Advances in Design and Specification Languages for SoCs

Edited by
Pierre Boulet

The ChDL series

Springer

ADVANCES IN DESIGN AND SPECIFICATION LANGUAGES FOR SoCs

Advances in Design and Specification Languages for SoCs

Selected Contributions from FDL'04

Edited by

Pierre Boulet, *Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, France*

Springer

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 0-387-26149-4 (HB)
ISBN-13 978-0-387-26149-2 (HB)
ISBN-10 0-387-26151-6 (e-book)
ISBN-13 978-0-387-26151-5 (e-book)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springeronline.com

Printed on acid-free paper

Printed in the Netherlands.

Contents

Preface

This book is the sixth in the ChDL (Chip Design Languages) series. Year 2004 has seen many efforts in the field of electronic and mixed technology circuit design languages. The industry has recognized the need for system level design as a way to enable the design of the next generation of embedded systems. This is demonstrated by the "ESL Now!" campaign that many companies are promoting. This year has also seen many interesting standardization efforts for system level design, such as SystemC TLM (`http://www.systemc.org/`) for transactional level modeling with SystemC, AUTOSAR (`http://www.autosar.org/`) for automotive embedded system applications, or SPIRIT (`http://www.spiritconsortium.org/`) for IP interchange. In the field of modeling languages, the Model Driven Architecture of the OMG (`http://www.omg.org/mda/`) has given rise to model driven engineering, which is a more general way of software engineering based on model transformations. As embedded systems are more and more programmable and as the design abstraction level rises, model driven methodologies are also considered for electronic system level design. In this context, the OMG has recently published a call for propositions for a UML 2.0 profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE).

The constraints on the design process of these next generation embedded systems are considerable: Real-time, power consumption, complexity, mixed technology integration, correctness, time to market, cost, ..., all contribute to the now famous "design gap". The existing tools are pushed to their limits when designing complex systems-on-chip (SoCs) and reuse has become one of the major ways to fill the gap.

In this very exciting moment in the field of electronic system design languages, the Forum on Specification and Design Languages (FDL'04) has been once again the main European event for this community. This book is a collection of the best papers from FDL'04 selected by the program chairs, Alain Vachoux, Piet van der Putten, Eugenio Villar and Wolfgang Müller.

This book is structured in four parts:

- Part I, Analog and Mixed-Signal Systems, presents five chapters covering issues in mixed-signal modeling.
- Part II, UML-Based System Specification and Design, is composed of five chapters with emphasis on model transformation approaches to system modeling.
- Part III, C/C++-Based System Design, is also structured as five chapters with SystemC as its main topic.
- Part IV, Invited Contributions, concludes the book with two invited chapters presenting the important topic of system verification, and the AUTOSAR initiative.

Together, the 17 chapters of this book present recent research advances in design and specification languages for SoCs. I hope that this book will be a thought provoking read to researchers, students and practitioners in the field of languages for electronic system design.

PIERRE BOULET
General Chair of FDL'04
Université des Sciences et Technologies de Lille
Lille, France, April 2005

Previous books

Christoph Grimm (Editor), "Languages for System Specification", Kluwer, 2004.

Eugenio Villar & Jean Mermet (Editors), "System Specification & Design Languages", Kluwer, 2003.

Anne Mignotte, Eugenio Villar & Lynn Horobin (Editors), "System on Chip Design Languages", Kluwer, 2002.

Jean Mermet (Editor), "Electronic Chips & Systems Design Languages", Kluwer, 2001.

Jean Mermet, Peter Ashenden and Ralf Seepold (Editors), "System-on-Chip Methodologies ans Design Languages", Kluwer, 2001.

I

ANALOG AND MIXED-SIGNAL SYSTEMS

Introduction

Alain Vachoux

Microelectronic Systems Laboratory
Swiss Federal Institute of Technology Lausanne, Switzerland
alain.vachoux@epfl.ch

This part includes a selection of five papers that have been presented in the AMS workshop of the FDL'04 conference. The papers have been revised to achieve book level quality and provide a good coverage of up-to-date mixed-signal modeling issues.

The first paper, "Refinement of Mixed-Signal Systems: Between HEAVEN and HELL", from Christoph Grimm et al., presents a mixed-signal design framework supporting the modeling of signal processing systems through a consistent refinement process from abstract descriptions (executable specifications) to pin-accurate models. The framework is based on a prototype extension of SystemC that supports the modeling and the simulation of mixed-signal systems. It uses the object-oriented capabilities of the language to provide so-called polymorphic signals, i.e. signals that may have different semantic interpretations depending on the level of abstraction or the model of computation considered (e.g., single-rate or multi-rate dataflow, continuous-time signal flow, discrete-event). The major benefits of the approach are twofold: artificial converters are no more required to couple modules in the modeled system and the complexity of coupling different models of computations or simulation kernels can be hidden from the modeler. The paper illustrates the approach with a case study from the automotive domain.

The second paper, "Mixed Nets, Conversion Models, and VHDL-AMS", from John Shields and Ernst Christen, addresses a similar issue but deliberately limited to the use of the VHDL-AMS language when it comes to develop mixed-signal structural models. The current definition of the VHDL-AMS language does not provide specific language elements and semantics to efficiently describe hierarchical structures involving components from different signal domains (i.e. continuous-time and event-driven), although it does provide all the elements to describe converter components and conversion behavior. The paper discusses possible modeling strategies and details one proposal that could be eventually formally integrated in the VHDL-AMS language definition. Interestingly enough, the proposal defines a new object, called a wire, that has some similarities with the polymorphic signals discussed in the first paper, essentially since there is a separation between module's behavior and communication.

The third paper, "Monte Carlo Simulation Using VHDL-AMS", from Ekkehart-Peter Wagner and Joachim Haase, proposes the development of VHDL-AMS packages to support statistical simulation of VHDL-AMS models. Re-

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 3–4.

quirements to support statistical modeling and simulation have been defined during the VHDL-AMS language definition phase but they appeared to not require the support of specific language elements. This paper presents a first implementation of some of these requirements and discusses issues related to the support of different statistical continuous-time and discrete distributions as well as correlations between statistical values. Some remaining open issues have to be addressed at the simulation tool level (e.g. multi simulation runs, post-processing capabilities), but a solid foundation can be built by providing specific VHDL-AMS declarations and subprograms. Such a proposal could even be developed further to culminate as a new VHDL-AMS companion standard.

The fourth paper, "Prediction of Conducted-Mode Emission of Complex IC's", from Anne-Marie Trullemans-Anckaert et al., presents a top-down design methodology that allows for taking into account physical effects due to the distribution of power in integrated circuits in order to meet electromagnetic compatibility (EMC) compliance as early as possible in the design phase. The approach uses behavioral VHDL-AMS models to allow full-chip simulations (including IOs) in a reasonable amount of time by abstracting the real behavior, but still allowing the modeling of physical effects such as current distribution and current spike density. The paper validates the methodology through its application to the design of an 8-bit microcontroller. One interesting outcome of the approach is the definition of modeling guidelines for developing IP blocks that may be included in a library and reused in many designs without the need to use detailed transistor-level netlists and to perform long electrical simulations.

The fifth and last paper in this part, "Practical Case Example of Inertial MEMS Modeling with VHDL-AMS", from Elena Martin et al., shows how to use a model-based top-down design methodology to design complex integrated systems including non-electrical parts such as sensors or actuators. Similarly to the previous paper, this paper discusses the modeling of physical effects, here mechanical and thermal effects, in abstract behavioral models of a microsystem and its associated electrical front-end and interface with a central processing unit. The use of a mixed-signal multi-domain hardware description language such as VHDL-AMS allows for obtaining a consistent model of the complete system, evaluating the influence of physical effects and adding proper compensation. The paper raises the need to develop model libraries which include multi-domain parameterized component models at various abstraction levels which can be characterizable from physical realizations.

It is my hope that this short introduction will incite you to go through the details of these five very interesting papers and to keep an attentive eye, or even contribute, to future editions of the AMS workshop in the FDL conference.

Chapter 1

REFINEMENT OF MIXED-SIGNAL SYSTEMS: BETWEEN HEAVEN AND HELL[1]

Christoph Grimm, Rüdiger Schroll, Klaus Waldschmidt
University of Frankfurt, Professur Technische Informatik

Abstract Very complex system are designed by stepwise refinement. This means that an abstract model is successively augmented with components and properties of an implementation. For signal processing, mixed-signal systems the refinement from block diagram level to analog or digital circuit requires a significant modeling effort: The means for description of abstract models (e.g. synchronous dataflow) and physical implementation (e.g. networks) are different and cannot be combined in a direct way. In this chapter, we introduce polymorphic signals which solve this problem, and give an overview of a framework for the refinement of Mixed-Signal Systems: HEAVEN/HELL.

Keywords: Mixed-Signal Systems, Design Methodology, Refinement.

1. Introduction

A key issue of system design is the analysis of different architectures. Especially for signal processing systems the 'design space' is often huge. Design issues such as partitioning (analog, digital ASIC, DSP+Software), determination of sample frequencies, bit widths, or precision of analog components determine quality, performance and costs of the system. In order to analyze and to verify the behavior of different architectures, models of each architecture are created and simulated. Unfortunately, this is a time consuming issue.

In *model based design*, the availability of many different modeling platforms (models of computation, MoCs) simplify the creation of models. A model of computation can be seen as an 'abstract processor' that can be programmed by a 'language', and that defines means for communication and synchronisation.

[1]This work has been partially supported by funds of the BMBF/edacentrum Project SAMS under Reference Number 01M3070D.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 5–20.

Examples for MoCs are finite state machines which are 'programmed' by states and transitions, or electrical networks that are programmed by components that are connected with nodes. Particular research on the use of different MoCs has been done within the design framework Ptolemy [Lee et al., 2003]. Although model based design supports the creation of new models, it does not support the re-use and modification of models within a design process.

The idea of *refinement* is to support the interactive, stepwise design process that leads from an abstract, executable specification to an implementation by

- integrating and modifying existing components, and
- augmenting the abstract, executable specification with new properties.

Figure 1.1 gives an overview of the design framework HEAVEN (**HE**terogeneous Systems Refinement, **A**nalysis and **V**erification **EN**vironment) that is used to demonstrate refinement of block diagram to a mixed-signal system, and which allows evalutation of different partitioning, bit widths, sample frequencies, noise and nonlinearities of analog circuits, etc. HEAVEN is supported by HELL (**HE**terogeneous Systems mode**l**ing **L**ibrary; figure 1.1 right). HELL provides behavioral models of physical effects that can be added to the 'ideal' behavior assumed in HEAVEN.

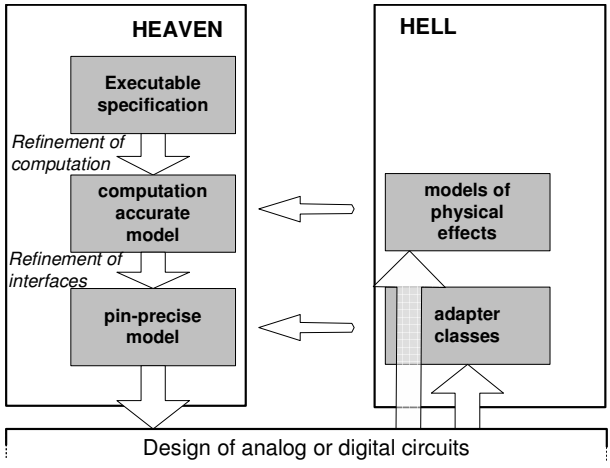


Figure 1.1. Refinement of signal processing applications to mixed-signal circuits with HEAVEN/HELL.

In HEAVEN/HELL, refinement of signal processing systems is enabled by polymorphic signals and adapter classes. Polymorphic signals automatically translate communication mechanisms used in the MoCs applied for modeling different realizations at different levels of abstraction. Therefore, polymorphic

signals allow designers to compose models in an intuitive and interactive way, just by connecting blocks.

Related work. The use of polymorphism for modeling heterogeneous systems is not new. Basic ideas for polymorphic models and signals in HEAVEN are also found in hybrid data-flow graphs [Grimm et al., 1996, Grimm et al., 1998], where signal types are converted implicitly, and the semantics of nodes is defined by firing rules. An implicit conversion of different signal types is also provided by Matlab/Simulink. However, Matlab/Simulink is restricted to block diagrams with discrete or continuous signals, and does not support modeling of analog or digital circuits or software.

SystemC 2.0 introduces a very generic approach, where signals are accessed via interfaces which can be realized in different ways. This allows one to introduce different models of computation by using different implementations of the interface [Swan, 2001]. However, the type checking between the interfaces is very strict, and in order to combine different models of computation, one has to use converter modules, for example.

In Ptolemy II/Chess, behavioral types [Lee et al., 2004] provide basically the same functionality as the signal interfaces in SystemC 2.0. In extension to SystemC 2.0, interface automata permit the coupling of different models of computation provided there is a subtype relation between them [Lee et al., 2004]. Note that the subtype relation applies to the value types, and to the protocols that implement a model of computation ('behavioral types'). Unfortunately, a subtype relation often does not exist, or can even be misleading because semantic is not considered.

In [Grimm, 2003], we introduce polymorphic signals for signal processing systems. Polymorphic signals provide methods that translate communication in different MoCs and at different levels of abstraction. However, the implementation described in [Grimm, 2003] is restricted to discrete event (DE) and static dataflow (SDF) MoCs. In the following we introduce application specific semantic types and a polymorphic signal class, that cover the MoCs used in signal processing applications at different levels of abstraction. Compared to Ptolemy, polymorphic signals are application specific, and are not a generic modeling property. The restriction to a domain of applications such as signal processing applications has the advantage, that we can assume that all (polymorphic) signals are approximations of an 'ideal', continuous-time signal. This gives all conversions an intuitive understanding.

Section 2 gives a rough overview of SystemC-AMS, describes the refinement methodology, and introduces general requirements of polymorphic signals for the refinement of signal processing systems. Section 3 describes a polymorphic signal class for the modeling of signal processing systems. Section 4 describes the application of polymorphic signals in a case study.

2. Refinement with HEAVEN

HEAVEN is built on top of SystemC and SystemC-AMS [Einwich et al.,2003, Vachoux et al., 2004]. SystemC-AMS, resp. an early prototype, the ASC-Library [Grimm, 2003], permits the modeling and simulation of signal processing systems. In the following we first give a brief description of SystemC-AMS. Then, we describe the refinement of signal processing applications to different mixed-signal architectures, and motivate polymorphic signals which support such a refinement.

2.1 SystemC-AMS

Layered approach. In SystemC 2.0 systems are specified by a structure of *modules*. The modules are connected by directed *signals*. Modules access signals via an interface, which is accessed via *ports*. SystemC-AMS provides means for the modeling of signal processing systems in SystemC. SystemC-AMS extensions are structured in three layers[Einwich et al.,2003]:

The *view layer* allows the designer the specification of behavior in different models of computation such as transfer functions, netlists, or a cluster of signal processing functions in the static dataflow MoC.

The *solver (simulator) layer* provides means which execute a specification given at the view layer, e.g. a coordinator which implements the static dataflow MoC, or which solves linear and non-linear differential equations.

The *synchronization layer* couples different solvers (simulators). For coupling different simulators, the static/synchronous data-flow (SDF) model of computation is used. Note, that both synchronization layer and solver layer introduce an underlying model of computation, but with different aims and requirements: The synchronization layer couples simulators which might also be external simulators such as SPICE. The solver (simulator) layer provides different means for the modeling and simulation of signal processing systems in SystemC.

Coordinator-Interface. Instances at the view layer have a unique interface (coordinator-interface). This interface allows the coordinator to control the execution of these objects, as well as their communication and synchronization.

Figure 1.2 gives an overview of a SystemC-AMS model which consists of discrete-event processes (left), and a cluster in SDF model of computation (modules 1-3, right). Before simulation starts, the SDF coordinator schedules the blocks of the SDF cluster. During simulation, the SDF coordinator executes the modules for each time step. In order to control execution of each module, the coordinator has access to all modules via the coordinator interface. Different simulators (here: SDF coordinator, and SystemC 2.0) are coupled via static data-flow MoC at the synchronization layer.

