI have 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 [73] and 2007 [75]. 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 [73].

This article is structured as follows. First, a comprehensive overview of UML and the most relevant extensions to UML for electronic system design is given in Sect. 2. The association between UML and SoC languages is discussed in Sect. 3, whereas the role of UML in SoC design flows is addressed in Sect. 4. Finally, Sect. 5 investigates the question of tool support, before Sect. 6 closes with a general conclusion.

2 The Unified Modelling Language

In 1995, Version 0.8 of the Unified Modeling Language (UML) was published as one of the first stable UML versions. It unified two major approaches in object-oriented (OO) modeling and software design, i.e., Booch and Rumbaugh. In 1997, UML merged with Jacobson's diagrams, forming the OMG standard UML 1.0. UML received several extensions in the next releases, but until 1.4 it consisted essentially of a grouping of diagrams for the purpose of software documentation and databases. In order to push UML as a modeling language which allows for automatic code generation by CASE tools, UML had to become a computationally complete programming language. Therefore, action semantics was developed and became deeply integrated with UML concepts in UML 2.0, a version which constituted a fundamental change and a huge step towards metamodel-based software modeling and tools. The last version at the time of this writing is UML 2.1.2 and does not significantly differ from 2.0 from the user's perspective.

UML concepts are defined in the OMG standard [53, 54] by means of Class Diagrams, constraints, and semantical outlines. *Constraints* are either given as OCL expressions (Object Constraint Language—a complementary OMG standard) [50] or as plain text. The *semantics* is given as textual outlines which can be imprecise. UML defines currently 13 different diagrams and is based on a number of major concepts. As these are highly intertwined, the information captured in UML diagrams is often redundant and overlaps. UML tools help guarantee the consistency of UML models by managing a repository of model elements appearing in the different diagrams and the links between these. Nevertheless, keeping a coherent modeling style is crucial in order to reduce the complexity in UML diagrams. In particular, it is good practice to keep the focus of each diagram on a specific design concern.

UML concepts refer to the notion of *classifiers*, the basic building blocks of UML for structure and behavior. Examples of classifiers of UML's structural part are class, interface, component, association, and generalization. Classifiers of the behavioral part are *activity*, *interaction*, and *state machine*. In the following sections, we present an overview of the most commonly applied classifiers and corresponding diagrams used for the design of electronic systems. Although we present them as specified in the OMG standard, the reader should be aware that several tools and applications implement variations of it.

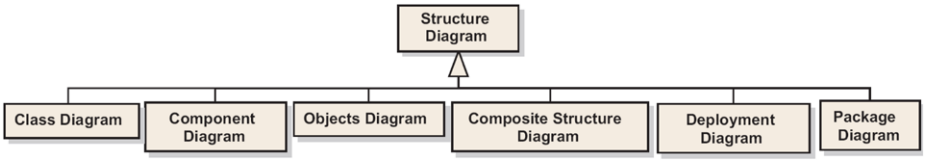
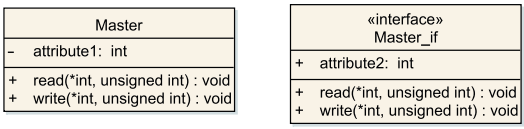


Fig. 1 UML structure diagrams

Fig. 2 UML class and interface



2.1 Structural parts

UML introduces a set of 6 structural diagrams (Fig. 1). In this section, we summarize classes, components, interfaces, objects, and relationships between them, as well as how to describe structural composition in UML.

2.1.1 Classes, components and interfaces

A *class* is a classifier which can own a set of properties and operations with different visibilities (public, private, protected). Its graphical representation (Fig. 2) is a rectangle with different sections (compartments). The upper compartment shows the *class identifier*. The other compartments may contain a list of *attributes* and *operations* as well as complementary structural definitions, if needed.

Object-oriented programming and description languages such C++ and SystemC support the concept of interfaces. In UML, an *interface* is a collection of operation signatures and attribute definitions. Figure 2 gives an example of a class and an interface, defined with the keyword «*interface*» above the interface name. An interface is provided for a set of classes. In turn, a class may implement an interface. Graphically, a half-circle stands for *required* (usage dependency) and a full-circle for *provided* (realization dependency) interface. Alternatively, the realization can be represented by a dashed line with white triangle and the usage by a dashed line with arrow, as illustrated by Fig. 3. The latter presentation is used when detailed information is preferred, instead of the interface name only.

A UML *component* is a specialized class. It is graphically represented by the keyword «*component*» or by a component icon. A component may own attributes and operations, and participate in generalizations and associations. It can provide a black-box view with its required and provided interfaces. In a white-box view, a component encapsulates several behavioral classifiers, which are listed in an extra compartment.

2.1.2 Relationships

The *realization* and *usage dependencies* introduced previously are specific forms of relationships. *Generalizations* (solid line with white triangle), *associations* (solid line), *compositions* (solid line with black diamond), and *aggregations* (solid line with white diamond) constitute other relationships defined in UML between classes. Generalizations specify inheritance relationships between classes, as supported by common object-oriented (OO) languages. Compositions and aggregations allow defining classes containing other classes, i.e.

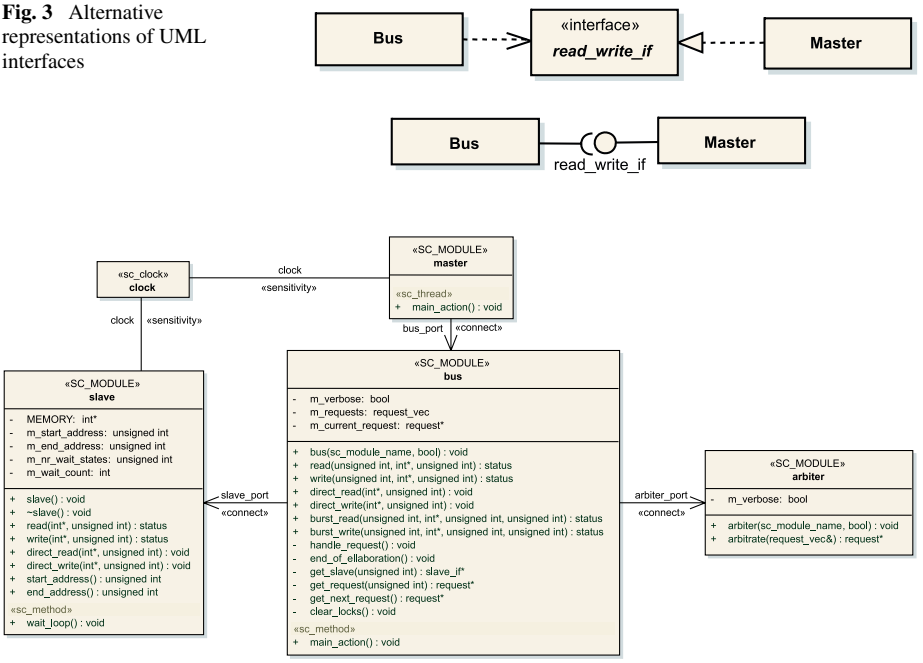


Fig. 3 Alternative representations of UML interfaces

Fig. 4 UML simple bus class diagram

structural hierarchies. These relationships are particularly useful to model OO software systems. In the context of electronic systems which include hardware parts, engineers typically compose these as instances of classes connected via associations. Their variations, such as SysML blocks connected by means of SysML connectors, may also be used and are presented in Sect. 2.3.3.

2.1.3 Stereotypes

Stereotypes allow users to define modeling elements derived from UML classifiers, such as classes and associations, and to customize these towards individual application domains. 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. 4 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 abstrac-

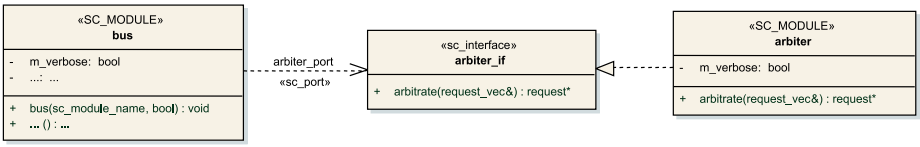


Fig. 5 UML arbiter interface

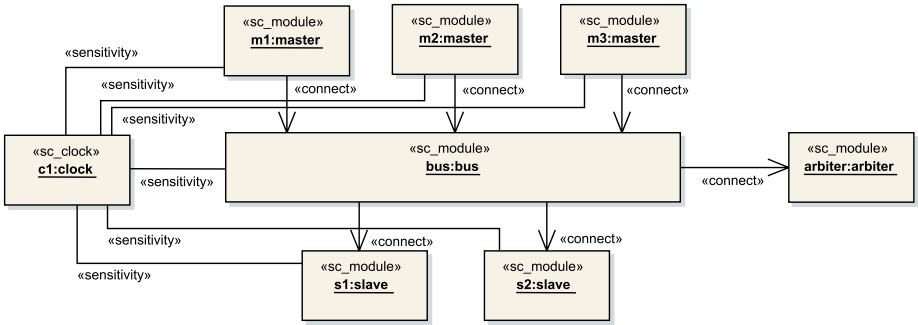


Fig. 6 UML object diagram example

tion 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. 5). 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.

2.1.4 Instances

Instantiated classes are denoted as *objects* and are graphically represented in a similar way as classes. Objects are identified by their object name, which is given by the underlined concatenation of an optional instance name, the ':' separator, and an optional class name. Though the object name can be just a name without separator, the full notation is recommended for clarity purposes. It is indeed a frequent mistake that objects are introduced or understood as classes and vice versa. Figure 6 illustrates a possible Object Diagram for the Class Diagram in Fig. 4 with three masters, two slaves and one arbiter connected to a bus. As shown by this example, the compartments for attribute values can be skipped for readability purposes.