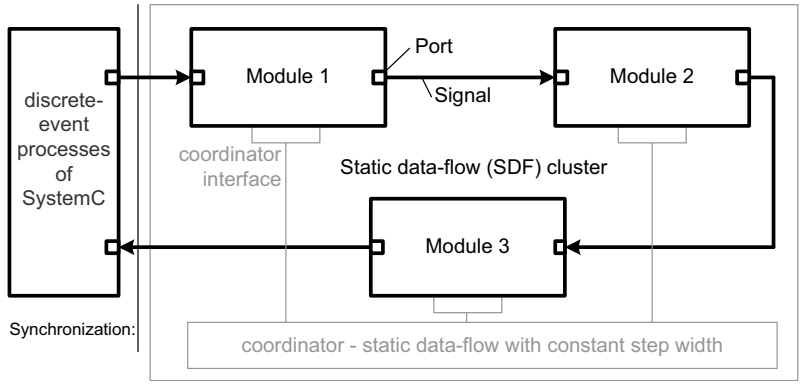


Figure 1.2. A model in SystemC-AMS.

2.2 Refinement of Computation in Signal Processing Systems

Executable specification. The design of signal processing systems begins with a block diagram which describes basic principles of the system. Sample frequencies, range of values, and bit widths are not yet known. For the first simulations, designers use the MoC 'continuous-time (CT) signal flow'. This model of computation assumes that all connections between modules have the semantics of mathematical equations, and that there is no order of execution or width of time steps that comes with this model of computation. Furthermore, signals have no limitation, and no quantization. Table 1.1 gives an overview of the modeling properties used in the executable specification.

Executable specification		
Signal	value type	No limitation
		No quantization
	sampling	No sampling
MoC	Continuous-time signal flow	

Table 1.1. Executable specification: model of computation and signals.

Note, that for simulation of the CT signal-flow model of computation, a discrete algorithm is required. This algorithm solves the mathematical equations, and determines the width of time steps. The time steps introduce an error. This error can be reduced by reducing the time steps, depending on the estimated error. Then, all blocks are simulated in the signal flow's direction. Cyclic dependencies can be broken by insertion of a delay. This is basically the static dataflow MoC, where the execution of the blocks is controlled by the estimated error.

Computation-accurate model. One aspect of system design is the evaluation of deviations introduced by different realizations. For signal processing systems, there are realizations with fundamentally different behavior: digital signal processing using a DSP or an ASIC, and analog realization. We can easily model the behavior of these implementations by replacing the MoC and/or the signal types of the executable specification by more appropriate ones, which implicitly include properties of a realization.

Properties of digital signal processing systems that have to be evaluated are sampling frequencies f_s, quantization steps Q, and range of values (limitation). A useful MoC for modeling digital signal processing is static dataflow with constant time steps $1/f_s$. Appropriate signal values are integers that model Q and the realization's range of values.

Properties of analog circuits that are modeled are limitation, limited band width and precision. Table 1.2 gives an overview of the modeling properties used in a computation accurate model.

DSP behavior		
Signal type	value type	Limitation $[lb, ub]$
		Quantization Q
	time steps	t_s
MoC	Static dataflow with constant time steps t_s	
Analog behavior		
Signal type	value type	Limitation $[lb, ub]$
		Precision S/N
	(time steps)	min. time step $t_{s,min}$
MoC	Continuous-time signal-flow	
	(simulated by static dataflow with adaptive time steps)	

Table 1.2. Analog and DSP behavioral model: models of computations and signal types.

The refinement of computation successively replaces modules of the executable specification by modules that model the implementation. The modules that model the implementation use a maybe different model of computation and a different signal type in order to model properties of the implementation. This especially affects the interaction with other blocks via ports, and may introduce incompatibilities or inconsistencies. Potential changes affect

Value types: Range and quantization/precision are restricted, e.g. from general 'real' to an 'integer' representation with limitation.

Interfaces: Changing the model of computation requires use of other interfaces and different protocols for the transport of data.

Semantics: Data is not only transported, but also might have different meanings, e.g. a sequence of bank account numbers, or an approximation of an continuous-time signal, or even nodes with Kirchhoff laws.

Because of these changes, the intuitive composition of a new model by just replacing a module by a more detailed one will not work: In most cases the resulting models are inconsistent, and require further actions to convert signal types, protocols, and semantics. Figure 1.3 gives an example for such inconsistencies due to refinements: The left two blocks have been replaced by models of a DSP implementation, and the right block models the continuous-time environment. Furthermore, the value type 'real' has been replaced by 8-bit numbers modeling the range of values from 0 to 255 ($Q = 1$) with limitation. Note, that the semantics of the signals remains unchanged: the signals are approximations of a continuous-time and continuous-value signal.

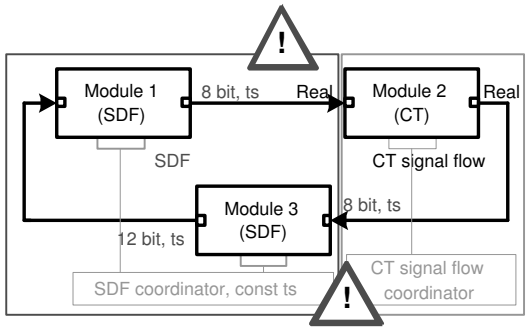


Figure 1.3. Computation-accurate model with inconsistencies due to refinement steps.

Polymorphic signals. Of course, designers can manually write converters that adapt value types, interfaces and consider changing semantics. In a limited range, this can be done automatically considering that range of values (and interfaces) are compatible with subtypes. Behavioral types in Ptolemy II convert protocols by construction of a common automata, but without considering the meaning of signals. This extends compatibility, but does not consider semantics of the data transported via a signal. Semantic issues can only be treated in an implicit or automatic way, if we know the semantics. In order to allow us to convert signals in a more general way, we assume that signals have *semantic types*. A semantic type of a signal is an abstract interpretation of a signal. In the following, we consider signal processing applications. The knowledge of the semantics allows us to formulate different views of one signal, that depends on the signature (interface, value type) used to access the signal. We call such signals *polymorphic signals* (for a domain of applications).

2.3 Refinement of Interfaces

A second aspect of system design is the explicit realization of the communication/synchronization which is implicitly specified by a model of computation. This can be done by the refinement of interfaces. The refinement of interfaces transforms the computation-accurate model to a model that has all ports of the implementation.

In digital ASICs, communication is realized by a clock signal and a controller that uses enable signals to control the communication and synchronization of the single modules. For specification of the ASIC itself at the register transfer level, the discrete event model of computation is used. Note, that there are already approaches for the refinement of communication, such as SystemC^{SV} [Siegmund et al., 2001], and the Master-Slave Library. However, they do not consider the fact that – at least in the ASC library and SystemC-AMS – the execution of the modules is controlled via a coordinator interface.

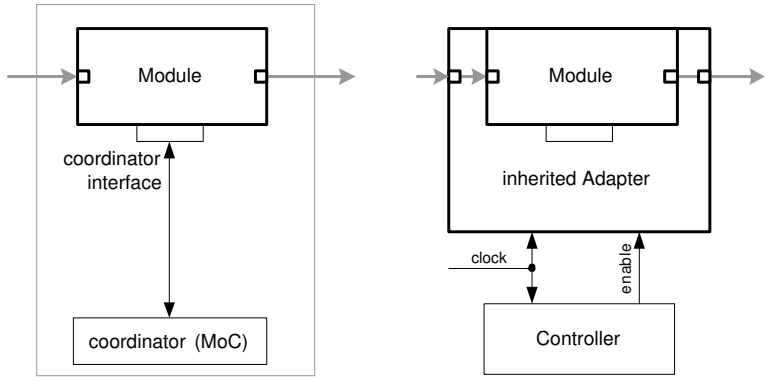


Figure 1.4. Refinement of interfaces by an inherited adapter class that uses the coordinator interface.

Figure 1.4 shows the refinement of interfaces. A module inherits an 'adapter class'. The adapter class translates clock and enable signals to an activation of the module via its coordinator interface.

3. Polymorphic signals for signal processing applications

In the following, we describe a polymorphic signal for signal processing applications. In signal processing applications, signals are more or less good approximations of continuous-time signals. Such systems are specified with the following models of computation, depending on the level of abstraction, and the implementation:

- Continuous-time signal flow (simulated by static dataflow with adaptive time steps).
- Static dataflow with constant time steps, but very often with different data rates resp. time steps (multi-rate systems).
- Discrete event system.
- Netlists.

As motivated in section 2, the refinement of signal processing systems is characterized by modules with different sample rates, ranges of values, etc. The polymorphic signal for signal processing applications supports this refinement by providing the following functionality:

- It implicitely adapts the range of values: The range of values of the writing port is adapted to the range of values of the reading port.
- It implicitely converts sample rates: The signal can have different samples rates for writing and reading ports.
- It implicitely converts different models of computation: The polymorphic signal can be read or written from the above mentioned models of computation.
- The polymorphic signal provides means for specification (or modeling) of noise and deviations for semi-symbolic analysis in HELL [Grimm et al., 2004].

Polymorphic signals can actually be used to couple modules in the supported models of computation without requiring the insertion of additional converters. In case that analog netlists are coupled, the polymorphic channel might even hide the complexity of simulator coupling — a designer just sees the channel in SystemC-AMS, and an additional node in the analog simulator.

Implementation. In SystemC, signals are accessed via ports with an interface that specifies a set of methods. Modules call these methods. A signal that is connected to a port must provide concrete methods that implement the methods called from the modules via the ports.

For each model of computation a port class is provided, for example `asc_sdf_const_in` for static dataflow with constant time steps. At the ports of the modules, attributes are specified that give additional information about the semantic interpretation of the signal to be accessed, such as:

- `value_unit` and `value_size` can specify a physical size that is associated with the abstract value at port.

- Boundaries for the range of values $[lb, ub]$.
- Time steps and rates of static (multi-rate) data flow.
- `max_deviation` and `max_noise` can specify allowed deviations of signals (for use in HELL [Grimm et al., 2004]).

If there are inequalities or incompatibilities between the ports that access a polymorphic signal, a conversion has to occur in order to permit a simulation. By default, virtual methods are called. These methods give a warning, and call simple conversion methods. The conversion methods can be overloaded by more appropriate ones, if needed.

`value_unit` and `value_size` of the reading port are compared with the writing port.

The range of values is checked and converted as follows: Let lb_{write} and ub_{write} be the lower and upper bounds of the writing port and lb_{read} and ub_{read} the lower and upper bound of the reading port. If the bounds are not equal, there might be a problem in the design; therefore, a warning is given. Then, by default, the polymorphic signal maps a written value $v_i \in [lb_{write}, ub_{write}]$ from the range of values of the writing port to a value $v_{i,read}$ from the range of values of the reading port $[lb_{read}, ub_{read}]$ as follows:

$$v_{i,read} = v_i * mult - lb_{write} * mult + lb_{read} \text{ with: } mult = \frac{(ub_{read} - lb_{read})}{(ub_{write} - lb_{write})}$$

The polymorphic signal can be written/read from ports of different models of computation. We use the following approach: The polymorphic signal inherits and implements the interfaces of all port types that are compatible with the signal as shown in figure 1.5. The methods that implement the interfaces translate read- or write- accesses into an internal, abstract representation.

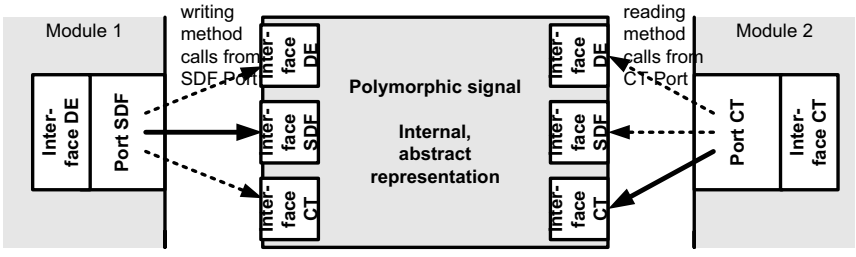


Figure 1.5. Implementation of polymorphic signals in SystemC-AMS.

In the internal, abstract representation, signals are represented by a queue of tuples $(value, time)$ ('events', 'samples'). An event (v_i, t_i) describes the point

of time t_i in sc_time and v_i is the into double converted value that was written at point of time t_i.

The size of the queue n is bounded and determined as follows:

$$n = \frac{t_{s,max} * k}{t_{s,min}}$$

where $t_{s,max}$ is the maximum possible time step, k the factor of multi-rate dataflow, and $t_{s,min}$ is the smallest possible time step.

If there are more than n events in the queue the oldest event leaves the queue. Writing processes from all models of computation are converted to this abstract representation of a signal. Then, the queue of n events $(value, time)$ describes the development of a signal in a time frame that covers at least the $t_{s,max}$ of the past.

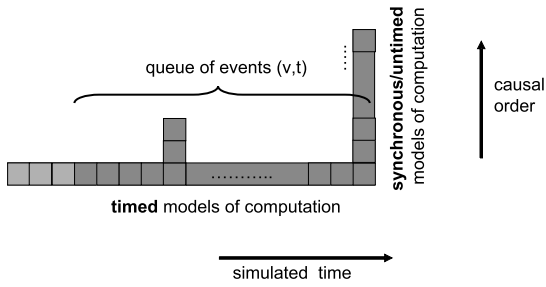


Figure 1.6. Abstract representation in polymorphic signal class by a queue of events $(value, time)$.

In general, a queue of a signal is given by:

$$(v_i, t_i), (v_{i+1}, t_{i+1}), (v_{i+2}, t_{i+2}), ..., (v_j, t_j)$$

Figure 1.6 shows an example of a queue. In the following, we describe the methods that modify (read/write) the abstract representation of signals. Synchronization and simulator coupling are implemented in SystemC-AMS resp. the ASC library (see [Grimm, 2003, Vachoux et al., 2004]). This synchronization first executes the modules of the AMS extensions using the last values from the discrete event simulation, and then executes the discrete event simulator.

The following methods are used for writing the queue; the size of the queue is limited, and adding a newer element automatically removes the oldest element:

Writing from SDF port: An event is added to the queue. For multi rate dataflow, several events can be added at once.

Writing from DE port: The value of the event to be written is compared with the last event's value. If the values are different, `request_update()` is called and triggers an event in the DE model. If the times are equal, the last event is deleted. Finally, the new event is added to the queue. This ensures that writing DE processes only add one event (t_i, v_i) to the queue.

Writing from CT port: An event is added to the queue. This method is applicable to CT signal flow model of computation. For netlists, additional actions are required that convert a physical size to a value of the events.

Note, that writing from SDF or CT model of computation might also trigger an event in the DE model of computation. However, this might not be useful or even cause problems. Usually, DE models are not activated by events at the data signals. DE models are usually activated by explicit control signals such as clock or enable signals, which is done by adapter classes that also provide such signals, and require an explicit controller.

For reading the queue of events that models the abstract signal, we use the following methods:

Reading from SDF ports: The value of the newest event is returned. For multi-rate dataflow, a conversion function is called which performs sample rate conversion. Sample rate conversion computes a weighted average of the values since the last call. Because time steps are constant, the time of the last call is the actual time minus the time step.

Reading from DE ports: The value of the newest event is returned.

Reading from CT ports: The value of the newest event is returned. This is applicable to the CT signal-flow model of computation, and for netlists. For netlists, additional actions are required that convert the abstract value to a physical size in a netlist.

Netlists and external simulators, Open issues. The implementation of polymorphic signals covers signals in SystemC-AMS, but not yet nodes in netlists. Actual work is to implement interfaces that support the coupling of external simulators, such as VHDL-AMS or SPICE, that also support simulation of netlists. For coupling external simulators of netlists, we extend the polymorphic signal using the following concept:

Netlists can write to polymorphic signals. This can be done by a port that leaves the netlist, and that specifies the physical sizes (e.g. a current, or a voltage), and its conversion to an abstract, non-conservative size. After conversion, this value is treated like a value from the CT signal-flow model of computation.

Netlists can 'read' from polymorphic signals, but require a small conversion circuit that is added to the netlist. This small conversion circuit is e.g. a cur-

rent or voltage source, and is also specified by attributes at the port (e.g. 1.7). Current work is to automatically insert such a circuit by a polymorphic signal into an external simulator.

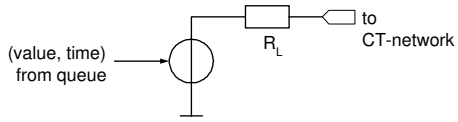


Figure 1.7. Simple: Connection of polymorphic signals with continuous-time network (CT-NET).