2.1.5 Structural compositions and hierarchies

UML provides Component, Composite Structure and Deployment Diagrams to describe structural compositions. Though *Component Diagrams* are applied mostly for the representation of relationships between classes, they can be used for instances as well. A Component Diagram can give a black-box view of components with provided and required interfaces as well as delegations. Additionally, it can also show information about the internal hierarchical structure (white-box view). Since the structural information in Component Diagrams

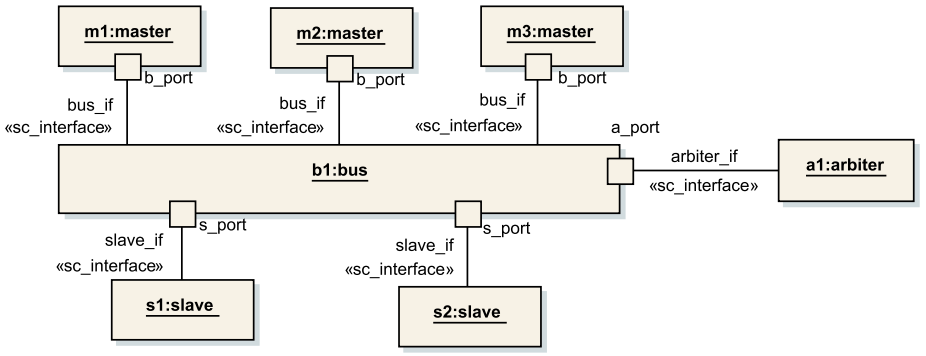


Fig. 7 UML composite structure diagram

may overlap with the corresponding Class Diagrams, Component Diagrams often provide just a complementary view of the same elements.

For engineering applications, *Composite Structure Diagrams* are the most frequently used means to represent hierarchically linked blocks. SysML's Internal Block Diagrams, for instance, are based on Composite Structure Diagrams and will be further detailed in Sect. 2.3.3. A Composite Structure Diagram depicts the internal structure of structured classifiers, such as structured classes, by describing the interaction between the internal parts, ports, and connectors. A *part* represents a set of instances which are owned by an object, for example, or instances of another classifier. Figure 7 shows a Composite Structure Diagram of our previous SystemC simple bus example. Note here that the names of the individual objects as parts are not underlined. In order to reduce complexity in diagrams, composite structures support the specification of a multiplicity after the instance name. *master* : *master* [3], for example, stands for three instances of a master.

Composite Structure Diagrams correspond to classical block diagrams, which are commonly used in engineering. These diagrams, when used with adequate stereotypes (e.g., for SystemC), provide excellent means to describe the structure of a system. They allow capturing the important but sometimes complex structural dependencies between modules, ports, and interfaces.

Alternatively, UML provides *Deployment Diagrams* to model the architecture of a system. They describe the assignment of software artifacts onto an execution platform given as nodes. The execution platform can be hardware, software, or particular middleware such as an operating system. *Nodes* are typically nested and are connected through communication paths. They cannot have ports, which limits strongly their expressiveness. As we can see from this definition, UML is mainly targeting software artifacts as development results. However, those diagrams can be applied to hardware models, as the result of individual steps of an electronic system development. In this context, Deployment Diagrams compare to the notion of a platform in SoC design. We do not go in further details here since in most cases Composite Structure Diagrams are sufficient for SoC engineering applications.

2.2 Behavioral parts

UML introduces a set of 7 behavioral diagrams (Fig. 8). These diagrams are based on the fundamental behavioral concepts given by the *action*, *interaction*, *activity*, and *state machine* classifiers. *Timing*, *Sequence*, *Communication*, and *Interaction Overview Diagrams* provide different views of interactions.

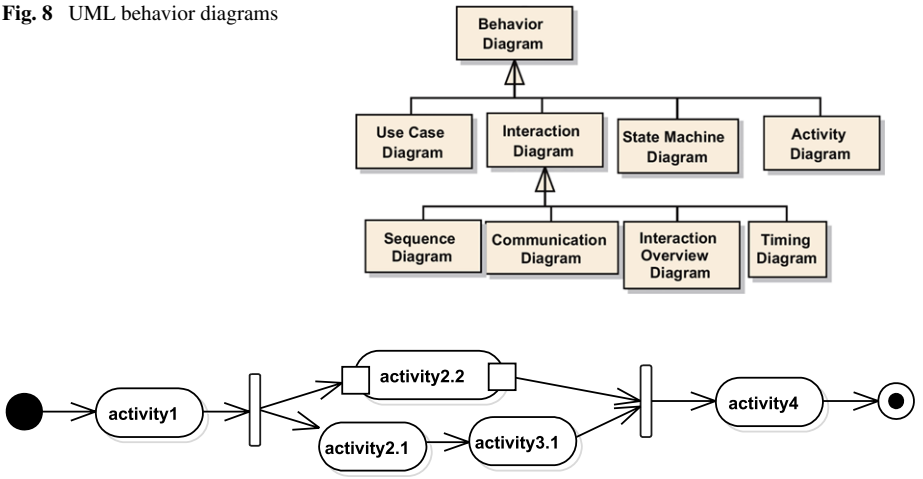


Fig. 8 UML behavior diagrams

Fig. 9 UML activity diagram example

2.2.1 Actions

UML defines a complex execution semantics based on the notion of actions. It should be emphasized that UML only defines the semantics of these actions, not the syntax. The latter depends on the tool environment which is used to enter expressions and statements. It is often a subset of the target language for code generation, such as Java or C++.

A UML *action* is its fundamental unit of behavior, which takes a set of inputs and converts them into a set of outputs. UML defines a basic set of actions to manipulate objects, variables, and links to create, delete, read, write, and clear them. However, more important for the execution semantics are the actions on which the basic communication scheme is based on. The communication between UML objects is implemented either through an *asynchronous message* (defined by a signal) or through *synchronous/asynchronous operation calls*. With an operation call, the call request is transmitted to the target object where the CallOperation action is invoked. In the case of synchronous calls, the calling object is blocked until the result is returned as a message through a Reply action. For message communication, a signal or class is instantiated and sent by a SendSignal or SendObject action. UML covers time events in expressions, where *at* specifies an absolute and *after* a relative point in time. This gives just relationships to a current global physical time where required. Beyond that notion of time, UML does not have any advanced concepts for simulation.

2.2.2 Activities

Activities are a combination of control flow and object flow graphs (Fig. 9), and are comparable to extended Petri nets. They are composed of nodes and directed edges. The existence of objects or data passing along the edge is indicated by an object flow pin (a rectangle at an activity, as illustrated in Fig. 9) or an object node (a rectangle in the middle of an edge). Otherwise, the edge defines a control flow with control token passing.

The execution semantics of an activity is based on token flows. Each action in an activity may execute zero, one, or more times for each activity execution. When a node begins execution, tokens are consumed from some or all of its input edges, and a token is placed on

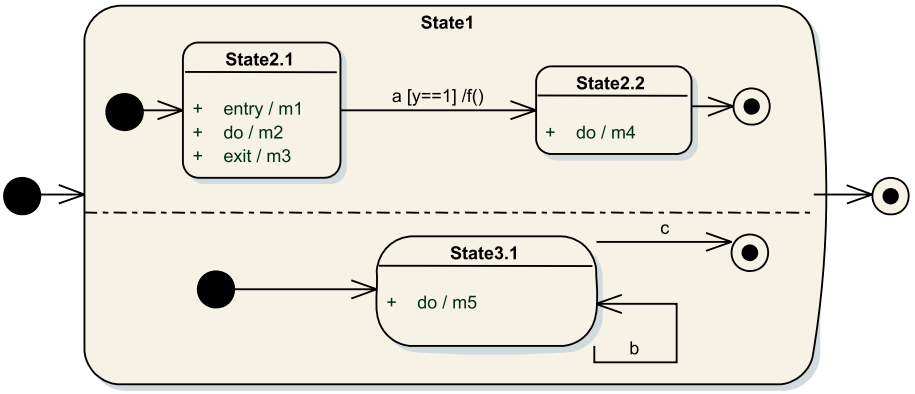


Fig. 10 UML behavior state machine example

the node. When a node completes execution, a token is removed from the node and tokens are offered to some or all of its output edges.

2.2.3 State machines

UML behavior state machines are a variant of Harel's StateCharts [30]. They are based on hierarchical finite state machines, composed of composite states with one or several orthogonal regions (Fig. 10). In addition to the classical StateCharts, UML supports the definition of choice and junction pseudo-states, as well as send and receive actions. The execution of a behavior state machine is defined by the *run-to-completion* semantics which differs slightly from Harel's StateCharts. The state machine arbitrarily dispatches an event occurrence from the event pool which is associated to its instance. The next event can only be dispatched if the processing of the current one has been completed. After the selection of an event, all transitions with corresponding trigger and true guards are enabled for firing. When a transition fires, the event is dispatched and the *do* behavior of the source state is terminated. Thereafter, the *effect* behavior of the transition, the *exit* behavior of the current state, and then the *entry* behavior of the next state are executed before the *do* behavior of the next state is executed.

2.2.4 Interactions

Interactions model the communication between concurrent objects. The most frequently applied graphical representation of interactions is in the form of *Sequence Diagrams*, which are an extension of the Message Sequence Charts (MSCs) from the Specification and Description Language (SDL) [33]. Sequence Diagrams are composed of a set of objects with a vertical lifeline. A bar on the lifeline indicates when the object is active. Horizontal links show the invocation of synchronous and asynchronous messages, where the end point of each link relates to an event occurrence. Additionally, UML provides *combined fragments* which support more advanced constructs, such as if-then-else, loops, parallel and critical sections. Lifelines can correspond to vertical time lines and the definition of timing constraints is supported.

Figure 11 shows a Sequence Diagram example of a blocking read of a bus master. The sequence starts with a read request of the master. This request initiates a bus loop with

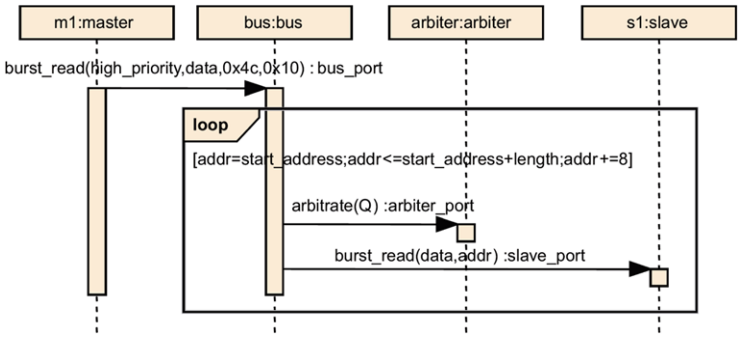


Fig. 11 UML sequence diagram example

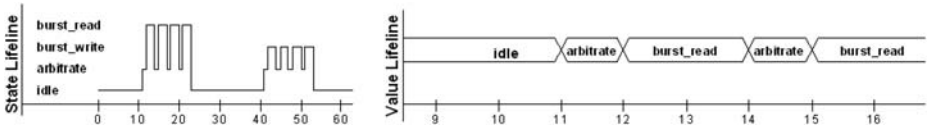


Fig. 12 UML timing diagram example

arbitration, and a burst read for single bytes from a start to an end address. Such diagram gives a comprehensive overview of the interaction of concurrently running objects. However, it lacks details and can hardly be used on its own. At least a corresponding class or object definition is required to provide the context of this diagram.

UML *Timing Diagrams* look similar to graphs with continuous waveforms, but do not directly correspond to them. UML Timing Diagrams can be lifelines for discrete states or discrete values along a horizontal time axis. Similarly to sequence diagrams, it is possible to annotate them with timing constraints. As an example, Fig. 12 illustrates a state life line on the left and a fraction of the corresponding value life line on the right for a burst read/write bus cycle.

2.2.5 Use cases

Use cases are essentially a textual description of the system's behavior in terms of its primary and secondary responses to external stimuli. The former corresponds to the expected reaction whereas the latter covers unexpected cases such as error conditions.

Use Case Diagrams can be used to represent the main functionalities of a system and their relationships. Use Case Diagrams basically depict actors interacting with the system and the services (use case titles) it should perform. Use Case Diagrams are basic and not tightly integrated with the other UML behaviors. As a result, the support of UML for the representation of use cases via Use Case Diagrams is rather limited. Nevertheless, the specification of electronic systems can greatly benefit from a use case analysis, as further detailed in Sect. 4. Developing and writing use cases requires skill and experience [39]. We refer the reader to more advance readings, such as [16, 17].

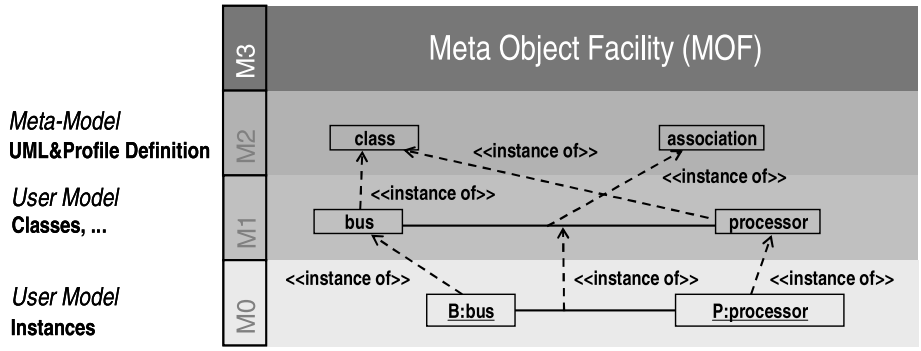


Fig. 13 UML modeling levels

2.3 UML extensions applicable to SoC design

2.3.1 The UML extension mechanism

UML itself is defined by a set of Class Diagrams with constraints and textual outlines, called the *UML metamodel*. To allow the definition of several modeling languages from a common base, the OMG defines an additional model describing the UML metamodel, i.e., the metametamodel. The metametamodel is referred to as M3 level and the metamodel as M2 level (Fig. 13). As users typically never use or apply M3 level, we skip its details here. M2 is the layer which defines a modeling language, i.e. UML in the present case. The application of UML, i.e., models developed using UML, is denoted as M1, the model layer. M0 corresponds to the run-time objects, i.e., instances of classes and state machines etc., and is called the run-time instance layer. In most cases, users apply a combination of M0 and M1 level in their daily modeling, without needing to work at M3 level.

The fact that UML is defined on the basis of a metamodel makes it extremely flexible, since an application specific customization can be easily defined by the extension of that metamodel trough the definition of stereotypes. We have already seen in Sect. 2 how to apply SystemC-oriented stereotypes in Class Diagrams. Such stereotypes have to be defined as extensions the UML metamodel first, before they can be used in Class Diagrams. Though the stereotypes were defined as extensions of the class and the interface metaclass in our examples, they can be defined for any other UML metaclass as well.

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. 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. 14, which shows an excerpt from the UML profile for SoC [51], which will be further discussed in Sect. 2.3.2. 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 black head). The two other extensions, *SoCClock* and

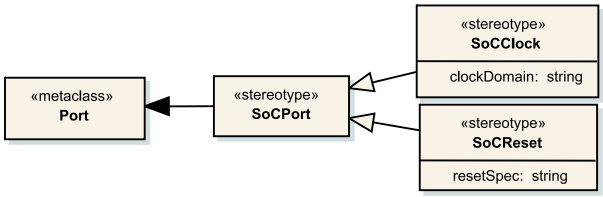


Fig. 14 UML stereotype definition

Table 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

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 [47], the UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms [48], the UML Profile for Schedulability, Performance and Time (SPT) [49], the UML Profile for Systems Engineering (SysML—System Modelling Language) [46], the UML Profile for SoC [51], and MARTE (Modeling and Analysis of Real-Time Embedded Systems) [52].

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

2.3.2 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 [51]. It targets mainly Transaction Level Modeling (TLM) SoC design and defines modeling concepts close to SystemC. Table 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 [29]. 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 and the UML tool vendor ArtisanSW (cf. Sect. 5). The subject of code generation is further discussed in Sect. 3.2.

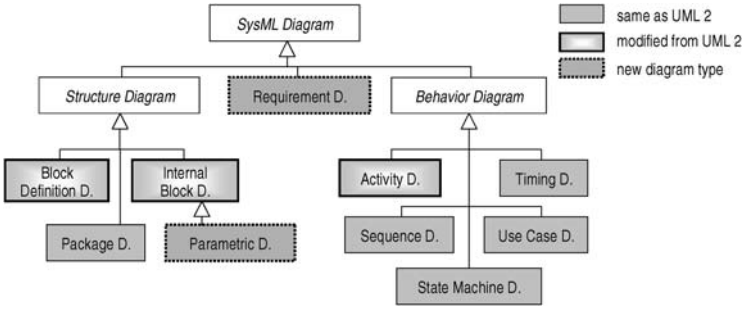


Fig. 15 Architecture of SysML

2.3.3 SysML

SysML is a UML profile which allows modeling systems from a domain neutral and Systems Engineering (SE) perspective [46]. 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,[2] including hardware and software. Strong similarities exist 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. SysML provides opportunities to improve UML-based SoC development processes with the successful experiences from the SE discipline [71]. The architecture of SysML is represented on Fig. 15.

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. A *Block Definition Diagram* defines features of a block and relationships between blocks, similarly to a Class Diagram. The *Internal Block Diagram*, the SysML version of the UML Composite Structure Diagram, shows the interconnection of the parts of a system and supports information flows between blocks, as present for example in the datapath of a signal-processing oriented SoC.

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.

[2] INCOSE defines Systems Engineering as "an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem (...). Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process which proceeds from concept to production to operation" [32].

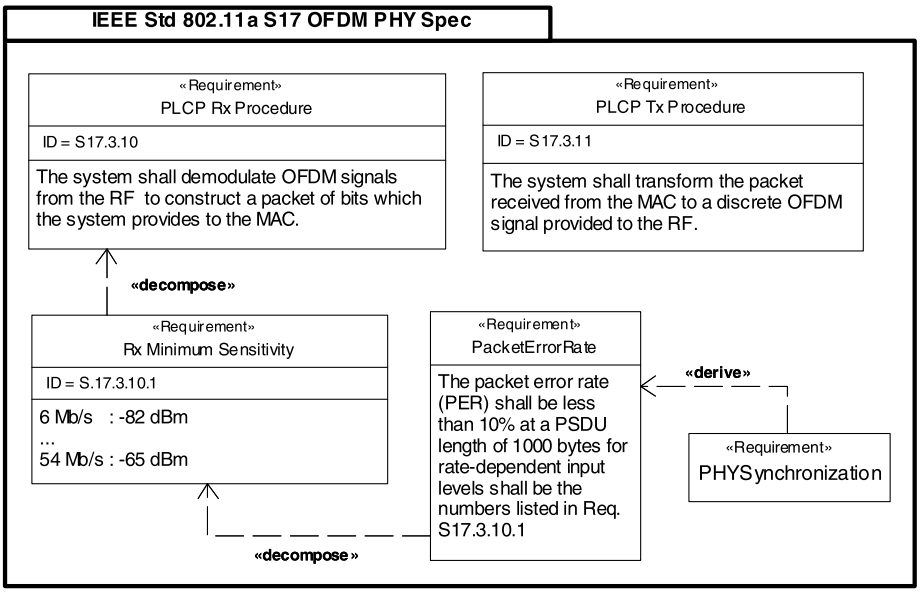


Fig. 16 Example of SysML requirement diagram

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. Use Cases help build up a sound understanding of the expected behavior of the system and validate the proposed architecture. However, requirements are often only traced to the 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. Requirements can be decomposed by means of the *containment* relationship in a similar way to that of Class Diagrams. The *trace* dependency relates derived requirements to source requirements. The system designed and the requirements are linked by a *satisfaction* dependency. Finally, the *verification* dependency associates a requirement with the test case used to verify this requirement. As an example, Fig. 16 shows how requirements from the IEEE 802.11a wireless LAN specification as well as derived implementation related requirements could be represented in SysML.