4. Case studies

For evaluation we have refined a PWM driver from an automotive application shown by Figure 1.8. The PWM driver is a control loop that controls the voltage of a power driver. The voltage is an average of pulses generated by a pulse generator. The voltage is measured, and a difference between the actual value and the programmed value is computed. Then, a PI controller computes a new pulse width. Figure 1.8 shows the executable specification, for which we used the model of computation continuous-time signal flow(CT-SF), as for example in Simulink block diagrams.

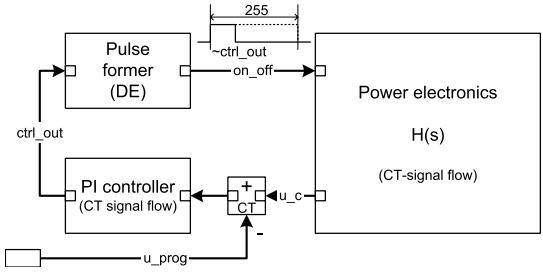


Figure 1.8. PWM driver: Executable specification.

For the refinement of computation we successively exchanged single modules with modules that are closer to implementation, and that use other models of computation (CT-signal flow → SDF → DE). We evaluated different partitionings of the design (e.g. using an analog implementation of the design), different sample rates, value ranges and bit widths. Most of these parameters have a tremendous impact to the dynamic behavior due to the nonlinear nature of the system. Figure 1.9 shows the design after refinement of interfaces.

Polymorphic signals were especially useful for the determination of sample rates, bit widths and value ranges: They allowed us to modify these parameters, or to replace single blocks by analog netlists — required sample rate

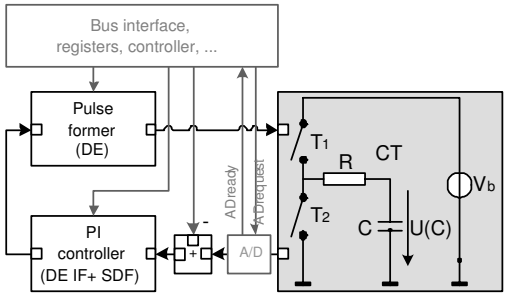


Figure 1.9. PWM driver: After refinement.

converters, A/D converters, etc. were automatically imitated by the polymorphic signals. Especially for design space exploration this has saved a lot of time for adapting models. Figure 1.10 shows two different output signals that are produced by changing the sample frequencies and models of computation.

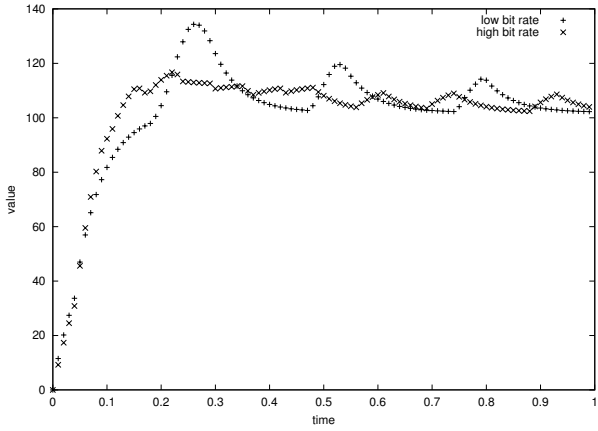


Figure 1.10. Output of PWM driver with different sample frequencies.

5. Discussion

SystemC provides a very generic approach for modeling digital systems. However, all models of computation are simulated by a discrete event simulator. Models of computation that have different interfaces or value types can only be converted, if they are subtypes. SystemC-AMS extends SystemC for modeling analog and mixed-signal systems. Semantic types resp. polymorphic signals as proposed in this paper permit an implicit conversion of different sig-

nal types and automatically treat semantic issues in the right way, e.g. the conversion of value ranges or sample rate reduction in signal processing applications.

Feedback from HELL. In first experiences, polymorphic signals allowed us to model and modify even complex systems with very little effort. However, modeling of mixed-signal systems at block diagram level negotiates many problems that occur in analog circuit design, and that have impact on system behavior. Therefore, HELL [Grimm et al., 2004] provides a framework for the semi-symbolic modeling and analysis of uncertainties introduced by the physical implementation. The aim of future work is to build a design framework that integrates HEAVEN and HELL of circuit design.

References

Karsten Einwich, Peter Schwarz, Christoph Grimm, and Klaus Waldschmidt. Mixed-Signal Extension for SystemC. In Eugenio Villar and Jean Mermet, editors, *System Specification and Design Languages*. Kluwer Academic Publishers, Apr 2003.

Christoph Grimm. Modeling and Refinement of Mixed Signal Systems with SystemC. In *SystemC – Methodologies and Applications*. Kluwer Academic Publisher (KAP), June 2003.

Christoph Grimm, Wilhelm Heupke, and Klaus Waldschmidt. Semi-Symbolic Modeling and Analysis of Noise in Heterogeneous Systems. In *Forum on Specification and Design Languages (FDL '04)*, Lille, France, September 2004.

Christoph Grimm and Klaus Waldschmidt. KIR – A graph-based model for description of mixed analog/digital systems. In *European Design Automation Conference*, Geneva, Switzerland, September 1996.

Christoph Grimm and Klaus Waldschmidt. Repartitioning and technology-mapping of electronic hybrid systems. In *Design, Automation and Test in Europe '98 (DATE)*, Paris, France, February 1998.

Edward Lee, Stephen Neuendorffer, and Michael Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, June 2003.

Edward Lee and Yuhong Xiong. A Behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 2004.

Robert Siegmund and Dietmar Müller. SystemC^{SV}: An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design. In *Design Automation and Test in Europe '01 (DATE 2001)*, Munich, 2001.

Stuart Swan. An Introduction to System-Level Modeling in SystemC 2.0. Technical report, Open SystemC Initiative, 2001.

Alain Vachoux, Christoph Grimm, and Karsten Einwich. Towards Analog and Mixed-Signal SoC Design with Systemc-AMS. In *IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)*, Perth, Australia, 2004.

Chapter 2

MIXED NETS, CONVERSION MODELS, AND VHDL-AMS

John Shields
Lynguent, Inc.
P.O. Box 19325
Portland, OR 97280-0325
jshields@ieee.org

Ernst Christen
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
Ernst.Christen@synopsys.com

Abstract AMS hardware description languages like VHDL-AMS provide features for modeling at discrete and continuous domains of abstraction and communicating between them. A mixed net arises in mixed-signal design as the result of interconnecting components modeled in different domains, in particular when connecting a discrete and a continuous port. Hardware description languages do not support such connections directly. They require the insertion of an appropriate conversion model between the dissimilar ports. Using conversion models correctly, a mixed net can be successfully partitioned and modeled with the desired blend of accuracy and performance.

This paper explains mixed nets and their various configurations, setting the requirements for needed conversion models. Conversion models are explained, including criteria for what makes a good one. Strategies for partitioning a mixed net and inserting conversion models are discussed. A proposal is made for extending VHDL-AMS to handle mixed nets and automatic insertion at elaboration.

Keywords: VHDL-AMS,Verilog-AMS,mixed signal,conversion models,elaboration

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 21–39.

1. Introduction

Our goal is to describe and simulate a mixed-signal system design with the required accuracy. By reducing model complexity, more of the overall system can be verified. But in order to verify some aspects, the most complex implementation models are needed. Mixed-signal HDLs like VHDL-AMS were designed to support these modeling tradeoffs. The design process using VHDL-AMS still leads to structural problems when switching between two models designed to represent the same component in different domains. Solving these problems efficiently requires conversion models to be systematically inserted.

1.1 Background and Motivation

Mixed-signal modeling involves using models in the discrete domain with models in the continuous domain. With VHDL-AMS, one can create rich models of physical systems of many different energy domains. In an electrical system, for example, the discrete domain is modeled by concurrently executing processes that communicate through signals of some logic type like std_logic. The electrical nature in the continuous domain is modeled with differential-algebraic equations of voltages, currents, and other unknowns. These models may be conservative systems, i.e., a circuit obeying Kirchoff's laws. They may also be signal flow models, where a non-conservative quantity (most likely voltage or current) flows through transfer functions.

A complete set of electrical components that can be used together for modeling a mixed-signal system will include digital models and both types of analog models. One component may have a set of models in different domains designed to be equivalent. Models that are from different domains have both different implementations *and* interfaces. Nevertheless, the interfaces are closely related. There is an equivalence between corresponding ports, i.e., a given logic signal port is equivalent to its corresponding electrical pin, voltage input, etc. At the same time, the analog model interface may have additional ports that are not relevant in the discrete model, such as power/ground connections.

Any component from the mixed-signal set may be connected to others to form a system model. Component models from different domains may be used interchangeably and should support flexible and efficient design styles. It follows that suitable conversions must exist between the discrete, the signal flow, and the conservative analog domains. Indeed, such conversion behavior is at the heart of languages like VHDL-AMS. The system model can be modeled in VHDL-AMS today. It turns out that suitable domain conversions must be designed as models themselves and be part of the component set. Effective system design styles can be supported if it is possible to specify reasonable rules to insert conversion models automatically and to bind implementations of such models to each instance of a conversion model.

1.2 Design Styles

Composing mixed-signal systems from a set of components is a structural task typically done best graphically, for example in a schematic entry system. Nevertheless, portions may be written directly in the HDL or generated from other design tools. A top-down design methodology leads from a high level of design abstraction at the system or behavioral level to a lower level of abstraction going toward the physical implementation level. Moving top-down in abstraction does not necessarily mean crossing modeling domain. Some components may move from behavioral to rtl to gate level to switch level and remain discrete models. The models ultimately exist as continuous models at the analog circuit level, yet there may be no need to use them in a system model for verification. If one can safely avoid using the circuit level model in the system model, the designer asserts that the digital model is equivalent to its analog counterpart. The designer further asserts that the component is sufficiently decoupled at this level in the system such that its analog aspects are accounted for and can be ignored.

There are few straightforward paths in top-down design for mixed-signal systems. Digital subsystems are decomposed with significant synthesis support. Analog subsystems have comparatively little synthesis. If your mixed-signal component set has analog components with a high level behavioral model and low level implementation model, it supports both top-down and bottom-up modeling. Designing an analog model may start with circuit topology at the implementation level and be modeled upwards, or from a top-down behavioral model and be modeled downwards. You adjust the parameterization to meet specifications. If you require an equivalent model at another abstraction level, there are two cases to consider. From the behavioral model, there is a creative leap to the implementation model and bottom-up verification to establish their equivalence. From the implementation model, there is a more organized and potentially automated transition upwards to the behavioral model. Bottom-up verification for equivalence is the same task, but here it takes the form of calibrating the behavioral model parameters to the implementation model.

Back at the system level, it may be ideal to simulate the entire system at the analog implementation level. When you choose not to, there is no substitute for bottom-up verification. As you proceed upwards in subsystem validation, component models start at the lowest level of abstraction and are swapped for equivalent models that are at higher levels of abstraction and/or cross domains from analog conservative to analog signal flow to discrete.

The common denominator of the top-down and bottom-up design styles is the need for a design composition system that is effective for hierarchical structure by providing flexible configuration of components and their underlying

simulation models. Schematic entry systems are effective, in part. They compose the structure flexibly and may be made configurable enough, but there is a difficult netlisting problem into the underlying HDL. VHDL-AMS can represent the structure provided the conversion models are explicitly included in the system model. VHDL-AMS also has the underlying features for design configuration that were intended to meet these needs. Unfortunately, they are not usable where component interfaces change across domains.

1.3 Problem Definition

Design styles suitable for mixed-signal systems need interchangeable component models from different domains, methods to efficiently switch between them, and automatic conversions between domains in order to meet the simulation accuracy and performance goals. VHDL-AMS supports the modeling needs for such components and the conversions, but lacks support to efficiently describe the hierarchical structure of the resulting systems so they can be reconfigured easily. After reviewing the mixed net modeling and conversion concepts, a solution to this problem will be proposed.

2. Mixed Nets and Conversion Models

A net consists of a root, typically a terminal, quantity or signal declared in a block, and all ports connected to the root, including transitive connections. Its structure is a tree. A mixed net is a net whose root and connected ports belong to different object classes. In a typical mixed net, some leaves of the tree may be terminal ports, other leaves may be quantity ports, and still others may be signal ports. The root and the ports higher up in the tree typically only define the connectivity between component instances; their semantics are usually not of much concern. Nonetheless, these ports and the root must be declared to be objects of a particular class: in VHDL-AMS, a terminal, a quantity, or a signal.

During simulation, it is the simulator's task to determine a value for each net, taking into consideration the contributions from the various ports that form the net (or more precisely, the contributions from the behavioral statements in which the names of the ports appear). VHDL-AMS defines semantics for uniform nets, that is, nets whose root and ports are either all terminals, or all quantities, or all signals. Other AMS languages have similar uniformity rules for nets. Therefore, to perform a simulation, a mixed net must be split into uniform portions using some partitioning strategy, and suitable code must be inserted at the boundaries of the different portions of the net to convert between the semantics of, for example, a terminal and a signal. The preferable way to manage such conversion code is to place it in *conversion models*, which then are instantiated such that they link the different portions of the net.

2.1 Net Partitioning Strategies

There are many different ways to split a mixed net into uniform portions, each with different properties. We discuss four strategies that embody different ideas, using VHDL-AMS terminology.

User-Defined Partitioning. In this strategy, the user defines the root of the net and each port of the net to have a particular object class: terminal, quantity, or signal. The object classes represent different modeling domains. Instances of conversion models are inserted between the formal and the actual of a port association element if the formal and actual are of different object classes. The benefit of this approach is that supporting it in VHDL-AMS requires few language changes. Its drawback is that even in simple situations it may be too difficult for a user to determine how to declare the ports in different parts of the net to achieve a certain goal (performance, accuracy). For example, it is easily possible that a mixed net might have several disjoint terminal nets (or nodes), each with a different potential.

The remaining three strategies have two aspects in common: We ignore the object class of the root and the intermediate ports and only honor the object class of the ports that are leaves of the tree forming the net. We also consider the object classes to correspond to abstraction levels, with a terminal being the most detailed and a signal being the most abstract.

Partitioning Driven by Elaboration. This strategy considers, for each vertex in the tree describing the net, the object class of the vertex and its immediate children and converts this portion of the net to the most detailed of these object classes. If the result is different from the object class of the vertex, then an instance of a conversion model is inserted between the vertex and its ancestor. The benefit of this approach is its simple elaboration rules. Its drawback is the sensitivity of its result to changes of a leaf port: replacing a leaf port connected higher up in the tree has more dramatic effects on the result than replacing a leaf port lower in the tree. That is, a structural design change may lead to an unexpected change in the mixed net representation and surprising behaviour.

Partitioning for Performance. The goal of this strategy is to minimize the number of instances of conversion models. Sub strategies include: a) simulating each mixed net as two or three uniform nets, each having a subset of the topology of the mixed net, and inserting instances of conversion models between the net replicas, and b) separating the signal net into two nets, one connecting all ports with mode **in**, the other, connecting all other signal ports, and inserting instances of conversion models between the terminal net (if any) and each signal net. The advantage of this strategy is performance. Its drawbacks are the complexity inherent in having multiple representations of a single net and the difficulty of incorporating drive and load characteristics in the con-

version models without adding significant features to the language, such as the capability to determine the fanins and fanouts of a port from inside a model.

Partitioning for Accuracy. In this strategy, a mixed net is converted in its entirety to a uniform net whose object class is that of the most detailed object class of the root or any port of the net. For example, if any terminal is connected to the net, then the net is converted to a terminal. Instances of conversion models are inserted between the net and any leaf port whose object class is different from the object class of the net. The benefits of this approach are its ability to accurately model drive and load characteristics at ports of a higher abstraction level and its predictability, which makes it easy to understand. Its drawback is the potentially large number of instances of conversion models, and performance may be affected by the models of the lowest abstraction connected to the net as well as by the number of conversion model instances

2.2 Categories of Conversion Models

The discussion about partitioning strategies yields the result that there is a need for conversion models that convert between the semantics of terminals, quantities, and signals. To also satisfy the VHDL-AMS rules about port association elements, which are based on the mode of the formal port, we end up with seven categories of conversion models:

- Terminal to signal with mode **in** (conservative to event-driven, commonly called a2d)
- Signal with mode **out** to terminal (event-driven to conservative, commonly called d2a)
- Signal with mode **inout** or **buffer** and terminal (commonly called bidirectional)
- Terminal to quantity with mode **in** (conservative to signal flow, called TQ below)
- Quantity with mode **out** to terminal (signal flow to conservative, called QT below)
- Quantity to signal with mode **in** (signal flow to event-driven, called QS below)
- Signal with mode **out** to quantity (event-driven to signal flow, called SQ below)

Note that there is no possibility to have a bidirectional conversion model between quantity and signal because of the semantics of a quantity net.