Allocations The concept of *allocation* in SysML is a more abstract form of deployment than in UML. It is a design time relationship between model elements which maps a source into a target. An allocation provides the generalized capability to allocate one model element to another. 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.

Pros and cons of SysML SysML simplifies UML in several aspects. It actually removes more diagrams than it introduces. For instance, SysML abandons the use of the Communication Diagram. The support for flows between blocks is meant to provide an equivalent and domain-neutral modeling capability. Similarly, the SysML allocation provides a more flexible deployment mechanism than Deployment Diagrams, which suit more specifically a software design viewpoint.

SysML can support the application of Systems Engineering approaches to SoC design. In recent years, classical chip design flows, which separate digital design aspects as much as possible from manufacturing issues, are increasingly threatened as the feature size of the VLSI circuit technology decreases towards the deep submicron range. This issue is exacerbated by increasing mask and design costs, which led to the distinction between fabless system providers and silicon providers. The former generally lack expertise in designing state-of-the-art submicron VLSI circuits, while the latter often do not have the global visibility and the overall system knowledge required to define a complex SoC. However, the successful construction of such systems requires a cross-functional team with system design knowledge combined with experienced SoC design groups from hardware and software domains, 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 semantics in significant parts, like UML. 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 [27, 38, 58, 71, 76].

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.

As a final remark, SysML, like UML, is not a methodology. SysML provides modeling means to enhance the communication of the design intent, but does not tell how these can be best applied within a design flow. SysML must therefore be complemented with a sound development process.

2.3.4 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 and its preliminary Beta 1 version released in 2007 [52]. The general purpose of MARTE is to define foundations for the modeling and analysis of real-time

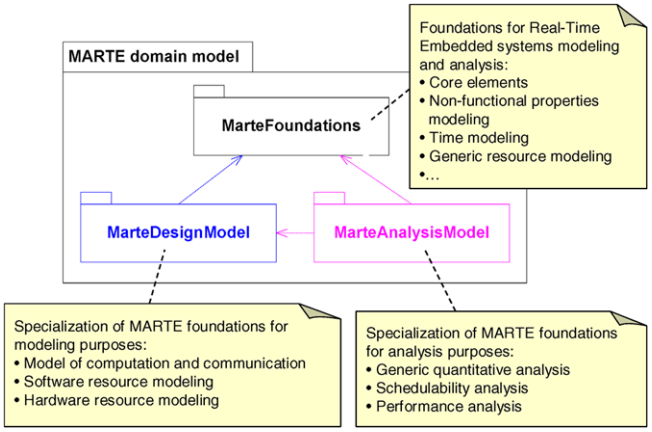


Fig. 17 Organization of the MARTE profile

embedded systems (RTES) including hardware aspects. MARTE is meant to replace the UML profile for 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 (Fig. 17). 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 complimentary 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. However, more experience from applications is needed before the pros and cons of MARTE can be assessed.

3 UML and design languages

3.1 Historical perspective

The relationship between UML models and text code follows the evolution towards model-centric approaches. Originally (Fig. 18(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. 18(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. 18(c)), is supported today for SW languages by several UML tools. Though technically possible [3], 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. 18(d)) via one-to-many translation rules. Such flexible generation is still in an infancy stage. The need to unify the different semantics of

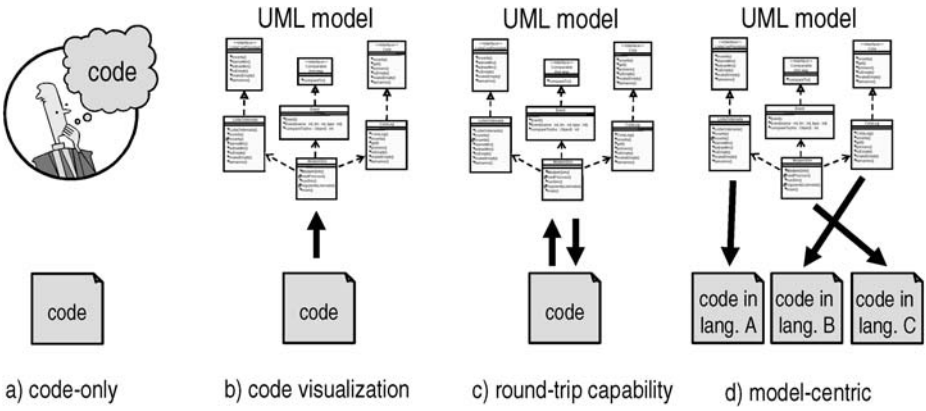


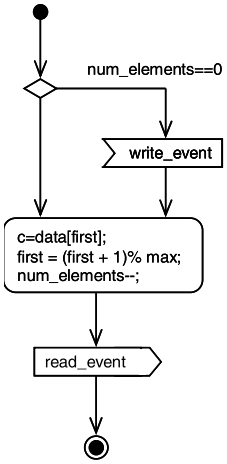
Fig. 18 Relationship between UML models and code

the target languages and HW/SW application domains with UML constitutes here a 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 support now automatic code generation towards HW (VHDL/Verilog) *and* SW (C/C++) languages from the same model [66]. 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.

3.2 One-to-one relationship

A language can only be executed if its syntax and semantics are 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 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. 18(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. 18(c)). This approach is typically used in the software domain for the generation of C, C++ or Java code. In the SoC context, UML can be associated with low level as well as electronic system level (ESL) languages. The abstraction level which can be reached is limited by the capabilities of the target language.

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 [42]. However, the main focus was always to generate synthesizable VHDL from StateCharts [30] and later from UML State Machines [2, 10, 18, 19]. In the context of UML, the Class and State Machine Diagrams




```
// File: .\fifo.cpp
#include "fifo.h"

// Operation implementations
void fifo::read(char& c)
{
//Debug No Sequence Diagram for this operation
//Debug Found Activity Diagram for this operation
    //Debug Activity Diagram Name: read
    //Debug AD Swim Lane found
    //Debug Code for Initial Node
    //Debug Code for Decision Node
    if (num_elements==0)
    {
        //Debug Code for Accept Event Node
        wait(write_event);
        //Debug Code for Action Node
        c = data(first);
        //Debug Code for Action Node
        first = (first + 1) % max;
        //Debug Code for Action Node
        num_elements--;
        //Debug Code for Send Signal Node
        read_event.notify();
        //Debug Code for Activity Final Node
    }
    else
    {
        //Debug Code for Action Node
        c = data(first);
        //Debug Code for Action Node
        first = (first + 1) % max;
        //Debug Code for Action Node
        num_elements--;
        //Debug Code for Send Signal Node
        read_event.notify();
        //Debug Code for Activity Final Node
    }
}
```

Fig. 19 Example of UML activity diagram and generated code [3, 57]

were the main diagrams used due to their importance in UML 1.x. 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 since the introduction of StateCharts.

UML and C/C++ based ESL languages Several design languages based on C/C++ (e.g., SpecC, Handel-C, ImpulseC, SystemC) have been developed in the last decade 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 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.

Tailoring UML towards SystemC in a 1-to-1 correspondence was first investigated in [7, 25, 56]. by a 1-to-1 correspondence between UML diagrams and SystemC code, as illustrated by Fig. 19. SysML may also be refined towards SystemC [58]. Several benefits

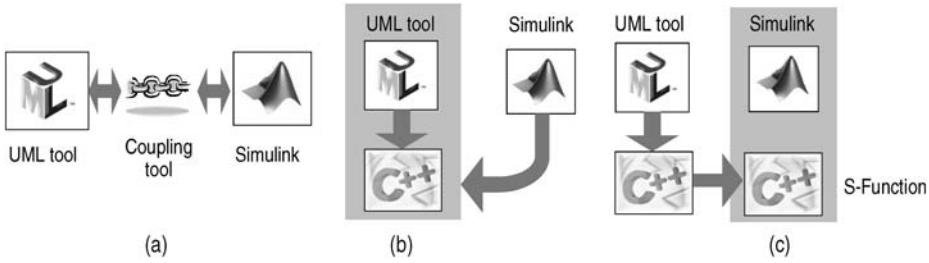


Fig. 20 UML and MATLAB/Simulink

were reported when UML/SysML is associated 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 [56]. 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 [25] have been a driving factor for the development of the UML profile for SoC (Sect. 2.3.2), and STMicroelectronics developed a proprietary UML/SystemC profile [60]. More recently, NXP and the UML tool vendor ArtisanSW collaborated in order to extend the C++ code generator of ArtisanSW so that it can generate SystemC code from UML models (Fig. 19) [3, 57]. As a side remark, Fig. 19 also illustrates that behavioral diagrams are sometimes not fully compliant to the UML standard.[3] 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 [14, 45, 79]. 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 [35, 37] or ImpulseC [77] 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 [65, 79].