It is apparent from this list that a conversion model always has a direction, even in the case of a bidirectional conversion model, where the direction changes over time, driven either by the operation of the conversion model or by control information such as switching (on the signal end of the conversion model) between high impedance (input) and driving (output) state.

Each category of conversion models is further parameterized by the type of the signal or quantity or the nature of the terminal on either end of the conversion model. Regardless of the partitioning strategy, any mechanism to bind a category of conversion models to a particular implementation of a conversion model must be rich enough to support this parameterization.

2.3 Implementation of Conversion Models

The language elements of VHDL-AMS are sufficient to implement any conversion model, regardless of the particular combination of input and output object and the corresponding types and/or natures. For an input or output object of class terminal, this includes the possibility of converting between its reference quantity (for an electrical terminal: the voltage w.r.t. ground) or its contribution quantity (for an electrical terminal: the current flowing through the terminal) and the value of the object at the other end of the conversion model.

Conversion models between terminals and quantities are straightforward to implement because of the closeness of the semantics of the two object classes. A TQ conversion model is essentially a quantity source whose value is controlled by the reference or contribution quantity of the terminal. Similarly, a QT conversion model is either a quantity controlled across source or a quantity controlled through source.

Conversion models between signals and quantities or terminals have some similarities in that they involve converting between discrete time semantics and continuous time semantics. For an a2d or a QS conversion model, this can be accomplished, in general, using a threshold based approach that involves a signal of the form Q'Above(E), where Q is a quantity and E is the threshold. For a d2a or an SQ conversion model, the general approach is that of a controlled source whose value is driven by the value of the signal. A bidirectional conversion model combines the functionality of an a2d and a d2a conversion model, possibly with some extra code to switch its direction. In the remainder of this section, we will focus on a2d and d2a conversion models; SQ and QS conversion models are essentially subsets of a2d and d2a. We further restrict the discussion to conversion models with an electrical terminal and a signal of type std_logic.

The specific implementation of an a2d or a d2a conversion model can be rather ideal, taking into consideration only the voltage (or current) and pos-

sibly the impedance at the terminal end, or very detailed, modeling the load or driving characteristics of a particular technology such as cmos. In either case, the conversion model can be parameterized to match the properties of a particular physical device.

Ideal conversion models are the easiest to implement. As examples, we show the implementation of ideal a2d and d2a conversion models converting to or from a voltage. The a2d conversion model is based on a finite state machine that drives the output to '1' if the input voltage exceeds a threshold vhi, to '0' if the input voltage is below vlo, and to 'X' if the input voltage stays between vlo and vhi for longer than a timeout. A possible implementation of the model and the corresponding FSM are shown in Figure 2.3 and Figure 2.1 respectively [Christen, 1999].

The corresponding ideal d2a conversion model can be implemented as a voltage source with an output resistance where both the output voltage and the resistance are controlled by the signal value. A possible implementation of the model is shown in Figure 2.2 [Christen, 1999].

To better reflect the load and driving characteristics of a particular technology, a model writer can write technology specific conversion models that implement the load (for a2d) or the driving (for d2a) characteristics of the technology. For example, the driving characteristics of a conversion model for the cmos technology can be modeled by describing the channel properties of the two transistors at the output of a cmos gate, with its operation controlled by the input signal value. Conversion models with such detail typically need additional ports that provide the power supply for the model and the reference. They also have parameters that let the user parameterize an instance of the model to reflect the driving properties of a particular port of a physical device. The mechanism to bind a particular instance of a conversion model to a design unit with the necessary detail must therefore support the specification of appropriate parameter values for that instance and the connection of its power and reference terminals (and any other port that may be required, for example a port that controls the direction of a bidirectional conversion model) to suitable objects in the block in which the conversion model is instantiated.

3. Current Approaches to Automatic Insertion of Conversion Models

Once appropriate conversion models have been designed and components that give rise to their use are available, the designer focuses on the system design task. The designer is engaged in structural composition tasks, and automatic insertion of conversion models is very desirable for improved productivity, repeatability, and correctness. There are automated solutions today, but none that is well integrated with the VHDL-AMS language.

```
library ieee;
use ieee.std_logic_1164.all;  use ieee.electrical_systems.all;
entity a2d is
   generic (vlo, vhi: REAL; – thresholds
     timeout: DELAY_LENGTH);
port ( terminal ain, ref: electrical;  signal dout: out std_logic);
end entity a2d;
architecture Hysteresis of a2d is
   type st4  is (unknown, zero, one, unstable);
   quantity vin  across ain  to ref;
   function level(vin, vlo, vhi: REAL) return st4 is
   begin
      if vin < vlo  then return zero;
      elsif vin > vhi  then return one;
      else return unknown;
      end if;
   end function level;
begin
   process
      variable state:st4 := level(vin, vlo, vhi);
   begin
      case state is
      when one =>
        dout <= '1';
         wait on vin'Above(vhi);
        state := unstable;
      when zero =>
        dout <= '0';
         wait on vin'Above(vlo);
        state := unstable;
      when unknown =>
        dout <= 'X';
         wait on vin'Above(vhi), vin'Above(vlo);
        state := level(vin, vlo, vhi);
      when unstable =>
         wait on vin'Above(vhi), vin'Above(vlo) for timeout;
        state := level(vin, vlo, vhi);
      end case;
   end process;
end architecture Hysteresis;
```

Figure 2.1. Ideal a2d Conversion Model

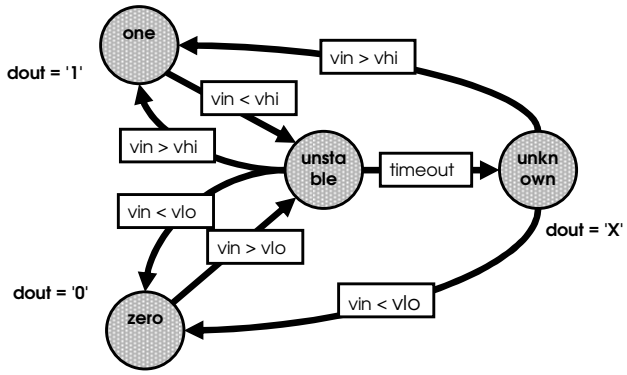


Figure 2.2. FSM For Ideal a2d Conversion Models

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.electrical_systems.all;
entity dac is
   generic (vlo : REAL := 0.2;
         vx : REAL := 2.5;
         vhi : REAL := 4.8;
         ron : REAL := 0.1;
         rwk : REAL := 1.0e4;
         rof : REAL := 1.0e9;
         tt : REAL := 1.0e-9)
   port ( signal din: in std_logic;
         terminal aout: electrical);
end entity dac;
architecture ideal of dac is
   type rt is array(std_logic) of REAL;
   constant r_table: rt := (ron, ron, ron, ron, rof, rwk, rwk, rwk, rof);
   constant v_table: rt := (vx, vx, vlo, vhi, vx, vx, vlo, vhi, vx);
   quantity vout across iout through aout;
   signal r, v: REAL;
begin
   r <= r_table(din);
   v <= v_table(din);
   vout == v'ramp(tt) + iout * r'ramp(tt);
end architecture ideal;
```

Figure 2.3. Ideal d2a Conversion Model

3.1 Netlisting

A common approach to automatic insertion of conversion models is based on extending netlisting tools in a schematic-based design environment. A schematics-based design is one captured and maintained using a schematic entry system. The schematics database is the master representation of the design. A netlister converts the information in the schematic database to an HDL representation. The netlister may insert conversion models automatically in the generated HDL source code. Often, annotation conventions in the form of global, sheet, symbol, and wire properties may be defined to drive the netlister's choice of conversion models, actual parameters, and location of insertion. An example of a robust implementation is the Synopsys Saber® Designer product. The inherent limitation here is that the master representation of the design must be in the schematics database in its entirety. Netlisting cannot insert conversions within portions of the design described in the HDL.

3.2 Verilog-AMS

The second approach involves insertion of conversion models during the elaboration phase of an HDL simulator. It is the more general and favored approach and applies equally well to schematics-based and HDL-based design. In this approach, the identification of conversion models, definition of mixed nets that require them, and specific locations for insertion are driven by features supported by the HDL. Verilog-AMS is the first AMS HDL with such features, and the discussion uses the terminology of this paper with Verilog-AMS terminology in parenthesis.

Verilog-AMS provides a straightforward mechanism to define a conversion model (connect module). It is distinct from a normal model (module) and relies on imposing a direction to a port of a continuous nature (discipline). The selection of a conversion model is driven by the explicit declaration of connect rules, and a mechanism exists to bind parameter actuals to the model in that declaration. These rules allow specifying conversion models between continuous and discrete disciplines. Conversion between signal flow and conservative disciplines does not require a conversion model; the meaning of such connections is defined by the discipline compatibility rules. The conversion model insertion between continuous and discrete disciplines can be configured to insert one model for all connections to/from discrete ports (merged rule) or one model per port (split rule). This capability to insert one model for all connections depends on the ability of a conversion model to look outside itself at the fanin/fanout of a port.

There are some lessons to learn. One shortcoming of the Verilog-AMS approach is due to its lack of strong typing in the base language. It is essential for a user to understand the nature of every object (net) connected together in a hi-

erarchical net (signal) to understand the impact of automatic conversion model insertion. Since it is possible to declare objects which are not strongly typed with respect to their nature (discipline), Verilog-AMS provides a capability to impose (force) a nature (discipline) on such objects. However, this discipline resolution is complex to understand and allows one to coerce relationships that may not make sense. It may lead to automatic model insertion in a manner not related to domain conversion.

More significant issues exist with the definition of the discipline resolution algorithm. There are two different algorithms that may be used by an implementation, basic and detailed, each allowing the insertion of conversion models that depend on looking outside the model at the fanin/fanout of a port for reasonable accuracy. (The basic algorithm virtually requires it.) This is a powerful feature, but not required to produce an accurate model of the mixed net (signal). A tool may support either algorithm, although they produce significantly different results. Therefore, Verilog-AMS designs that employ automatic insertion of conversion models are not portable.

4. Automatic Insertion of Conversion Models in VHDL-AMS

The following outline proposes new language features for VHDL-AMS structural modeling and automatic insertion of conversion models. Broad requirements are stated and the structural wire is introduced. Mixed nets may be constructed with wires with good semantics for all connections. User configured conversion models are inserted automatically where needed during elaboration. Open issues are noted.

4.1 Requirements

We believe that the following requirements must be met to provide robust support in the language for structural composition of designs containing instances of models at various abstraction levels.

1 Ability to configure a design with versions of components that differ in modeling domain, but represent the same device, easily.

2 Ability to structurally connect to ports of such components such that equivalent ports do not have to be re-connected, when models of different domains are swapped in.

3 Ability to automatically insert conversion models between domains to preserve an accurate representation of the design

4 Ability to model conversion at various abstraction levels (e.g. ideal, simple mos, detailed mos, etc.)

5 Ability to specify additional interface elements of a conversion model to be connected to model things like power supply accurately.

6 Ability to configure precisely and succinctly what conversion model is instantiated at each instance of a mixed connection in the design.

7 Preserve all VHDL-AMS semantics for strong type and nature checking.

The first requirement is outside the scope of conversion model insertion, but has a close relationship to it. With the current language definition, the instantiation statement must be rewritten when its component interface changes, which is what happens across domains.

4.2 The Structural Wire

The first feature needed for robust support of design composition provides the ability to create mixed nets while preserving the strong typing of the language. We propose to introduce a structural wire as a new object class into the language. The kinds of objects of interest are the terminal, the quantity, the signal and the wire. A wire is declared and may be used to connect to a port of a model. The corresponding port formal may be any object (i.e., signal, quantity, terminal, or another wire). While the other object classes have a specific subtype or subnature, a wire is purely structural and one is allowed to connect a wire to anything. A wire does have a shape. The concept of its shape refers to whether it is a scalar or a composite wire. A composite wire may be an array or a record. The declaration of the shape of a wire has same flexibility for arrays and records as subtypes and subnatures, as it builds on the current language features cleanly: one thinks of declaring a wire to be of the same shape as an existing subtype or subnature. The elements of a wire are named in a similar fashion as the elements of an object of the associated subtype or subnature: indexed names and selected names whose prefix is a composite wire are supported. A subelement of a wire is a wire.

The proposed language extensions to support wires include the semantics of wires and shapes, two attribute names that define the shape of a type or nature, the syntax of a wire declaration, and trivial enhancements to the object and interface declarations to include wires. The relevant new syntax elements are as follows:

T'SHAPE
Kind: Shape.
Prefix: Any type denoted by the static name T.
Result: The shape of the type denoted by T.
N'SHAPE
Kind: Shape.

Prefix: Any nature denoted by the static name N.
Result: The shape of the nature denoted by N.
wire_declaration ::=
wire identifier_list : shape_indication ;
interface_wire_declaration ::=
wire identifier_list : shape_indication
shape_indication ::=
type_mark ' SHAPE | nature_mark ' SHAPE

4.3 The Behavioral View of a Wire

There is a need to reference a wire as if it were a signal, quantity, or terminal in behavioral code, but that is not allowed due to the strong typing of the language. One workaround is to re-factor the design to always isolate the behavioral code from its structural interfaces at the block interface level. But that is as onerous as manual insertion of conversion models!

We believe that a better approach is to provide such access to a wire through the concept of a view of a wire. A wire view specifies sufficient information about the class, typing, and mode of access to satisfy all strong typing rules of VHDL-AMS. In effect, one is saying this wire is viewed as an object of the desired type or nature. Of course, this may ultimately imply automatic insertion of an appropriate conversion model in the block where the wire view is referenced. The proposed definitions for wire views are as follows:

W'TERMINAL(N)
Kind: Terminal.
Prefix: Any wire denoted by the static name W.
Parameter: A nature mark denoted by the name N.
Result nature: The nature defined by the nature mark N.
Result: A terminal whose nature is N.
Restrictions: N'SHAPE must match the shape of W
W'QUANTITY(T, mode)
Kind: Quantity.
Prefix: Any wire denoted by the static name W.
Parameters: T: A type mark denoted by the name T.
mode: The mode specifying how the quantity defined by the wire view is used. Must be either **in** or **out**.
Result type: The type defined by the type mark T.
Result: A quantity whose type is T and whose mode is as specified.
Restrictions: T'SHAPE must match the shape of W
W'SIGNAL(T, mode)

Kind:	Signal.
Prefix:	Any wire denoted by the static name W.
Parameters:	T: A type mark denoted by the name T. mode: The mode specifying how the signal defined by the wire view is used. Must be **in**, **out**, **inout**, or **buffer**.
Result type:	The type defined by the type mark T.
Result:	A signal whose type is T and whose mode is as specified
Restrictions:	T'SHAPE must match the shape of W

4.4 The Elaborated Model of the Mixed Net

The wire object forms a part of a mixed net. When a VHDL-AMS design is elaborated, the mixed net is elaborated. After semantic checks, a simulatable model will be produced, complete with automatically inserted conversion models wherever needed.

The elaboration of a mixed net involves:

1 Insertion of wire views at each port association element where either the formal or the actual, but not both, is a wire

2 Overall classification of the mixed net, which determines how it is modeled.

3 Determining the specific type or nature of each of its wires.

4 Mode propagation and semantic checks of each connection to determine validity

5 Binding of conversion models to each wire view

In a first step to classify a wire implementing a mixed net, each wire that is associated with an actual or formal that is not a wire is replaced by a wire view. The object class, type or nature, and mode are obtained from the object associated with the wire. After this replacement has been made, each connection has a formal and an actual that match in object class and type or nature, thereby satisfying the strong typing of the language.

In the second step, each wire is converted to a terminal, a quantity, or a signal. If any wire view anywhere in the mixed net is a terminal, the entire mixed net will be classified as a node. If not, but if there is a wire view that is a quantity, the mixed will be classified as a quantity net. There are rules governing the formulation of quantity nets to account for solvability. Finally, if all wire views are signals, the entire mixed net will be classified as a signal net. In either case, the nature of the node or the type of the quantity or signal net is obtained from the appropriate wire views. The result of following these precedence rules

is to create a net that preserves the accuracy of the connected objects. Wire conversion fails if a mixed net has incompatible wire views, for example two wire views that are terminals of different natures, or two wire views that are quantities of different types. In other words, automatically inserted conversion models may only serve to convert between different domains. They are not a back door to subvert strong typing.

When a mixed net has been classified to be of a particular class and type or nature, it is elaborated as if it were a net of that class and type or nature. If it is a quantity net or a signal net, each formal port that was converted from a wire must be given a mode. The mode is determined using the modes of all wire views of this port and of all formal ports with which this port is associated as an actual. The rules guarantee that the language rules about modes at a connection are satisfied.

The result of these steps is a consistent implementation of each mixed net at the accuracy requested, and clearly identified locations where conversion models must be inserted. These locations are the locations of the wire views. Since the properties of the converted wire are known as well as the properties of the other end of each wire view, we have enough information to bind one representative of a collection of conversion models to each wire view. Of course, if the two ends of a wire view are type or nature compatible, no conversion is needed.

4.5 Automatic Conversion Models and Wire Configuration Rules

The remaining issue is the specification of a particular entity/architecture for each instance of a conversion model and its proper instantiation. A *wire configuration specification* identifies a collection or a class of wires and associates binding information with the wire views of these wires. The wires may appear in the port association list or the declarative region of the block in which the wire configuration specification appears and any block nested within the block.

```
wire_configuration_specification ::=
    for wire_object_specification
        { conversion_specification }
    end for ;
object_specification ::=
    terminal name_list : nature_mark
    | quantity name_list : [ in | out ] type_mark
    | signal name_list : [ mode ] type_mark
name_list ::=
    simple_name { , simple_name }
    | others
    | all
```

A *conversion specification* associates binding information with wire views of the wires identified by the enclosing wire configuration specification.

conversion_specification ::=
for object_specification binding_indication ;

The binding indication of a conversion specification supports binding any entity/architecture pair with a wire view specified by the combination of its object specification and wire object specification of the enclosing wire configuration specification. For an a2d, d2a or bidirectional conversion model, this includes architectures converting between the reference quantity or the contribution quantity of the terminal involved and the object at the other end of the conversion model. Additionally, the generic map and port map of the binding indication provide the means to associate the formal arguments and ports of the conversion model specified by the binding indication with actual arguments and ports. Semantically, the existing definition of a binding indication must be extended slightly.

Wire configuration specifications may appear anywhere a configuration specification may appear, and additionally in the declarative region of an entity. They may also be separately specified in a configuration declaration. A wire configuration specification applies to the elaboration of the region in which it has been declared and in the sub hierarchy of the design rooted at that region, unless it is superseded by a more specific rule. (It is a detail to state carefully that there is a similar mapping of a rule declared in a block of configuration declaration to sub hierarchies of the design.). A wire configuration specification supersedes a prior rule if it specifies the same wire view, but appears lower in the design hierarchy. There are other possible ways to map rules to the design hierarchy, but that is a usability issue we don't discuss here.

4.6 Examples

Digital and analog implementation of inverter model using explicit wire views:

```
library ieee;
use ieee.electrical_systems.all;
entity inverter is
      port (wire input, output: REAL'SHAPE;
        terminal supply: electrical);
end entity inverter;
architecture digital of inverter is
begin
     output'SIGNAL(BIT, out) <= not input'SIGNAL(BIT,in);
```

end architecture digital;
architecture analog **of** inverter **is**
quantity vin **across** input'TERMINAL(electrical);
quantity vout **across** iout **through** output'TERMINAL(electrical);
quantity vcc **across** supply;
begin
vout == vcc – vin;
end architecture analog;

Wire configurations:
for terminal w1, w2: electrical
– named wires converted to terminals with nature electrical
for signal all: **in** std_logic **use entity** work.mosa2d
port map (a=>**wire**, d=>**signal**);
end for;
for terminal others: electrical
– other wires converted to terminals with nature electrical
for signal all: **in** std_logic **use entity** work.a2d
port map (a=>**wire**, d=>**signal**);
for signal all: **out** std_logic **use entity** work.d2a
port map (d=>**signal**, a=>**wire**);
for quantity all: **in** REAL **use entity** work.tq
port map (a=>**wire**, q=>**quantity**);
end for;

4.7 Open Issues

There are open issues at several levels. Conceptually, we believe there is insufficient information if a wire as a formal is assiciated with a quantity or signal as an actual and the wire is converted to a node. In this situation, no information is available as to what the mode of the corresponding wire view should be. It is unresolved whether it is possible to support a connection association element whose formal is a wire and whose actual is not a wire. Definitionally, we have not worked out the semantics to make a wire configuration specification applicable across a sub hierarchy of a design. The definition of the elaboration semantics that involve the steps after a wire has been classified as either a terminal, or a quantity, or a signal, are incomplete. Initialization of a net needs further analysis. Many other places in the LRM need minor changes to support the described functionality; these places have not been identified.

We chose a mixed net partitioning model for accuracy and rejected additional requirements that may improve performance by reducing conversion model count. The complexity in language definition as well as for user, model writer, and simulator implementor is judged not to be worth the potential gain.

Perhaps there are some important use cases that we are overlooking. In any case, adding support for such a partitioning strategy is not likely to invalidate any of the proposed language changes, only to extend them.

5. Conclusion

There is need to support structural design methodologies in mixed-signal modeling that lead to the creation of mixed nets. VHDL-AMS is effective at describing a wide range of mixed systems, but structural decomposition or bottom-up composition of mixed-signal components is not well supported. In particular, connections across domains, that is, mixed nets, are not allowed. The proposed language extensions provide the needed support for these methodologies using the concept of a wire and automatically inserted conversion models. It is possible to design good conversion models to effectively balance accuracy and performance of the mixed net. The wire object class adds important structural flexibility to the language while, through the wire resolution rules, preserving the strong semantics of the language type and nature system. Rule-based automatic conversion model insertion supports accuracy with very fine discrimination of what conversion model to use in any mixed connection, yet makes it very simple to generalize about model choice. Overall, there is a good opportunity to improve VHDL-AMS to support the re-configuration of mixed-signal systems effectively. The authors are working through a formal language change proposal for VHDL-AMS and welcome feedback.

References

P. Ashenden, G. Peterson, D. Teegarden: *The System Designer's Guide to VHDL-AMS.* Morgan-Kaufman Publishers; 2003.

E. Christen, K.Bakalar, A.M. Dewey, E. Moser: *Analog and Mixed-Signal Modeling Using the VHDL-AMS Language*; Tutorial at 36th Design Automation Conference, 1999

IEEE Std. 1076.1 - 1999 IEEE Standard VHDL Analog and Mixed-Signal Extensions

Verilog-AMS language Reference Manual, Version 2.0. Open Verilog International; February, 2000.

Chapter 3

MONTE CARLO SIMULATION USING VHDL-AMS

Ekkehart-Peter Wagner
Siemens VDO Automotive AG
Regensburg, Germany
ekkehart-peter.wagner@siemens.com

Joachim Haase
Fraunhofer-Institut Integrierte Schaltungen
Branch Lab EAS Dresden, Germany
Joachim.Haase@eas.iis.fraunhofer.de

Abstract Monte Carlo simulation is widely used in Spice-like circuit simulators. It allows to obtain statistical information derived from estimates of the random variability of circuit parameters. Multiple simulation runs are carried out with different sets of parameters. VHDL-AMS provides flexible possibilities to specify nominal and tolerance values and their distributions. Correlation between parameters can easily be taken into account. This is especially important if behavioral models are considered. The paper describes requirements and implementation aspects of the Monte Carlo simulation using VHDL-AMS.

Keywords: Monte Carlo simulation, VHDL-AMS

1. Introduction

Within industrial applications the tolerance- and worst-case-analysis considering all known influencing factors of design parameters are required very often. Reliability and yield of electronic circuits depend on the statistical characteristics of such parameters. One method for analyzing the effects of tolerances is simulation using Monte Carlo methods. In a Monte Carlo simulation, a mathematical model of a system is repeatedly evaluated. Each run uses different values of system parameters. The selection of the parameter values is

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 41–54.

made randomly with respect to given distribution functions [O'Connor, 2002]. Monte Carlo simulation is very time consuming. A lot of simulation runs are required to investigate the behavior of a system subject to the statistical distribution of parameters. Nevertheless, Monte Carlo simulation is very favored simulating electrical circuits and systems. It is widely supported by Spice-like simulation engines. Monte Carlo features are usually available for frequency and time domain analysis [Vlach and Singhal, 1994].

In VHDL-AMS [IEEE Std 1076.1, 1999] applications it becomes increasingly interesting to make Monte Carlo features available. The basic requirements for statistical simulation linked to VHDL-AMS are summarized by Christen in [Christen, online]. His paper concludes that support for statistical modeling can be provided using VHDL packages. He discusses the requirements for Monte Carlo and time series simulation support during the phase of the VHDL-AMS language design. However, at the moment, eight years later, a uniform standard approach to solve these problems in existing VHDL-AMS simulators still does not exist to the knowledge of the authors. Some ideas concerning time series simulation were reported in [Monnerie et al, 2003]. In this paper we present *first experiences* how to implement some of the requirements for Monte Carlo simulation [Christen, online]

- Usage of the same model for nominal and Monte Carlo analysis
- Assignment of different statistical distributions that are parameterizable to each constant
- Support of continuous and discrete distributions
- Possibility to specify correlation between constants

From a practical point of view the following points should also be mentioned

- Independent random number generation for any constant
- Reproducibility of Monte Carlo simulation within the same simulation tool

Reproducibility of Monte Carlo simulation in different VHDL-AMS simulation tools would be desirable. We will start with a discussion of the implementation of random number generators for Monte Carlo simulation. Afterwards, we will show how to implement these generators in VHDL-AMS. We will continue with a simple example and conclude with some remarks about further directions.

2. Random Number Generators

2.1 Basic Problems

Initialization of the Pseudo-Random Number Generator. One of the basic problems in Monte Carlo simulation is the generation of random numbers. In Monte Carlo simulations of electrical circuits pseudo-random numbers are typically used. Different approaches to generate such numbers exist. The MATH_REAL package [IEEE Std 1076.2, 1996] of the IEEE library provides a procedure UNIFORM that returns a pseudo-random number with uniform distribution in the open interval $(0, 1)$. The procedure is declared in the following way:

```
procedure UNIFORM(variable SEED1,SEED2:inout POSITIVE;
                  variable X:out REAL);
```

The algorithm is based on the combination of two multiplicative linear congruential generators. It was published by L'Ecuyer [L'Ecuyer, 1988]. An advantage of the L'Ecuyer generator is its long period [Graham, 1992]. The VHDL implementation requires the seed values (SEED1, SEED2) to be initialized before the first call to UNIFORM. The seed values are modified after each call to UNIFORM. In order to generate a chain of pseudo-random numbers, the seed values shall be set only in the first call of the procedure (see Annex A.3 of [IEEE Std 1076.2, 1996]). In the next call the seed values from the previous call have to be used. A different chain of numbers is started every time the seed values are set.

The Ada implementation of the L'Ecuyer generator provides an INITIALIZE procedure that sets two global initial seed values that are updated during every call of the random number generator [8]. An equivalent procedure is not available in the MATH_REAL package. However, a similar functionality is needed in Monte Carlo simulations. Thus, the pseudo-random generator is used to initialize constant objects declared in different design units. The state of the generator has to be passed from one call to the next one by using seed values from a previous call. This can be done in a well-defined way for instance inside a PROCESS statement. The seed values can be held in VARIABLE objects. This approach can not be used e.g. during initialization of generic constants or constants that are declared in different design units. Thus, another approach has to be used. The state of the random number generator can be held for example in a

- SHARED VARIABLE or a
- FILE

IEEE DASC P1076a Shared Variable Working Group specified mutually exclusive access semantics for shared variables [IEEE PAR 1076a, online]. If this

work could be the base for an extension of the capabilities of random number generation in the IEEE packages. The seeds could be global variables, functions to initialize their values (INIT_SEED) could be provided, and the UNIFORM procedure would have to be modified accordingly. But shared variables are currently not implemented in all available VHDL-AMS simulators.

Due to these existing limitations concerning shared variables we followed the second approach. Seed values are read and written into a file before and after a call of the UNIFORM procedure in the context of Monte Carlo simulation. If at the beginning of a Monte Carlo simulation run the same file is used then the same results will be produced by the simulator. It is assumed that in the elaboration phase (see [IEEE Std 1076.1, 1999], chapter 12) the calls of the UNIFORM procedure will be carried out in a sequential manner. The elaboration is carried out in the same way prior to the execution phase in every simulation run. Thus, reproducibility is assured if the elaboration phase starts with the same file. On the other hand, every run during Monte Carlo simulation starts with a different file and can work with a different parameter set. The next simulation run starts with the updated file of the last run. This procedure

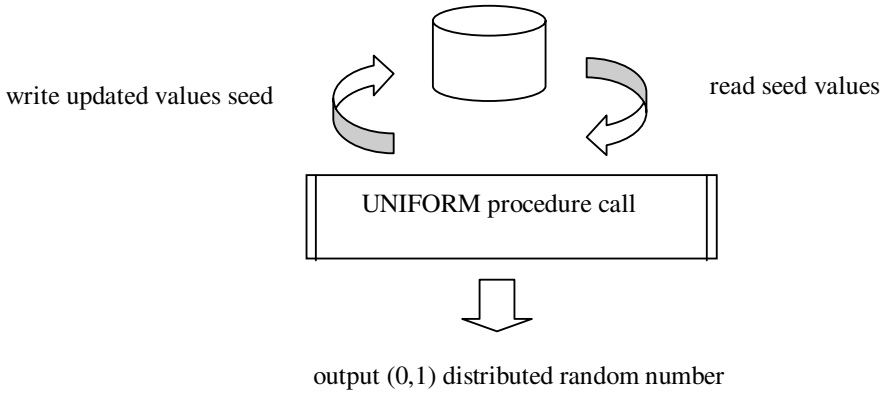


Figure 3.1. UNIFORM procedure call in the elaboration phase.

only sufficiently works in the elaboration phase. It is evident that it is not cycle pure. Thus, it can not be applied in the execution phase of a VHDL-AMS simulation.

Transformation of Uniform Random Distribution. Many process and device model parameters are not $(0, 1)$ uniform distributed. Generally applied distributions used in Monte Carlo simulation are for example:

- Uniform distribution between a and b $(a < b)$

- Gaussian distribution $N(\mu, \sigma)$ (also called normal distribution) with mean value μ and standard deviation σ
- Bernoulli distribution having two possible outcomes with probability $p = 0.5$.

Other distributions as triangular and lognormal distributions can also be implemented. Furthermore, the support of user-defined discrete and continuous distributions is expected. Non-uniform distributed random numbers can be generated using von NeumannÆs method of generating random samples by evaluating the position of uniform random numbers in a given rectangle or by transformation. The first methods generate samples from any distribution whose probability density function is piecewise continuous and monotonic [Forsythe, 1972]. It can be used to take user-defined continuous distributions into account. In the second approach, a $(0, 1)$ uniform distributed value is transformed through a function to a new value that follows a non-uniform distribution. How this works will be shown in the following examples. Let X be a $(0, 1)$ uniform random distributed number then

$$Y = a + (b - a) \cdot X \tag{3.1}$$

is a uniform distributed number between a and b. Let X_1 and X_2 be independent $(0, 1)$ uniform distributed numbers then

$$Y_1 = \mu + \sigma \cdot \sqrt{-2 \cdot \ln(X_1)} \cdot \cos(2\pi \cdot X_2) \tag{3.2}$$

$$Y_2 = \mu + \sigma \cdot \sqrt{-2 \cdot \ln(X_1)} \cdot \sin(2\pi \cdot X_2) \tag{3.3}$$

are $N(\mu, \sigma)$ normal distributed numbers [Box and Muller, 1958]. Another way is to start with 12 $(0, 1)$ uniform distributed numbers $X_i (i = 1 \text{ to } 12)$ then

$$Y = \mu + \sigma \cdot \left(\sum_{i=1}^{n} X_i - 6 \right) \tag{3.4}$$

is also $N(m, \sigma)$ normal distributed [Schrüfer, 1990]. Let X be a $(0, 1)$ uniform distributed number then

$$Y = \begin{cases} v_1 \in \mathbb{R} & \text{for} \quad X \leq 0.5 \\ v_2 \in \mathbb{R} & \text{otherwise} \end{cases} \tag{3.5}$$

is a Bernoulli distributed number with the two real values v_1 and v_2 that occur with the same probability. We can interpret v_1 and v_2 as minimum and maximum of a parameter. We use the name worst case distribution for this distribution in the following. In the same way, random numbers with other distributions can be generated. Figure 3.2 shows how to combine these random

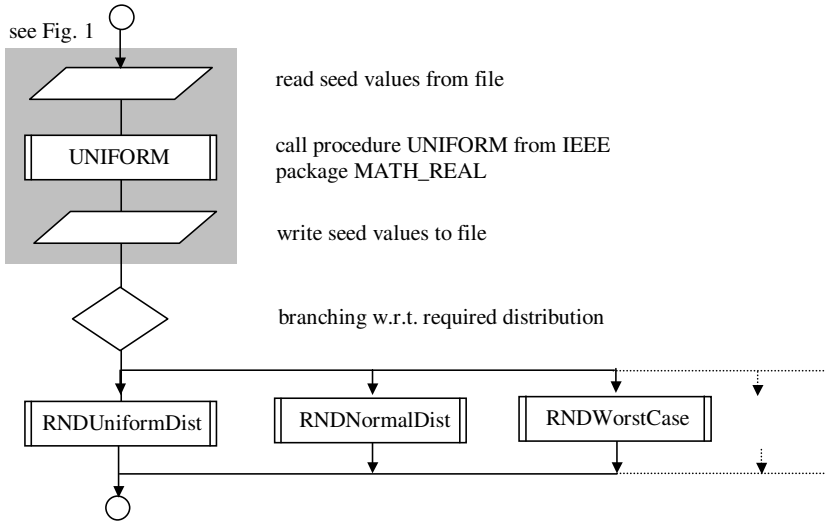


Figure 3.2. Principle of non-uniform random number generation.

number generators in VHDL-AMS. The mean value of the random numbers is the nominal value of the random constant that has to be initialized during Monte Carlo simulation. Figure 3.2 describes the main structure of a function RND that can be used to initialize random constants in Monte Carlo simulation runs. The function will be introduced in the next section. Figure 3.2 can be extended by further general distributions.

Correlation between Random Numbers. In some cases, statistical circuit simulation requires the consideration of the correlation between random variables. This correlation is described by the correlation matrix R. The correlation matrix is a symmetric positive (semi-)definite matrix. Only in the case of Gaussian random numbers it is easy to generate correlated Gaussian random numbers [Esbaugh, 1992]. It is assumed that $Y_1, Y_2, \ldots, Y_n$ shall be Gaussian random numbers with mean values $\mu_1, \mu_2, \ldots, \mu_n$ and standard deviation $\sigma_1, \sigma_2, \ldots, \sigma_n$. R is the correlation matrix. The element r_{ij} $(-1 \leq r_{ij} \leq 1)$ describes the correlation between Y_i and Y_j. To make $Y_1, Y_2, \ldots, Y_n$ available, $N(0,1)$ normal distributed independent random numbers $X_1, X_2, \ldots, X_n$ are generated. A Cholesky decomposition of R is carried out, i.e. $R = G^T \cdot G$.

Then it follows

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{pmatrix} + \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{pmatrix} \cdot G^T \cdot \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{pmatrix} \tag{3.6}$$

That means in the general case, an algorithm that carries out Cholesky decomposition has to be implemented. In simple cases (i.e. for small n) the equation (3.6) can be solved analytically.

Example Y_1 and Y_2 are Gaussian random numbers with mean values m_1 and m_2,and standard deviation σ_1 and σ_2. The correlation between Y_1 and Y_2 is r_{12}. We get

$$R = \begin{pmatrix} 1 & r_{12} \\ r_{12} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ r_{12} & \sqrt{1-r_{12}^2} \end{pmatrix} \cdot \begin{pmatrix} 1 & r_{12} \\ 0 & \sqrt{1-r_{12}^2} \end{pmatrix} \tag{3.7}$$

and

$$\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} + \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ r_{12} & \sqrt{1-r_{12}^2} \end{pmatrix} \cdot \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \tag{3.8}$$

That means

$$Y_1 = \mu_1 + \sigma_1 \cdot X_1 \tag{3.9}$$

$$Y_2 = \mu_2 + \sigma_1 \cdot r_{12} \cdot X_1 + \sigma_2 \cdot \sqrt{1-r_{12}^2} \cdot X_2 \tag{3.10}$$

The equations (3.9) and (3.10) can easily be implemented. In practice, non-Gaussian data have to be considered in numerous applications. Their probability density function can be expressed in many cases by a truncated Gram-Charlier series expansion using central moments. Different algorithms are proposed to generate correlated non-Gaussian random variables. A special approach that uses the first four central moments is suggested in [Karvanen, 2003]. The handling of correlated random parameters depends on a lot of requirements that may differ from application to application. VHDL-AMS provides a lot of facilities to support these requirements. However, it seems to be difficult or requires a high effort to make some general methods available to generate correlated non-Gaussian numbers beside trivial cases (e.g. two correlated Gaussian variables). This is, we do not consider these methods in the following.

2.2 Implementation in VHDL-AMS

To realize the functionality described in section 2.1 two VHDL-AMS packages were developed:

- `package STATISTIC_GLOBAL`
- `package STATISTIC`

Both should be compiled into a logical library symbolically named MONTE_CARLO_LIB.

Package STATISTIC_GLOBAL. In the header of the package STATISTIC_GLOBAL two deferred constants are declared