UML and MATLAB/Simulink Two main approaches allow coupling the execution of UML and MATLAB/Simulink models: cosimulation, and integration based on a common underlying executable language (typically C++) [72].

In case of cosimulation (Fig. 20(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 cosimulation approach is implemented in Exite ACE from Extessy AG, which allows, e.g., coupling a Simulink model with Artisan Software Studio [67] or Telelogic Rhapsody. A similar simulation platform is proposed in [31] for IBM Rational Rose RealTime.

The alternative approach is to resort to a common execution language. In absence of tool support for MATLAB code generation from UML, the classical solution is to generate

[3] A *merge node* is indeed normally required in order to merge the two alternate flows.

C/C++ code from MATLAB, 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. 20(b)) or inside the Simulink model (Fig. 20(c)). This solution is adopted, for instance, in the Constellation framework from Real-Time Innovation, in the GeneralStore integration platform [59], or in 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 appears 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 cosimulation 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 (cf. Sect. 5). 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.

3.3 One-to-many relationship

Some UML tools, such as Mentor Graphics Bridgepoint or Kennedy Carter iUML, 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. 18(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 approach, often called *executable* (xUML) or *executable and translatable UML* (xtUML) [44], 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 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 [66] 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. Provided that synthesis tools taking as input C or C++ based SoC languages gain more popularity, xtUML tools could theoretically support flexible generation of SW and HW implementations, where the SW part of the system is produced by a model compiler optimizing the generated code for an embedded processor, while the HW part is generated targeting a behavioral synthesis tool.

4 UML and SoC design flows

4.1 Development processes

UML is a rich and complex notation that can address complex systems and help improve cross-disciplinary communication, but is neither a method nor a development process. A development process stipulates which activities should be performed by which roles during which part of the product development. A method is a particular combination of a notation and a development process.

UML can contribute to more efficient communication within the project team members *if* it is used in the context of a well-defined development process and with the support of appropriate tools. Tooling aspects will be further developed in Sect. 5.

The importance of a customized development process, which assists SoC engineers in improving their design using UML, is too often neglected. Still, the absence of a sound methodology and poor understanding of the purposes of using UML lead inevitably to failures and unrealistic expectations [9]. A well-defined process provides significant advantages:

- It structures thinking and actions, coordinates the activity of the project team, and ensures a consistent evolution of work.
- It eases the measure of progress by identifying the deliverables at crucial steps, resulting in effective means of planning. Project predictability is improved in terms of effort and calendar time. The definition of artifacts produced during the development clarifies which information is communicated to the different stakeholders and identifies project goals. These milestones prevent activities to take over as an end in its own right, a risk exacerbated in model-based design flows.
- It encourages continuous assessment of the product designed and the process itself. The quality of current and future products is improved, e.g., in terms of number/severity of defects, reusability, and flexibility to requirements changes.

Modern development processes for software [36], embedded software [20], and systems engineering [6] follow iterative frameworks such as Boehm's spiral model [13]. In disciplines such as automotive and aerospace software development, however, we can still find processes relying on sequential models like the waterfall [61] and the V-model [24], due to their support of safety standards such as IEC 61508, DIN V VDE 0801, and DO 178-B. A traditional waterfall process (Fig. 21(a)) assumes a clear separation of concerns between the tasks which are executed sequentially. Such 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. 21(b) with the example of the Rational Unified Process [36]. 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 (Fig. 22) based on Simulink, where models with increasing

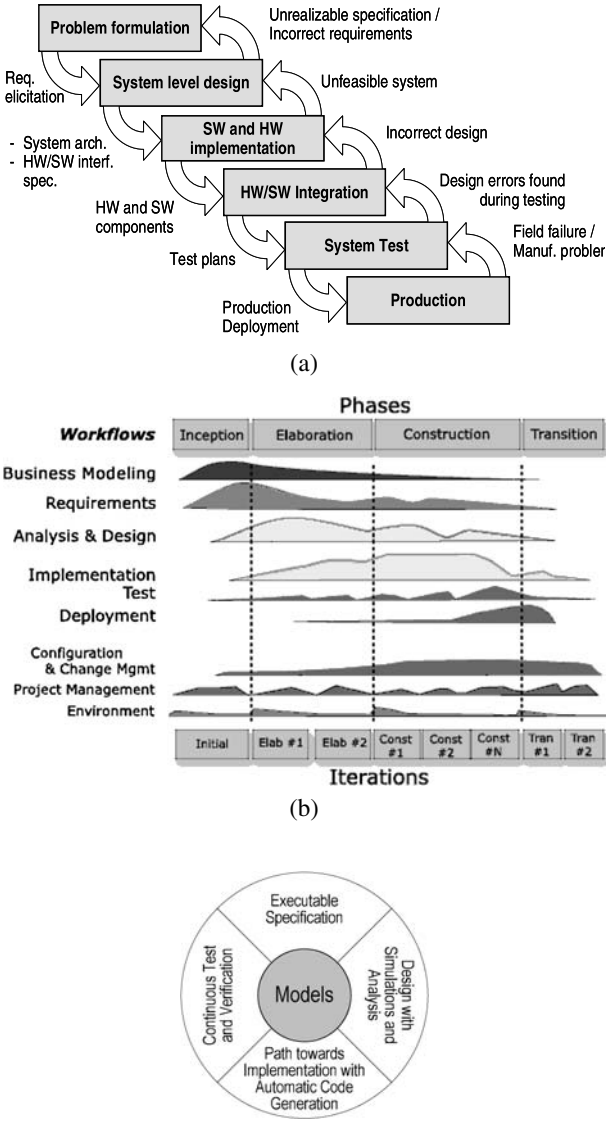


Fig. 21 Waterfall vs. iterative development processes (excerpt from [36])

Fig. 22 Model-based design flow (adapted from [66])

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.

While general iterative process frameworks constitute a valuable starting point, it is however crucial to customize these and capture the specific process flows, activities, and milestones that will be employed for a concrete project and application domain. In the SoC context, strict first time right requirements and exponentially increasing mask costs prevent successive physical (custom design) implementations. Nevertheless, executable models based on UML and ESL languages provide a means to support iterative development process customized towards SoC design, such as in Fig. 23 where the size of the circles is proportional

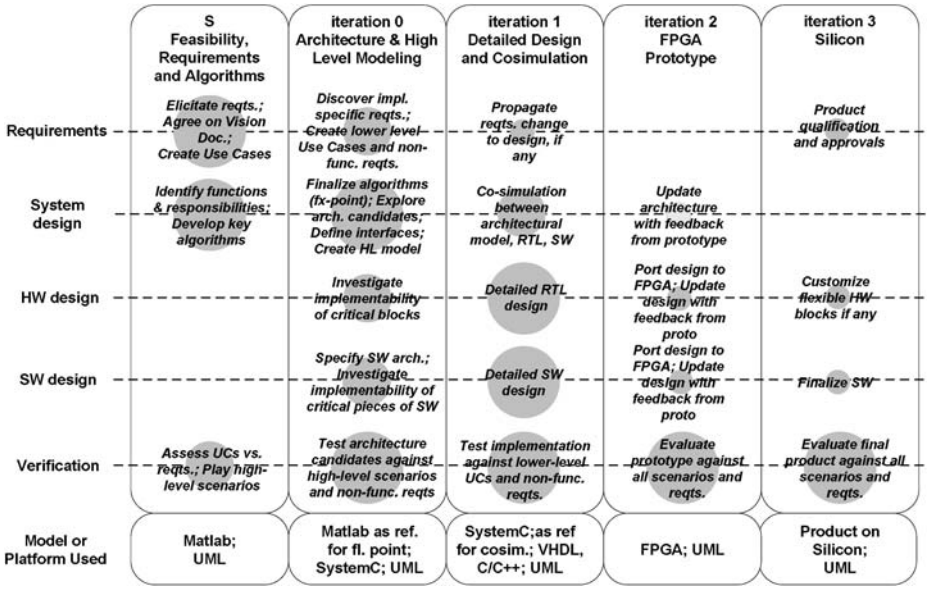


Fig. 23 Example of an iterative development process customized towards SoC design [56]

to the importance of the effort spent at each iteration.. 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 HW and SW implementation languages, and if the generated code can be further synthesized or cross-compiled.

4.2 UML in SoC development processes

UML can be used in many different ways for the design of electronics systems. First, UML can be used for specification purposes. As an example, the specification expressed in UML is integrated in [8] with the ACES hardware/software codesign platform developed at NEC, which includes high-level synthesis from C/C++ and coverification tools. The specification process with UML may start with designers sketching out few diagrams to perform an early analysis of the requirements (e.g., with the help of SysML Requirements Diagrams). In addition, they may develop use cases and scenarios in order to verify the behavior of the system and explore architectures and design alternatives. Though UML is used here mainly for documentation purposes and support for discussions, the benefits gained from a specification analysis with the help of UML should not be underestimated. For example, a survey conducted by the Japan Electronics and Information Technology Industries Association (JEITA) revealed that 65% of the reasons for redesign are related to incorrect or ambiguous specifications, above customer requests (23%) [34]. However, it has been reported [78] that if UML is used to help validate the consistency and the completeness of the SoC specification, errors can be detected and eliminated earlier. The following reasons have been identified: (1) a graphical description of the specification contributes to a better understanding and review of the functional specification of the system, and (2) UML supports the specification from different viewpoints, i.e., the functionality, data structures,

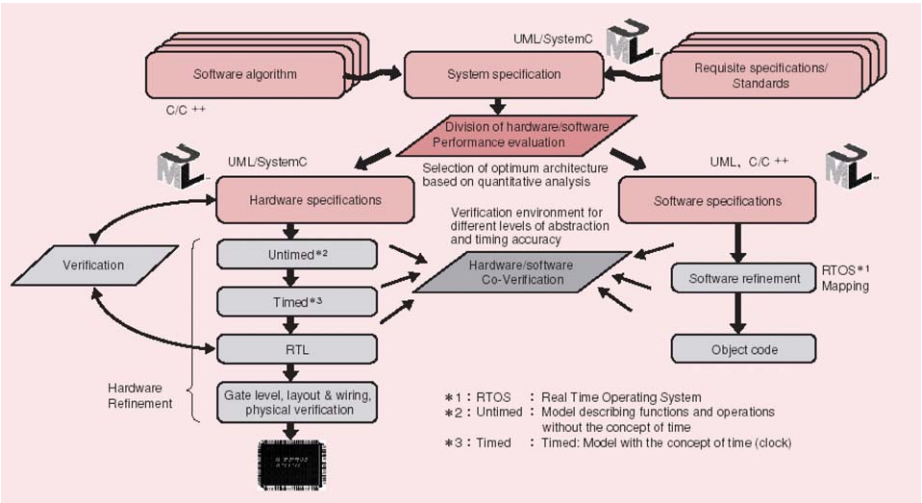


Fig. 24 Application of UML for the specification and description of electronic systems' architectures (excerpt from [25])

architecture, and behavior can be represented in a unified environment with different diagrams. However, UML only defines different types of diagrams and does not describe how to best apply them. It is crucial to have a well-defined development process to guide the use of UML, identify which diagrams should be used and for what purpose. Additionally, UML lacks of features as a specification language and may not be enough, despite its complexity. For example, UML Use Case Diagrams carry only limited information, but a use case analysis brings tangible added value by encouraging thorough thinking about the behavior of the system. Alternative scenarios, which are extensions to the main success scenario, provide one of the greatest values of use cases because they contribute to designing robust and fault resistant systems. Besides its application at specification level, UML can also be used to describe the structure and behavior of the system at architectural level (Fig. 24). A one-to-one correspondence between UML diagrams and the code corresponding to a high-level and executable model of the architecture can be established. The implementation details in such models are abstracted to a degree which depends on the purposes of the model and the simulation speed which must be achieved. The use case analysis can be performed iteratively at the level of subsystems in a similar way as it was done at top level. Subsystem use cases are derived from the top level requirements and the architecture decomposition of the system. Non-functional (performance) requirements are increasingly introduced as the system architecture is refined, and can be further captured with SysML Requirements Diagrams.

The association of UML/SysML and ESL languages provides tremendous opportunities to unify the specification, co-design, and verification tasks of HW/SW systems. As an example, Fig. 25 shows how architectural models, UML diagrams, and MATLAB can be associated and complement each other. First, the system architecture and behavior is validated via a systematic derivation of test scenarios from the specification in UML and the top-level use case analysis, where the system is seen as a black box. As a result, logical errors in the implementation can be more efficiently detected [78]. The possible paths in the use cases give rise to a collection of scenarios, which can be reused at different abstraction levels until prototype test. Additionally, use cases contribute to the principle of testing early and often.

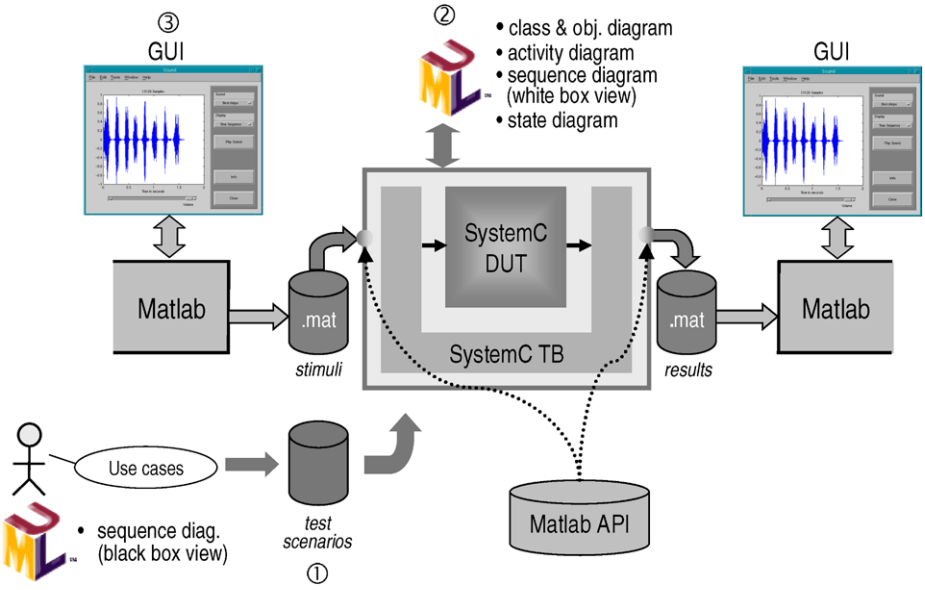


Fig. 25 Improving the traceability between specification and architecture [74]

The added value of use cases as a means for regular, systematic, and effective testing depends, however, on their rigor and formality. Second, the architecture and behavior of the system modeled in C/C++ or a derived language can be represented in UML (white box view). The code of the architecture model can be generated from UML diagrams in order to reduce the coding effort and the modeling time. Automatic code generation is a crucial feature to prevent the advantages of ESL to be wiped out by the modeling effort [26]. Finally, the lacking support for analysis and verification of the performance of the architecture in UML tools can be alleviated by a back-end support in MATLAB. This includes for example the possibility to compare the simulation outputs of signal processing intensive parts of the system with results expected from the specification of algorithms in MATLAB, or the design of graphical user interfaces (GUI) to exercise the model and display results.

Although architecture and design space exploration is mostly a manual task, research works on automatic partitioning and design space exploration from UML models have been reported [1, 64]. In [64], a tool and design flow were proposed for architectural exploration from the UML models of a system, based on the optimization of a cost function and a library of available blocks implemented in HW and in SW (Fig. 26). Following the suggested design flow, an optimized component allocation, task mapping, and scheduling is obtained. Behavioral synthesis from UML models has also been reported [5].

At lower abstraction levels (RTL), UML can still be used as a notation for documentation purposes or in close synchronization with code. Recent advances in C/C++ synthesis and the lack of commercial tool support may limit in the future the use of UML in association with RTL languages. Nevertheless, test suites identified during use case analysis still play a role for verification purposes until final product test (Fig. 23). In addition, the traceability mechanisms defined in SysML can help assess the impact of specification changes on the implementation.

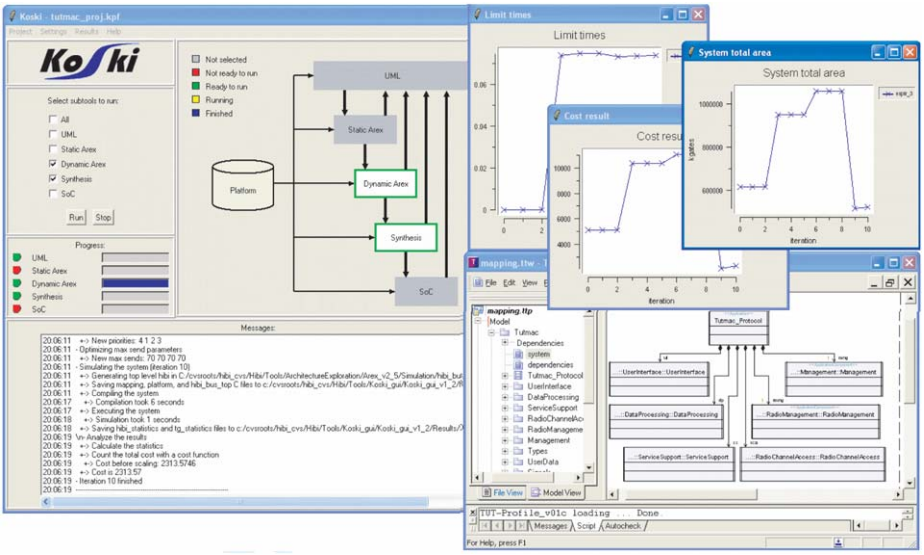


Fig. 26 Screen snapshot of the Koski tool for design flow space exploration and partitioning (excerpt from [64])

5 Tool support

Appropriate tool support is crucial for a successful adoption of UML for SoC design, but is still developing at the time of writing. As shown by the UML-SoC Workshop survey results [73], this is perceived as one of the major issues preventing UML from being widely adopted in the context of electronic systems design. Tools based on UML should ideally support an impressive list of goals, such as:

Modeling support. Tools must support all UML diagrams and help users to build rapidly correct UML models. Incorrect syntax or semantics must be detected. Tools must be aware of equivalences and relationships between modeling constructs appearing in several diagrams. Changes in a diagram must be propagated if these affect other diagrams.

Simulation support. Performance analysis of the system, debugging and step-by-step visualization of the activity of the system, export of simulation data, etc., must be flexible.

Semantics and customization. Tools must help solve the issue of lacking semantics in UML. Users must be able to define their own customization of UML, while making sure it does not violate the semantics of UML.

Abstraction levels. Users should be able to navigate the system both horizontally (along the parts of the system at a given abstraction level) and vertically (through abstraction levels).

Code and documentation generation. Code generation from UML models towards a range of SoC languages should be supported, as well as synchronization between model and code. Similarly, the generation of documentation from models must be automated.

Tool interoperability. The import/export of models from/towards other UML modeling tools, as well as the import/export of IP blocks described with metadata specifications, such as IP-XACT [63], should be supported. Tools must integrate seamlessly UML models with requirements management or office suites of tools on one hand, and SoC design tools on the other hand.

Version control. Tools should integrate seamlessly with version control mechanisms.