```
constant GLOBAL_STATISTIC : GLOBAL_STATISTIC_TYPE;
constant GLOBAL_FILE_NAME : STRING;
```

The first constant allows to decide whether an analysis with nominal values or a Monte Carlo simulation shall be carried out. The enumerated type GLOBAL_STATISTIC_TYPE consists of the values GLOBAL_NOMINAL and GLOBAL_MONTE_CARLO. The initialization of the constant is done in the package body (see also [Christen, online]). The constant GLOBAL_FILE_NAME has to be initialized with the relative or full name of the file that carries the seed values (compare Figure 3.1). The values are saved in ASCII format. Prior to the first simulation, the initial values must meet the requirements concerning SEED1 and SEED2 that are parameters of the UNIFORM procedure [IEEE Std 1076.2, 1996].

Package STATISTIC. In the package body a function is declared that realizes a random generator with $(0, 1)$ distribution:

```
impure function UNIFORM01
  return REAL is
      variable RESULT   : REAL;
      variable SEED     : INTEGER_VECTOR (0 to 1);
  begin
      SEED := READ_SEED;
    UNIFORM (SEED(0), SEED(1), RESULT);
    WRITE_SEED (SEED);
return RESULT;
end function UNIFORM01;
```

READ_SEED and WRITE_SEED are two further functions to read and write from a file characterized by the constant GLOBAL_FILE_NAME. The function UNIFORM01 corresponds to Figure 3.1. At the moment, the functions RNDUniformDistDIST, RNDNormalDist, and RNDWorst Case that correspond with the equations (3.1), (3.4), and (3.5) resp. are declared. Other distributions will be supplemented in the future. The code of the function RNDUniformDist demonstrates the implementation of (3.1). The first parameter is NOMINAL_VALUE that corresponds to the mean value μ. The second parameter

TOL determines $a = \mu \cdot (1 - TOL)$ and $b = \mu \cdot (1 + TOL)$ in (3.1). The third parameter RND01 is transferred from the result of a call of the random number generator UNIFORM01:

```
function RNDUniformDist
  (NOMINAL_VALUE : REAL; TOL : REAL; RND01 : REAL)
  return REAL is
    variable A      : REAL;
    variable B      : REAL;
    variable RESULT : REAL;
  begin
    A      := NOMINAL_VALUE*(1.0 - TOL);
    B      := NOMINAL_VALUE*(1.0 + TOL);
    RESULT := A + RND01*(B - A);
return RESULT;
end function RNDUniformDist;
```

In the header of the package STATISTIC, the functions SET_TOL_UniformDist, SET_TOL_NormalDist, SET_TOL_WorstCase, and RND are made available:

```
-- Set tolerances for uniform distributed values equ. (1)

   function SET_TOL_UniformDist (
       TOL : REAL      -- A = NOMINAL_VALUE*(1.0-TOL)
       )               -- B = NOMINAL_VALUE*(1.0+TOL)
   return TOL_DATA;

-- Set tolerances for normal distributed values equ. (4)

   function SET_TOL_NormalDist (
        SIGMA : REAL   -- standard deviation
        )
   return TOL_DATA;

-- Set tolerances for Bernoulli distribution equ. (5)

   function SET_TOL_WorstCase (
        TOL : REAL    -- V1 = NOMINAL_VALUE*(1.0-TOL)
        )             -- V2 = NOMINAL_VALUE*(1.0+TOL)
   return TOL_DATA;


-- Function that changes  NOMINAL_VALUE w.r.t. tolerances
```

```
function RND  (
     NOMINAL_VALUE : REAL;
     TOL           : TOL_DATA)
return REAL;
```

Using the SET_TOL functions a value can be assigned to a data object of the type TOL_DATA that is also declared in the package STATISTIC. By evaluating the data object the type of the distribution and the tolerance values can be determined. The function RND realizes the flow given by Figure 3.2. If the constant GLOBAL_STATISTIC from the package STATISTIC_GLOBAL is set to GLOBAL_NOMINAL then the function RND returns the NOMINAL_VALUE. Otherwise, it generates a random number with a mean value that equals the NOMINAL_VALUE and with a distribution given by the second parameter TOL. The TOL parameter can be initialized with the SET_TOL functions.

Usage of Packages. The functions can be used together with existing models. Let us have a look at the VHDL-AMS model of a resistor. P and M are the electrical terminals. The value of the resistance is given by the generic parameter R:

```
library IEEE;
use IEEE.ELECTRICAL_SYSTEMS.all;
entity RESISTOR is
  generic (R : REAL);
  port (terminal P, N : ELECTRICAL);
end entity RESISTOR;

architecture BASIC of RESISTOR is
  quantity V across I through P to N;
begin
  V == R*I;
end architecture BASIC;
```

This model can be instantiated in a VHDL-AMS architecture. The functions that are declared in the header of the package STATISTIC can be used to assign random values to the generic parameter R. This may look like

```
library MONTE_CARLO_LIB;
use MONTE_CARLO_LIB.STATISTIC.all;
...
R1: entity RESISTOR (BASIC)
   generic map (R => RND(5.0E3, SET_TOL_WorstCase(0.01))
   port map    (P => ..., N => ...)
```

The nominal value of the resistance is 5.0 kΩ. During Monte Carlo Simulation 5.0 kΩ± 1% are used. Following this approach, existing models can be used in Monte Carlo simulation. Furthermore, it is also possible to define special architectures that describe elements with given tolerances. For instance, the following architecture TEN_PERC describes a resistor with 10 % tolerance:

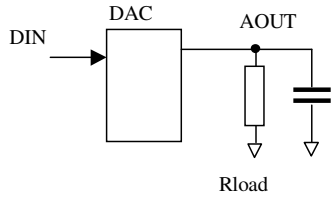
```
library MONTE_CARLO_LIB;
use MONTE_CARLO_LIB.STATISTIC.all;
architecture TEN_PERC of RESISTOR is
  constant TOL : TOL_TYPE := SET_TOL_WorstCase(0.1);
  constant RES : REAL     := RND(R, TOL);
  quantity V across I through P to N;
begin
  V == RES*I;
end architecture TEN_PERC;
```

This model can then be instantiated without special knowledge of the statistical packages:

```
...
R1: entity RESISTOR (TEN_PERC)
      generic map (R => 5.0E3)
      port map    (P => ..., N => ...)
...
```

3. Example

One of the advantages of using Monte Carlo simulation with VHDL-AMS is the possibility to apply it on mixed-signal circuits. Figure 3.3 shows a typical example.



VHDL-AMS model (extract)