Collaborative modeling environment. Different teams are involved while designing complex electronic systems and may work on different subsystems in parallel. Such collaborative design effort needs to be properly synchronized by the UML tool.

This list of desired features is supported to a significantly varying degree by the available UML tools. The following paragraphs give an overview of the current situation of the UML tool market in the context of SoC design.

Modeling and analysis capabilities These vary tremendously among the tools and are more suitable for SW development rather than SoC design. The most basic tools merely allow drawing simple UML diagrams. More complex UML tools provide support for simulation and detailed modeling, but full implementation of the UML standard and compliancy can hardly be found. The majority of the tools support the most important diagrams, modeling constructs and profiles, but not necessarily all of them. The subset of diagrams implemented in most tools includes Class, Composite Structure, Object, State Machine Diagrams, Activity, Sequence, and Timing Diagrams. SysML has already gained the support of several tool vendors. The UML profile for SoC, on the other hand, is far less supported. The complexity of UML 2.x, the major changes brought to UML 1.x, and the number of existing profiles contribute to this situation.

In terms of verification and analysis, UML tools can generate code and animate behavioral diagrams while simulating the model, allowing users to quickly verify the functionality of the system. However, support for in-depth analysis of the system's performance against real-time constraints, or communication bottlenecks, for example, is lacking. Users may need to export simulation data for analysis in other environments. Furthermore, the communication model in UML is based on asynchronous events and is not adequate for HW modeling, where a real discrete event engine would be needed.

Interoperability Two aspects can be distinguished: the interoperability between UML tools, and the ease of integration between these and tools used in typical SoC design flows.

Regarding the former, UML tools typically store models in a proprietary format but allow importing/exporting models in XML Metadata Interchange (XMI) format. XMI is a text based interchange format which is standardized by OMG. However, incompatibilities between the implementations of XMI by different UML tool vendors prevent seamless model interchange. Furthermore, only relationships and modeling elements are captured in XMI, whereas most of the visual details of the created diagrams are discarded. As a result, diagrammatic layout is typically not preserved when moving to a different UML tool. However, the possibility to store UML models in XMI format provide interesting opportunities for UML tools to be compliant with specifications for IP metadata such as Spirit IP-XACT [63]. This standard allows IP blocks to be described in a specific XML format in order to facilitate the connection of IP blocks from various sources. As XMI is based on XML, IP integration in UML tools is easy by extending the import/export capability of UML tools to be compliant with IP-XACT. This subject is currently investigated by several groups. The Open SoC Design Platform for Reuse and Integration of IPs (SPRINT) project [55] is investigating the extension of IP-XACT towards ESL design as well as UML profiles for TLM and their potential for IP exchange.

In terms of the ease of integration with SoC design environments, several UML tools on the market can be integrated with popular IDEs for SW development. However, the integration solution is typically proprietary to the UML tool vendor, and users might not always find a UML tool providing integration with their preferred IDEs. As a result, customized solutions may need to be developed. In this context, the Eclipse design framework [70] can

CASE tools	IBM (Rational), Sparx Systems, Gentleware, ...
Support for RT embedded systems	Mentor, ArtisanSW, IBM (I-Logix/Telelogic), ...
Tools at the crossroad of domains	Extessy, Real-Time Innovation, ...
In-house tools/extensions for SoC design	STMicroelectronics, U. Tampere, U. Loughborough, ...
Commercial tools with support for SoC design	ArtisanSW, Axilica, CATS

Fig. 27 Examples of commercial tool vendors and research activities related to tool support

help bridge the gap between UML and IDEs for embedded systems, by virtue of its extension capability based on plug-ins. On the HW side, the integration between UML tools and HW design flows and tools is not directly supported by commercial UML tools, although several ad-hoc solutions have been investigated. So far the various efforts to generate synthesizable code from UML models have been carried out in research context, e.g., [10, 18, 19, 62, 65, 77, 79], but some have recently been commercialized [68].

UML tool market The market of Electronic System Level design tools is still moving. The definition of ESL is variable, the range of application domains wide, design flows are manifold and languages numerous. As a result of such a difficult market to grasp, support for ESL from the "big Three" EDA vendors is still lacking, with the exception of Mentor Graphics, and design houses have been allocating important resources developing their own ESL tools in-house [26]. Several high-level tool vendors have attractive solutions after years of steady development, however. Still, many of these companies are small and their offering recent, making corporate buyers reluctant to take the risk of changing their design process. In the EDA context, UML appears as a recent technology which is still being explored, despite its long application in the SW domain. Major EDA vendors have been observing its evolution in the past years but will only invest in tool support once a strong demand for it is identified and customer needs clarified. Nevertheless, some UML tool vendors are trying to expand their market share to electronic systems by extending their tools in collaboration with design houses [57].

UML tools applicable for ESL design can be classified in different categories (Fig. 27). Traditional *Computer Aided Software Engineering (CASE) tools* are design environments which are used to model primarily software systems, and support code generation towards C++, Java, or Ada, for example. IBM Rational Rose, Sparx Systems Enterprise Architect, and Gentleware Poseidon are typical examples of tools which fit in this category. A more detailed classification of this type of tools can be found in [11]. UML modeling tools with *support for real-time and embedded systems* provide modeling and code generation capabilities which suit the design of real-time embedded SW applications. Products such as Telelogic Rhapsody (now acquired by IBM), Artisan Software Studio or IBM Rational Rose RealTime, provide proprietary mechanisms to represent timeliness properties and execute UML models with timing annotations. Several UML tools, such as Telelogic Rhapsody and Tau, or the Mentor Graphics EDGE UML suite, provide support for RTOS programming interfaces like Green Hills Integrity or Mentor Graphics Nucleus, respectively. Tools *at the crossroad of domains* cover different application areas. For instance, the Extessy Exite coupling tool [23] allows co-simulating Simulink models with UML State Machine Diagrams in Telelogic Rhapsody or Artisan Software Studio. As a second example, Real-Time Innovation Constellation is a UML-based integrated platform for real-time systems which can be used for system level semiconductor design, although it is initially meant first for control and robotics. It provides a graphical environment with execution tracing and data visualization tools, generation of C++ code with support for concurrent behaviors similar

Strengths	*Weaknesses*
• While originally applicable to the SW domain, UML is now a general purpose language which can be customized towards specific application domains. Several extensions useful for SoC design are now available. • UML allows for a unified representation of the requirements, the structure and the behavior of heterogeneous electronics systems. The communication between system architects, HW and SW designers is improved and specification errors reduced as they share a common notation. • UML provides scalable modeling means which can be applied at various abstraction levels. • UML can be associated with SoC languages, allowing for automatic code generation. Coding effort is reduced and design time cut. • UML tools can provide a common and structured environment for the documentation of system specification and design information.	• UML lacks of semantics. Precise semantics is required to really make the UML a communication mean, but too much precision would prevent UML to be applied to different domains which cannot be unified under a single semantics. Users have to specify and agree on what they estimate to be sufficiently accurate semantics in their context. • UML profiles (customization) may have incompatibilities, running against the purpose of unification. • UML is rich but also complex. Users may have to determine a subset sufficient for their needs. Successful application of UML requires experience. Engineers without a multi-disciplinary background need to be introduced to the modeling concepts, development process and notations. • Tool support for SoC design, exploration, analysis and verification is lacking. Ad-hoc solutions may have to be developed.
Opportunities	*Threats*
• The growing complexity of electronic systems requires the use of concise and flexible design languages which help fill the gap between specification and implementation, and unify the representation and analysis of heterogeneous domains. • The increasing quality and number of behavioral synthesis tools enhances the popularity of C/C++ based languages as design entry point for signal processing intensive HW blocks. The association of such tools with the code generation capability of UML tools contributes to bridge the gap between specification and implementation. • IP metadata standards based on XML (e.g. Spirit Consortium) and the XMI import/export capability of UML tools share common roots. UML tools could contribute to the automation of IP integration.	• ESL point tools integrating an increasing number of features, e.g. Simulink in the context of DSP intensive applications. UML can capture cross-disciplinary aspects but may be weaker than competition when considering a single domain. • Cultural issues. Besides basic engineering functions, designers must increasingly become knowledgeable in all phases, from initial concept to manufacturing, as well as in multiple application domains. A sound development process, necessary to guide the application of UML, promotes this attitude but cultural differences still put the acceptance of multi-disciplinary and innovative technologies at risk. • Education. Lack of experience and/or absence of development process guiding the use of UML translates into both unrealistic expectation and misapplication of technology.

Fig. 28 SWOT analysis in the context of adopting UML for SoC design

to SystemC, and means to integrate C++ code generated from MATLAB/Simulink into the C++ code produced by Constellation. *In-house Tools* and *Tool Extensions* are developed by users. The tools developed at Tampere University to support the Koski design flow [64] or the ChipFryer from Loughborough University [68] belong to the former category, whereas the latter includes customizations and/or extensions of commercial tools with extra features. For example, the specification task performed using Rhapsody is followed by a behavioral synthesis tool developed by NEC in [8]. Sparx Systems Enterprise Architect is extended to support SystemC code generation from UML in [12]. The Mentor Graphics BridgePoint model compiler, which transforms UML design models into C or C++ code, allows its rule-based translation templates to be modified so that the same UML model can be used to generate code for other targets, such as VHDL [43], or C++ compliant with Mentor CatapultC behavioral synthesis tool. *Commercial tools with support for SoC design* are still limited in number. ArtisanSW supports SystemC code generation within Studio (Fig. 19) [3]. CATS [15] provides a framework which can generate a SystemC skeleton out of UML models class diagrams. Axilica has recently launched the first behavioral synthesis tool from UML towards SystemC and RTL [5], based on the ChipFryer technology.

As a conclusion, it is crucial to choose a UML tool according to a well-defined list of user-defined criteria and desired features, in order to assess the strengths and weaknesses of each solution.

6 Conclusions

Figure 28 summarizes our view on the adoption of UML and SoC design in a Strength, Weaknesses, Opportunities and Threats (SWOT) analysis. As shown in previous sections, a focus on language or notation aspects is insufficient when considering UML for electronic system level design. The application of the language must be associated with a reflection on design flow aspects, how the language should be used, and for which purposes. The adoption of UML in an organization provides therefore 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.

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). We also would like to thank Tim Schattkowsky for fruitful discussions and his comments.

References

1. Ahmed W, Myers D (2006) Faster exploration of high level design alternatives using UML for better partitions. In: Proc design, automation and test in Europe (DATE) conf
2. Akehurst D et al (2007) Compiling UML state diagrams into VHDL: an experiment in using model driven development. In: Proc forum specification & design languages (FDL)
3. Artisan SW. From UML to SystemC—model driven development for SoC. Webinar, http://www.artisansw.com
4. AUTOSAR—Automotive System Architecture http://www.autosar.org
5. Axilica FalconML. http://www.axilica.com
6. Bahill A, Gissing B (1998) Re-evaluating systems engineering concepts using systems thinking. IEEE Trans Syst Man Cybern C, 516–527
7. Baresi L et al (2003) SystemC code generation from UML models. Springer, Berlin
8. Basu AS et al (2005) A methodology for bridging the gap between UML & codesign. In: UML for SoC Design. Springer, Berlin
9. Bell A (2004) Death by UML fever. ACM Queue 2(1)
10. Björklund D, Lilius J (2002) From UML behavioral descriptions to efficient synthesizable VHDL. In: 20th IEEE NORCHIP conf
11. Blechar M (2006) Magic quadrant for OOA&D tools (2H06 to 1H07). Technical report, Gartner Research Report G00140111
12. Bocchio S, Riccobene E, Rosti A, Scandurra P (2005) A SoC design flow based on UML 2.0 and SystemC. In: Proc 2nd UML-SoC workshop at 42nd DAC conf
13. Boehm B (1988) A spiral model of software development and enhancement. IEEE Comput 21(5):61–72

14. Boudour R, Kimour M (2006) From design specification to SystemC. J Comput Sci 2:201–204
15. CATS XModelink. http://www.zipc.com/english/product/xmodelink/index.html
16. Cockburn A (2000) Writing effective use cases. Addison-Wesley, Reading
17. Cockburn A (2002) Use cases, ten years later. STQe 4(2):1–9
18. Coyle F, Thornton M (2005) From UML to HDL: a model driven architectural approach to hardware-software co-design. In: Proc information syst: new generations conf (ISNG)
19. Damasevicius R, Stuikys V (2004) Application of UML for hardware design based on design process model. In: Proc Asia and South Pacific Design Automation Conf (ASP-DAC)
20. Douglass B (2004) Real time UML. Addison-Wesley, Reading
21. dSPACE. http://www.dspace.com
22. Electronics Weekly & Celoxica (2005) Survey of system design trends. Technical report
23. Extessy. http://www.extessy.com
24. Forsberg K, Mooz H (1995) Application of the vee to incremental and evolutionary development. In: Proc 5th annual int symp national council on systems engineering
25. Fujitsu (2002) New SoC design methodology based on UML and C programming languages. FIND 20(4):3–6
26. Goering R (2005) Tools missing as ESL rolls. EE Times
27. Goering R (2006) System-level design language arrives. EE Times
28. Grell D (2003) Rad am Draht, Innovationslawine in der Autotechnik. c't 14:170–183
29. Grötker T, Liao S, Martin G, Swan S (2002) System design with systemC. Springer, Berlin
30. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
31. Hooman J et al (2004) Coupling simulink and UML models. In: Proc symp FORMS/FORMATS
32. International Council on Systems Engineering. http://www.incose.org
33. ITU-T study group 17—Languages for telecommunication systems (2002) ITU-T recommendation Z. 100: specification and description language (SDL). Technical report
34. JEITA (2000) LSI Biwako Workshop paper. JEITA System Level Design Study Group
35. Katayama T (2005) Extraction of transformation rules from UML diagrams to SpecC. IEICE Trans Inf Syst 88(6):1126–1133
36. Kruchten P (2003) The rational unified process: an introduction. Addison-Wesley, Reading
37. Kumaraswamy A, Mulvaney D (2005) A Novel EDA flow for SoC designs based on specification capture. In: Proc ESC division mini-conference
38. Laemmermann S et al (2006) Automatic generation of verification properties for SoC design from SysML diagrams. In: Proc 3rd UML-SoC workshop at 44th DAC conf
39. Lilly S (1999) Use case pitfalls: top 10 problems from real projects using use cases. In: Proc technology of object-oriented languages and systems (TOOLS), pp 174–183
40. Martin G, Mueller W (eds) (2005) UML for SoC design. Springer, Berlin
41. McGrath D (2005) Unified modeling language gaining traction for SoC design. EE Times
42. McUmber W, Cheng B (1999) UML-based analysis of embedded systems using a mapping to VHDL. In: Proc 4th IEEE int symp high-assurance systems engineering
43. Mellor S (2005) Keynote talk: the gap between specification and synthesis. In: Proc forum on specification and design languages (FDL)
44. Mellor S, Balcer M (2002) Executable UML: a foundation for model-driven architecture. Addison-Wesley, Reading
45. Nguyen K et al (2004) Model-driven SoC design via executable UML to SystemC. In: Proc 25th IEEE int. real-time systems symposium (RTSS)
46. OMG. OMG systems modeling language specification
47. OMG (2004) UML 2.0 testing profile specification v2.0
48. OMG (2004) UML profile for modeling QoS and fault tolerance characteristics and mechanisms
49. OMG (2005) UML profile for schedulability, performance, and time (SPT) specification v1.1
50. OMG (2006) Object constraint language specification v2.0
51. OMG (2006) UML profile for system on a chip (SoC) specification v1.0.1
52. OMG (2007) A UML profile for MARTE Beta 1
53. OMG (2007) UML v2.1.1 infrastructure specification
54. OMG (2007) UML v2.1.1 superstructure specification
55. Open SoC Design Platform for Reuse and Integration of IPs (SPRINT). Project http://www.sprint-project.net
56. Pauwels M et al (2003) A design methodology for the development of a complex system-on-chip using UML and executable system models. In: Villar E, Mermet J (eds) System specification and design languages. Springer, Berlin, pp 129–141
57. Ramanan M (2006) SoC, UML and MDA—an investigation. In: Proc 3rd UML-SoC workshop at 43rd DAC conf

58. Raslan W et al (2007) Mapping SysML to SystemC. In: Proc forum spec design lang (FDL)
59. Reichmann C, Gebauer D, Müller-Glaser K (2004) Model level coupling of heterogeneous embedded systems. In: Proc 2nd RTAS workshop on model-driven embedded systems
60. Riccobene E, Rosti A, Scandurra P (2004) Improving SoC design flow by means of MDA and UML profiles. In: Proc 3rd workshop in software model engineering
61. Royce W (1970) Managing the development of large software systems: concepts and techniques. In: Proc of IEEE WESCON
62. Shen X-Y, Chen J (2006) A homomorphic mapping based algorithm to generate synthesizable verilog from UML. Comput Sci 33(4):247–249
63. Spirit Consortium. http://www.spiritconsortium.org
64. Kangas T et al (2006) UML-based multiprocessor SoC design framework. ACM Trans Embed Comput Syst (TECS) 5(2):281–320
65. Tan W, Thiagarajan P, Wong W, Zhu Y (2004) Synthesizable SystemC code from UML models. In: Proc 1st UML for SoC workshop at 41st DAC conf
66. The Mathworks (2007) Model-based design for embedded signal processing with simulink
67. Thompson H et al (2004) A flexible environment for rapid prototyping and analysis of distributed real-tme safety-critical systems. In: Proc ARTISAN real-time users conf
68. Thomson R, Chouliaras V, Mulvaney D (2007) From UML to structural hardware designs. In: Proc 4th UML-SoC workshop at 44th DAC conf
69. UML-SoC Workshop Website. http://www.c-lab.de/uml-soc
70. UML2 Eclipse Project. http://www.eclipse.org
71. Vanderperren Y (2005) Keynote talk: SysML and systems engineering applied to UML-based SoC design. In: Proc 2nd UML-SoC workshop at 42nd DAC conf
72. Vanderperren Y, Dehaene W (2006) From UML/SysML to Matlab/Simulink: current state and future perspectives. In: Proc design automation and test in Europe (DATE) conf
73. Vanderperren Y, Wolfe J (2006) UML-SoC design survey. Available at http://www.c-lab.de/uml-soc
74. Vanderperren Y, Pauwels M, Dehaene W, Berna A, Özdemir F (2003) A SystemC based system-on-chip modelling and design methodology. In: SystemC: methodologies and applications. Springer, Berlin, pp 1–27
75. Vanderperren Y, Wolfe J, Douglass BP (2007) UML-SoC design survey. Available at http://www.c-lab.de/uml-soc
76. Viehl A et al (2006) Formal performance analysis and simulation of UML/SysML models for ESL design. In: Proc design, automation and test in Europe (DATE) conf
77. Wu YF, Xu Y (2007) Model-driven SoC/SoPC design via UML to impulse C. In: Proc 4th UML-SoC design workshop at 44th DAC conf
78. Zhu Q, Oishi R, Hasegawa T, Nakata T (2005) Integrating UML into SoC design process. In: Proc design, automation and test in Europe (DATE) conf
79. Zhu Y et al (2005) 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 & applic (RTCSA)