```
-- 1 % tolerance
constant t1 : TOL_DATA
   := SET_TOL_WorstCase(0.01);
...
constant Rload : REAL
   := RND (1.0E6, t1);
constant Cload : REAL
   := RND (1.0E-12, t1);
```

Values of load resistor and load capacitor are random parameters. The statistical influence of output resistors of the DAC is investigated in the Monte Carlo simulation.

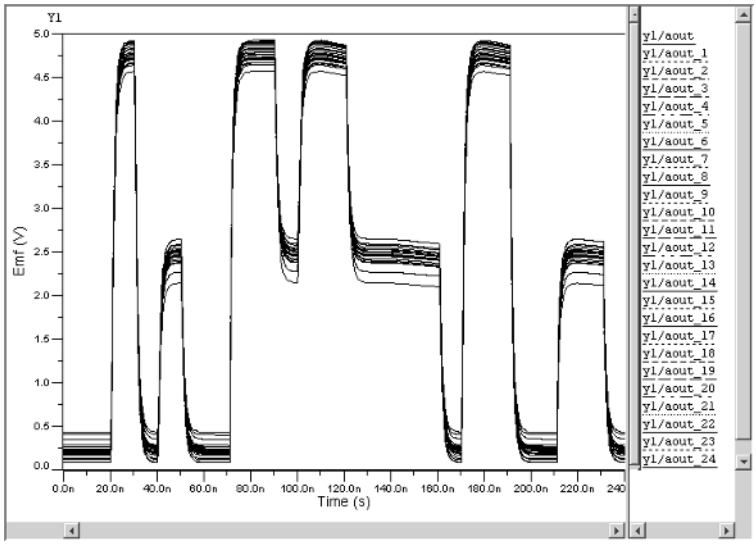


Figure 3.3. DAC with input signal DIN and voltages at AOUT.

4. Further Directions

We gained first experiences with Monte Carlo simulation using VHDL-AMS. Further work will include other probability distributions. We will also include user-defined discrete and piecewise-linear distributions. We also have to check the quality of the generated numbers. To our opinion, it should be checked whether the definition of statistical packages for Monte Carlo simulation could be part of further activities of the 1076.1 working group. A problem to be solved in a unified and easy way particularly concerns the initialization of UNIFORM or an equivalent procedure and the update of the seed values in the Monte Carlo simulation runs. It should be assured that Monte Carlo simulation using VHDL-AMS delivers the same results in different simulators. Simulators should support Monte Carlo simulation of VHDL-AMS descriptions. Some aspects are for instance

- Supplement of multi-run-simulations into the list of available analyses. The simulation program should know that the Monte Carlo feature is used. This could avoid unnecessary repetition of some of the stages of the evaluation phase as for instance reading the netlist.
- Support of the initialization of (global) seed values in an easy way
- Implementation of statistical post-processing-tools (for generating histograms, calculating envelopes, mean values, variances, ...)

Furthermore, the usage of Monte Carlo simulation together with the behavioral modeling language VHDL-AMS opens a lot of other opportunities. For instance, results from Monte Carlo simulation could be used for the generation of response surface models [Box and Draper,1987]. In this case, parameters and selected simulation results of each run should be saved and evaluated afterwards. One could also influence the generation of parameters for different simulation runs by some add-on tools. There is no limit to other ideas.

References

Box, G.E.P., and Draper, N.R. (1987). *Empirical Model-Building and Response Surfaces.* New York: John Wiley & Sons.

Box, G.E.P., and M.E. Muller. "A Note on the Generation of Random Normal Deviates," Annals Math. Stat. 29(1958), pp. 610–611.

Christen, E. "Statistical Modeling," Available: `http://www.vhdl.org/analog/wwwpages/language_proposal/STAT.html`

Esbaugh, K.S. "Generation of correlated parameters for statistical circuit simulation," Trans. on CAD 11(1992)10, pp. 1198–1206.

Forsythe, G.E. (1972). *Von Neumann's comparison method for random sampling from the normal and other distributions.* Report CS-TR-72-254. Stanford University. Available: `ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/72/254/CS-TR-72-254.pdf`

Graham, W.N. "A Comparison of Four Pseudo Random Number Generators Implemented in Ada," ACM SIGSIM Simulation Digest 22(1992)2, pp. 3–18.

IEEE Standard VHDL Analog and Mixed-Signal Extensions (IEEE Std 1076.1-1999). Approved 18 March 1999. Available: `http://www.designers-guide.com/Modeling/1076.1-1999.pdf`

IEEE Standard VHDL Mathematical Packages (IEEE Std 1076.2-1996). Approved 19 September 1996.

Karvanen, J. "Generation of Correlated Non-Gaussian Random Variables from Independent Components," Proc. 4th Int. Symposium on Independent Component Analysis and Blind Signal Separation ICA 2003, April 2003, Nara (Japan), pp. 769–774.

L'Ecuyer, P. "Efficient and Portable Combined Random Number Generators," Communications of the ACM 31(1988)6, pp. 742–774.

Monnerie, G., N. Lewis, D. Dallet, H. Levi, and Robbe, M. "Modelling of transient noise sources with VHDL-AMS and normative spectral interpretation," Proc. Forum on Specification & Design Languages FDL'03, September 23–26, 2003, Frankfurt/M., pp. 108-119.

O'Connor P.D.T. (2002). *Practical Reliability Engineering.* Chichester: John Wiley & Sons Ldt.

Schrüfer, E. (1990). *Signalverarbeitung.* München-Wien: Carl Hanser Verlag.

Shared Variable WG (IEEE PAR 1076a) Homepage. Available: `http://www.eda.org/svwg/`

SystemVision. Mentor Graphics Corp. Product Information. Available: `http://www.mentor.com/system`

Vlach, J., and K. Singhal (1994). *Computer Methods for Circuit Analysis and Design.* New York: Van Nostrand Reinhold.

Chapter 4

EARLY PREDICTION OF CONDUCTED-MODE EMISSION OF COMPLEX IC'S

Anne-Marie Trullemans-Anckaert[1], Richard Perdriau[2], Mohamed Ramdani[2] and Jean-Luc Levant[3]
[1]*UCL-DICE Louvain-la-Neuve Belgium,* [2]*ESEO Angers France,* [3]*ATMEL Nantes France*

Abstract A new design methodology is presented for predicting the conducted-mode emission generated by an integrated circuit. Using the Integrated Circuit Electromagnetic Model (ICEM) developed by the International Electro-technical Commission (IEC), the influence of the internal power supply distribution is modeled, and the sensitivity to design options or external factors such as supply voltage variations may be studied. Using ICEM models written in VHDL-AMS leads to efficient simulation, from the early steps of the design process, of self-perturbation and self-immunity of a complex integrated circuit. These ICEM models may be part of an IP-block definition, preserving confidentiality. Providing an early stage information on the EMC quality of the chip facilitates the way to a first-time working silicon. A full 8-bit micro-controller with core, memories and I/O blocks, from an existing industrial design, is used to validate the methodology.

Keywords: Electromagnetic compatibility, ICEM, VHDL-AMS, prediction, modeling, simulation.

1. Introduction

Early performance estimation and quality validation remains a key concern in the design of complex IC's. IP's definition and design reuse solve partly that problem for area, delay and even power, but characteristics like Electromagnetic compatibility (EMC) compliance is rarely addressed. Industrial designs are directly concerned with electromagnetic compatibility, particularly for portable equipments: an electronic system must be certified for given emission and susceptibility levels. Traditionally EMC compliance was only con-

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 55–67.

sidered at the board level. This includes of course radiating effects but also simple supply transients: the di/dt of supply currents will directly pollute the environment. And the higher complexity, lower dimensions and higher switching rates of modern chips will produce higher spike density on the supply rails. It becomes a real necessity to be able to estimate, from the first steps of the design process, the effect of architecture choices on EMC properties.

Till recently, the classical approach to this problem was to measure produced chips, generally in packages, to certify the design. The only correction action may be to choose another package, change some lumped elements or even re-design the chip, with a new silicon, but without any real help for the chip designer. Only recently [Steinecke et al., 2004] some comparison between models and measures, using gate level models was presented. But of course this leads to huge simulations, and requires to know the precise gate structure and routing details at the time of each simulation.

The focus of this research is to address the problem at the architecture level, as early as possible during the design process, to efficiently estimate current supply transients of a full chip, in a given package. At this level, each block is better handled as a macro-function, without precise detail of the internal structure, even if the structure is already frozen, as for example when reusing IP's. Therefore we need an EMC model for each basic block. The model chosen is directly inspired from the ICEM model [IEC EMC Task Force, 2001]. Classically, this model is derived from measures on a packaged chip in activity, and is used to analyze the effect of the package itself on the CEM performances. Defining ICEM models at the level of the macro-function basic blocks, and assembling them in a mixed-mode simulation environment, we got a full chip CEM model. As for the classical ICEM model, the basic block CEM models are constructed to model the activity dependent current supply transients of the particular block, the digital functionality itself being modeled by classical VHDL code. The full chip simulation is then run in a VHDL-AMS environment, to allow complex stimuli to be applied, corresponding to a given digital activity. Interconnections of blocks are modeled by lumped parasitic elements, according to place and route of the blocks. To validate this model approach, the results were compared to measures in the precise case of an industrial chip, an 8-bit micro-controller. The results gives good correspondence, at reasonable computer cost.

2. Modeling IC conducted emission

The ICEM proposal developed in 2001 [IEC EMC Task Force, 2001] was developed to model the effect of parasitic elements of board, package and chip itself, on the spike shape of supply currents, and so helps to analyze the electromagnetic compatibility of a chip in its environment, in the domain of con-

ducted emission. The parameters of the model are obtained by standard measures [IBIS 4.0] on a chip after foundry.

Figure 4.1 gives the principle of the measure, along with the equivalent model which is extracted. In this model, the activity of the chip itself is mod-

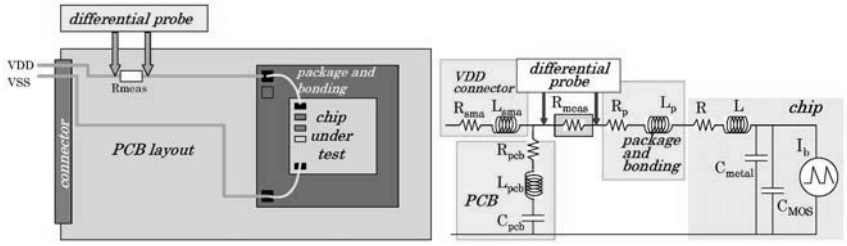


Figure 4.1. Principle of the ICEM measure and equivalent model

eled by a complex switched current source I_b, and the internal power distribution is modeled with lumped elements, R, L and C_{metal}. C_{MOS} globalizes the capacitance effect of the active elements. The power supply connector is modeled by R_{sma} and L_{sma}, and the PCB wiring by R_{pcb}, L_{pcb} and C_{pcb}. This approach uses a simple lumped model[1], limited here to the VDD rail effect. R_{meas} is a small value resistance added to measure the supply current with a differential probe. It should be noted that the I_b current is not directly measurable, but the values of the lumped elements can be derived by special analysis techniques [Levant et al., 2002]. This approach is an efficient way to analyze the effect of supply decoupling or a particular package on di/dt for example, and is actually used in the final step of a PC board design.

3. The proposed methodology

In a top-down approach based on design reuse, the design itself is viewed as an assembly process of well defined structures, characterized in terms of silicon area, signal delays and power consumption, but generally with no information on the internal details, for confidentiality reasons. Moreover, considering all the details in a simulation will lead to excessive computer cost. VHDL has solved part of the simulation bottleneck for digital systems, and gives good evaluation of signal delays. Some extensions allow the estimation of power, but no information is generally available in the digital domain for transients on the supply rails. One objective of this research is to facilitate the full chip simulation of supply transients, using high-level VHDL-AMS models for the basic blocks, and ICEM models for the floor plan and package parasitic elements.

The global model is build by assembling the basic block models: each block is modeled with an ICEM model representing its internal activity and parasitic elements, and interconnections of the blocs are modeled by lumped elements. Of course here the basic blocks are macro-functions which may be very complex: these are classically of the level of complexity of what is stored as memory or core IP's in an IP library.

To obtain the ICEM parameters of a block, a good practice may be to simulate this block as a SPICE model, in the particular environment of a given chip (or any realistic environment), the rest of the chip being modeled by pure digital VHDL, in order to stimulate the block with realistic patterns. The ICEM model is then derived by fitting the current source parameters to match the time transients of the supply currents. Passive elements are mainly metal line and parasitic MOS capacitances, which may be obtained from the SPICE net list, knowing the internal structure. This structure is then summarized in a lumped model, masking its details.

This job has to be done only once, and the result is stored in the IP library, within the IP model. If the block itself is structured, a structured model may be similarly derived. The practical example here after will give some strategy to derive particular models.

On the top of the full chip, a lumped model, derived from measures, is added for the package and the printed board. Every switched current source is parametrized by the digital stimuli of the block, and a VHDL-AMS model is used to model the current source. The interest to use a VHDL-AMS model for the transient current of a given digital part is to produce an activity model driven by the actual applied signals, generated by simple VHDL digital models, which are efficient in simulation. This leads to a light mixed-mode model for complex chips, just fitted to the exact desired characteristics.

4. A practical example

The example of a particular micro-controller, the 8-bit μC 80C51 'VIPER' from ATMEL, in a 0.35μ technology [Perdriau et al., 2002], is used to present the proposed methodology. Figure 4.2 shows the 4 main blocks of the internal structure of this μC: the CPU core, the EEPROM memory storing the program code, the SRAM memory for data, and the I/O drivers.

Figure 4.3 shows the VDD supply current measured in four different operational modes, from which spectral characteristics can be derived. These modes are chosen to see the effect of the different architectural blocks on the supply consumption and transients: in the RESET mode, only the CPU core is active; in the NOP mode, the opcode address decoder is added; in the instruction RRA (rotate right accumulator) the ALU is active; the MOV instruction uses the address generator and the SRAM.

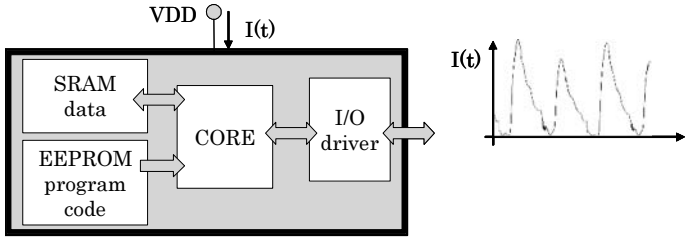


Figure 4.2. Example of IC structure under test for ICEM measures

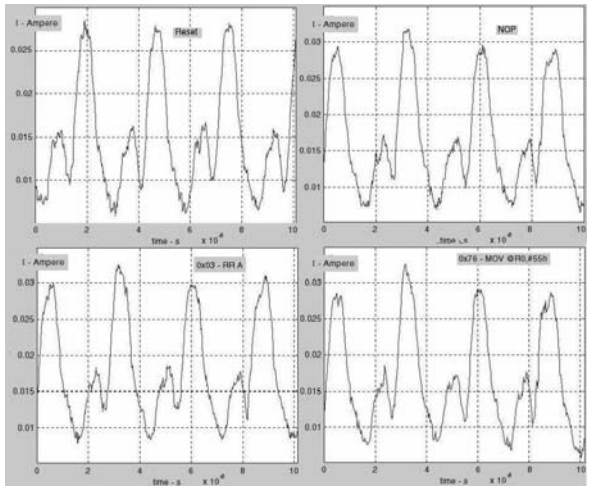


Figure 4.3. Measurement of the VDD supply current of an 8-bit μcontroller in 4 activity modes (8-bit μcontroller 80C51 'VIPER' from ATMEL, in a PLCC 44 pins package, techno 0.35μ)

From these experiments, it is clear that the ALU action and the memory accesses are negligible, and that no instruction dependence is visible : only very little spikes or offset changes can be observed. The main reason is probably that in the case of such a CISC processor, all these spikes come from the clock tree distribution, the ALU and the memories being much more slower. So the first idea to focus on the CORE activity, and the I/O drivers, in order to derive a model for this internal activity, which will be used to build the full chip model.

A general global model (Figure 4.4) is used to simulate the full chip for EMC characterization. Each basic block is here represented by a transient current source which models its dynamic internal activity. This can be obtained from full SPICE-level simulation, which is of course very time-consuming. The main blocks to be modeled as VHDL-AMS models are the CPU Core and

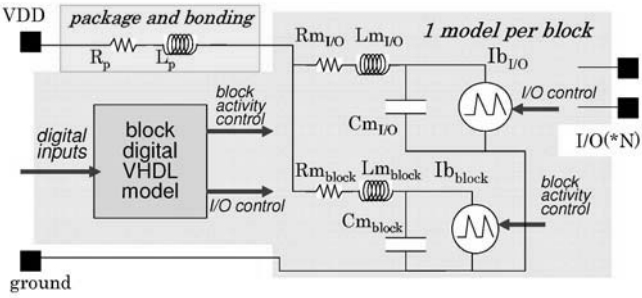


Figure 4.4. Block-based model of an IC for ICEM characterization

the I/O modules, which are dominant here. Passive components are derived from floor plan and P&R information, or could be simple prospective values. To derive the model, every block is successively simulated in its environment, the others being simulated by simple digital VHDL models. So the analyzed block is simulated in a realistic environment, corresponding to the complex behavior.

Example of a general block model:

```
ENTITY ICEM_IP_Model IS
 GENERIC (Tr :  real); -- rising time
 PORT (controls :  IN std_logic; -- * N inputs
       TERMINAL Vdd, Vss :  electrical);
END ENTITY ICEM_IP_Model;
ARCHITECTURE ICEM OF ICEM_IP_Model IS
 -- CONSTANT definitions for internal R, C and L
 TERMINAL Vddgen :  electrical;
 QUANTITY Vb ACROSS Isw,Ic THROUGH Vddgen TO Vss;
 QUANTITY Vrl ACROSS Irl THROUGH Vdd TO Vddgen;
BEGIN
 -- Isw computation :  to be adapted
  Ic == (Cmos + Cmetal) * Vb'dot;
  Vrl == Rint * Irl + Lint * Irl'dot;
END ARCHITECTURE ICEM;
```

Each block is modeled by an entity connected between the internal Vdd and Vss rails (PORTs defined as electrical TERMINALs), with digital control inputs (PORT signals IN). GENERICs are used to have the input rise time delay as a parameter to the model. Local wiring parameters (R_m, L_m) and metal and MOS capacitances (global value in C_m) are here constants internal

to the architecture (GENERICs could also be used). The internal switching current source has to be adapted to the particular blocks modeled. Here too, parameters may be internal constants or GENERICs.

5. The CPU core model

From a complete SPICE simulation of the core in RESET mode, an event-driven, piecewise linear (PWL) model, matching the simulated waves, is derived [Levant et al., 2002], and used to model the supply current activity of the core, in response to the digital inputs.

Example of the structure of the entity and architecture definitions:

```
ENTITY CoreGenerator IS
  GENERIC (Tr :  real); -- rise time for control signals
  PORT (XTAL1A : IN std_logic;
        TERMINAL Vdd, Vss :  electrical);
END ENTITY CoreGenerator;
ARCHITECTURE a OF CoreGenerator IS
  -- CONSTANT and local SIGNAL definitions
BEGIN
  PROCESS
  BEGIN
    LOOP -- for waiting driving signal XTAL1A
      -- compute PWL parameters and activates Tstart
    END LOOP;
  END PROCESS;
  IF domain = quiescent_domain USE
    -- starting values in DC
  ELSE
    -- compute currents
  END USE;
  BREAK ON Tstart;
END ARCHITECTURE a;
```

In this model, XTAL1A is a simple digital VHDL signal, but Vdd and Vss are VHDL-AMS terminals (electrical), to model the current activity. The generic map gives the particular rising delay of the signal, which is used in the model.

As expected, the internal current spikes produce external reduced spikes, due to the smoothing effect of packaging and supply decoupling. Compared to measures in RESET mode, this gives acceptable results (Figure 4.5): peak values and transition times, compared to the top left screen of Figure 4.3 are

in the same range, and very close. The main difference is the lack of the small additional pulse in each clock period, which comes from the clock driver. The simulation time goes down from 3 hours for the SPICE net list simulation to 4 seconds for the VHDL-AMS model.

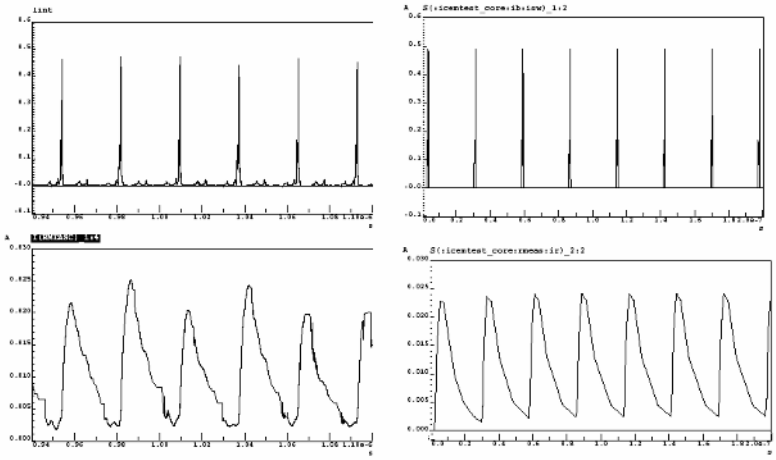


Figure 4.5. Core current in the RESET phase - Left: complete SPICE simulation: internal (top) and external (bottom) current - Right: VHDL-AMS model: internal (top) and external (bottom) current

6. The memory blocks

A VHDL-AMS model for the SRAM memory block was developed, and is detailed in [Perdriau et al., 2002, Perdriau, 2004]. The model is derived from the analysis of the internal architecture of the 1280-byte SRAM block from ATMEL. Separate models are given for the address decoder, which activity is address-dependent, and for the memory cells, which are not (only small differences are observed due to the output drivers). This allows to fit well with complete SPICE simulations, but in this application the spike current values are in the order of 1/20 of the CORE currents. Even if not really significant in this analysis, the VHDL-AMS model will be added to the standard VITAL model of the SRAM in the complete simulation. A simulation example of a memory access takes 2 seconds for this model, compared to 1 hour for the equivalent SPICE net list.

Example of the structure of the entity and architecture definitions:

```
ENTITY RAM1280Generator_h IS
  GENERIC (Tr :  real); -- rising time
  PORT (ADD : IN std_logic_vector(10 DOWNTO 0);
```

```
        DATA : IN std_logic_vector(7 DOWNTO 0);
        ME, WEN : IN std_logic;
        TERMINAL Vdd, Vss :  electrical);
END ENTITY RAM1280Generator_h;
ARCHITECTURE a OF RAM1280Generator_h IS
  QUANTITY Vb ACROSS Ib THROUGH Vss TO Vdd;
  CONSTANT DecPulseTiYZ:real_vector... -- Intensity vectors
  CONSTANT DecPulseIiYZ01:real_vector... -- 0->1 transition
  CONSTANT DecPulseIiYZ10:real_vector... -- 1->0 transition
  -- local CONSTANT and SIGNAL definitions
BEGIN
  ...
  PROCESS -- address decoder
    ...
  BEGIN
    LOOP
      WAIT UNTIL ADD'event;
      -- compute Hamming distances for X and YZ decoders
      PeriodStart := now;
      FOR n IN DecPulseTiYZ'low+1 TO DecPulseTiYZ'high LOOP
        Istartd <= ...; -- Current at start point
        deltaId <= ...; -- Current variation between points
        IF n = DecPulseTiYZ'low+1 THEN
          -- time for start and end points
        ELSE
          Tstartd <= ...; Tendd <= ...;
          WAIT FOR DecPulseTiYZ(n)-DecPulseTiYZ(n-1);
        END IF;
      END LOOP;
      deltaId <= 0.0; Istartd <= 0.0; previousADD <= ADD;
    END LOOP;
  END PROCESS;
  -- idem for X decoder and cells
  -- Current pulse generation memory cell activity
  IF domain = quiescent_domain USE
    Ib == Istartd;
  ELSE
    Ib == Istartd + deltaId*(now-Tstartd)/(Tendd-Tstartd)
    + Istartm + deltaIm*(now-Tstartm)/(Tendm-Tstartm);
```

```
END USE;
BREAK ON Tstartd, Tstartm;
...
```

SPICE analysis of the EEPROM leads to even lower values: access to one memory cell gives spike values of the current in the order of 1/35 of the value for the same operation on an SRAM cell. For this part, only a digital simulation, based on the VITAL model, will be used, and produces the digital signals driving the rest of the chip.

7. The I/O drivers

The I/O buffers are modeled by an IBIS [IBIS 4.0] model, adapted for ICEM simulations. The main changes are that, in the buffer, the MOS drain currents are functions of gate/source and gate/drain voltages, and must take into account the non ideal transient times of the control signals generated by the core. Moreover, concerning the supply currents, the limiting diodes must be considered, in the case of excessive I/O voltages. Values obtained may be in the order of the CORE current spikes, and are very dependent on the I/O data itself.

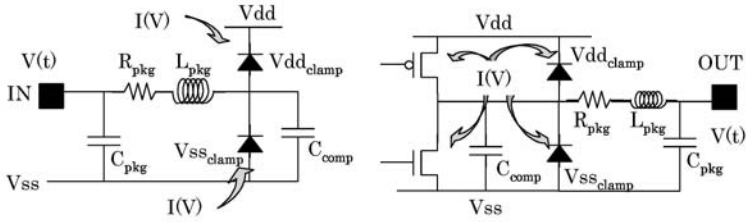


Figure 4.6. IBIS I/O Models: input (left) and output (right)

The VHDL-AMS model of the buffer [Perdriau, 2004], with generic parameters, implements table lookup and interpolation functions for MOS and diode characteristics. Parameters are the MOS tables, IBIS passive parameter values, and rise and fall times of the control signals. The MOS characteristics are split into 3 regions: one triode and two saturated regions.

Example of the structure of the entity and architecture definitions:

```
ENTITY totempole IS
  GENERIC (-- generic for MOS and diode parameters);
  PORT (io :  in std_logic;
        TERMINAL Tvdd, Tvss, pad :  electrical);
END ENTITY totempole;
ARCHITECTURE behavioral OF totempole IS
```

```
  QUANTITY vpmos ACROSS ipmos THROUGH Tvdd to pad; -- PMOS
  QUANTITY vnmos ACROSS inmos THROUGH pad to Tvss; -- NMOS
  SIGNAL in_realP, in_realN : real := 0.0;
  QUANTITY in_rampP, in_rampN : real := 0.0;
BEGIN
  -- NMOS and PMOS grid signals
  -- and quantity with rising time
  in_realP <= 1.0 WHEN io = '1' ELSE 0.0;
  in_realN <= 1.0 WHEN io = '0' ELSE 0.0;
  in_rampP == in_realP'ramp(tr,tr);
  in_rampN == in_realN'ramp(tr,tr);
  -- Currents from the parametric MOS tables
  ipmos == interpolate_mos(...);
  inmos == interpolate_mos(...);
END ARCHITECTURE behavioral;
```

Using these pad and buffer models, it is now possible to model the influence of the input and output of the clock driver. A real clock signal is generated, with a rise time of 3 ns, as in the experimental conditions. Putting these models directly on the supply rails results in the external current shown at Figure 4.7.

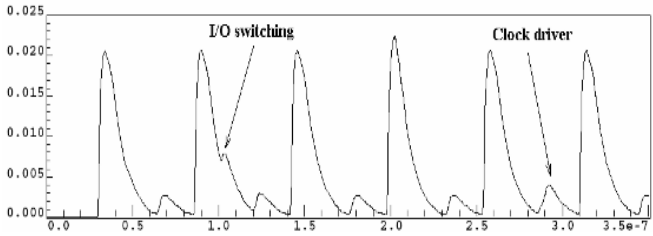


Figure 4.7. External current with I/O switching and clock driver

With this added I/O models, the intermediate pulse appears, correlated with the clock driver activity. On this simulation, the effect of I/O switching on the supply current is also visible. Their rise time and period are higher, and will produce low frequency perturbations. A complete simulation of the I/O ports in the system environment takes no more than 10 seconds.

8. Conclusion

The basic idea of the methodology presented here to address the ICEM evaluation, is to model the dynamic activity of each block with a transient current

source driven by the digital inputs, and, adding lumped elements for the passive components, focus on the supply rail currents of the whole chip, using high level mixed-mode simulation. These models have to be developed for each of the blocks which have a significant impact on the global behavior, in terms of supply currents.

Detailed SPICE simulations requires the knowledge of the internal structure, including precise implementations and place and route information, only accessible by parameter extraction on the mask data. This can only be done by the original designer of the block itself. From the experiments presented here, it seems possible to define a higher level model, in VHDL-AMS language, which could be put in the library of the IP block, for the final designer. Integrated in signal integrity tools, and supplied to PCB and system designers, these will allow fast board-level simulations of parasitic conducted emission.

The system designer may then even specify the expected EMC performance of the IC and verify the sensitivity of the design to previously described parameters, speeding up the design flow and supplying a "correct-by-design" circuit.

The example chosen here is a CISC μcontroller, which dynamic activity is mostly dominated by the clock tree activity, and not by the actual program running. For a RISC processor for example, it should be mandatory to implement, in the VHDL-AMS model of the core, an activity model for the input and output instruction bus activity for example. Defining activity classes based on instruction classes is a possible approach to this problem. The analysis of the effect of pipelining of instructions are future possible extensions of this work.

This work was done in close cooperation with ATMEL, to analyze the CEM properties of a chip after foundry in order to validate the methodology. It was then used by ATMEL during the re-design of a bad run of another project [Levant et al., 2004]: the good predictions of this modeling approach would have given a first-time correct chip, thus reducing foundry costs.

Standardization of this approach will be based on the new version of the IBIS model (version 4.1), which includes I/O VHDL-AMS descriptions, and on the work of the ICEM normalization group of UTE.

Notes

1. The model used here is limited to a lumped R-L-C approach, acceptable for this precise IC. For higher clock rates and future chips, a transmission line model will be necessary.

References

IBIS (I/O Buffer Information Specification) version 4.0. ANSI-EIA. `http://eda.org/pub/ibis/ver4.0/ver4_0.pdf`

IEC EMC Task Force. *IEC62014-3: Integrated circuit electromagnetic model*. Draft technical report, IEC, November 2001. `http://intrage.insatlse.fr/~etienne/icemcdv.PDF`

J. L. Levant, M. Ramdani, and R. Perdriau. *Power supply network modeling*. EMC Compo 2002, pp. 75-78, November 2002.

J. L. Levant, M. Ramdani, and R. Perdriau. *PLL Jitter Improvement using the ICEM Model*. EMC Compo 04, pp. 129-137.

R. Perdriau, D. Lambert, A.M. Trullemans, M. Ramdani, and J.L. Levant. *A VHDL-AMS simulation methodology for transient supply current extraction*. EMC Compo 2002, pp. 99-104, November 2002.

R. Perdriau. *Méthodologie de prédiction des niveaux d'émission conduite dans les circuits intégrés, à l'aide de VHDL-AMS*. Thèse de doctorat UCL, mars 2004.

T. Steinecke, H. Köhne, M. Schmidt. *Modeling, Simulation and Measurement of Conducted Emissions on Chip Level*. EMC Compo 04, pp. 21-26.

Chapter 5

PRACTICAL CASE EXAMPLE OF INERTIAL MEMS MODELING WITH VHDL-AMS

Elena Martín[1,2], Laura Barrachina[1,2],Carles Ferrer[1,2]

[1]*Institut de Microelectrònica de Barcelona (IMB-CNM, CSIC)*

[2]*Departament d'Informàtica, Universitat Autònoma de Barcelona*
Campus Universitari, Bellaterra 08193 (Barcelona), Spain
Telephone: (+34) 93 594 77 00 (ext 1205,1206)
Fax: (+34) 93 580 14 96

Abstract Considering the evolution of the development of Sensors and Actuators a different learning curve between MEMS and integrated circuits is found. One of the possible solutions is to extend the use of design and simulation tools and languages design to integrate electromagnetic, thermal systems, etc. The modelling becomes a complex task, due to the fact that each component of the system can belong to different physical domains and can present a different better representation level. After that phase, the fabrication has to be done to assemble all the parts (device, analogue, mixed and digital) to finally do the test and qualification. The example that will be presented is a modelling of a Smart Sensor using VHDL-AMS. The system is composed by an accelerometer, its associated output circuitry and a sensor's bus interface.

Keywords: Modelling, VHDL-AMS, MEMS, Accelerometer

1. Introduction

Present world market on research and development on Sensors and Actuators is about US $\$ 10^{10}$. This is an "insignificant" amount of money compared to US $\$ 30\ 10^{10}$ market for Microelectronics. In the same way, the learning curve of MEMS(Micro Electro Mechanical Systems) does not follow the growth for integrated circuits. One of the possible solutions could be the ex-

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 69–84.

tension of the use of integrated circuit design to Microsystems design thanks to the addition of new design and simulation tools that help the integration of electromagnetic systems, thermal systems, etc. with the integrated circuits in only one design EDA environment.

This will permit to simplify the design process in order to obtain a more reliable Microsystems and to reduce the design time and cost. A simplified **design flow for mixed models**, will start from initial specifications including environment and technical characteristics [Ferrer, 2003].

The next step is **partitioning into basic components** (include sensors, actuators, analogue and digital circuitry) to design an abstract structure that meets the initial requirements. The modelling becomes a complex task, due to the fact that each component presents a better representation level (system, device, physical and process), in the case of MEMS also particular physical domain should be considered. Depending on their nature, energy and signal domains values will be calculated. The natural values are obtained using the concepts of continuum theory which couple different domains in terms of partial differential equations [Voigt, 1998].

However, despite the computational power of modern computers and the availability of highly efficient finite-element codes and related numerical methods, it is hardly feasible to solve the model equations in their continuous form for realistic Microsystems in their full size and complexity. Therefore, to keep the computational effort for such problems within acceptable limits, the proper and **adequate level of simulation** is analog circuit simulation based on a **network theory** which, on the one hand preserves the basic conservation laws expressed in the continuum models and, on the other hand, reduces the set of state variables to a number that is still manageable with one of the circuit simulation tools as they are commonly used in the **microelectronic community**.

After that phase, the fabrication has to be done to assemble all the parts to finally do the test and qualification.

The example that will be presented is a **modelling of a Smart Sensor using VHDL-AMS**. The system is composed by an accelerometer, its associated output circuitry and a sensor's bus interface.

2. Design Methodology

Methodology for integrated digital circuits design is achieved with a **top-down methodology**, associated to standard language modelling languages like VHDL. It can be extended for the use to Microsystems design thanks to the analogue extensions of new languages (like VHDL-AMS for VHDL), that allow to apply similar techniques to design and model systems.

A design flow methodology for mixed models will be based in methodologies of scalable functionality, robust design and feedback techniques. It will

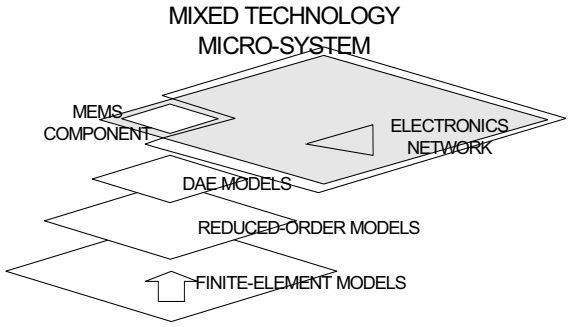


Figure 5.1. Model hierarchy.

start from **initial specifications**, including environment and technical characteristics.

The next step is **partitioning** into basic components. The Digital design can be automated made through the use of VHDL hardware description language, while the mixed-signal design can be developed using the new VHDL-AMS extension. After that phase the fabrication has to be done to **assemble** all the parts and finally do the **test and qualification**.

Including **MEMS** design in this general flow design methodology is a new challenge [Hanna, 1999] due to the fact that different physical domains and levels of abstraction have to be considered, from component to system, through subsystem, see Fig 5.1. The development of a design hierarchy allows the designer to mix levels of abstraction to observe and evaluate interactions between interdependent subsystems. Systems described hierarchically using multiple mixed levels of abstraction can employ different combinations of top-down and bottom-up techniques.

There are different ways to describe a microsystem, like as a geometrical structure or like a mathematical description. The **geometrical structure** is based on generalized Kirchoffian networks which are based on parameterizable analytical elements models. The **mathematical description** is based on the discretization of the system using DAE, ODE or algebraic equations and can be summarized in two: numerically generated based in order reduction that produces reduced system matrices and behavioral models obtained with black-box models [Schwarz, 2001].

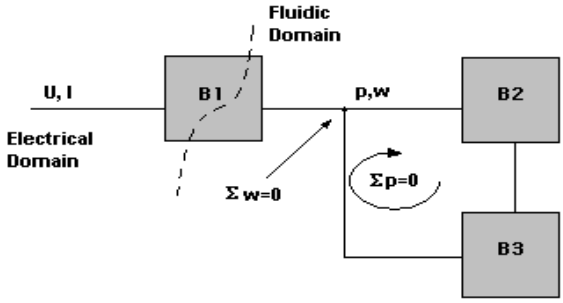


Figure 5.2. Example of Kirchhoff's in different domains [Voigt, 1998]

2.1 MEMS Behavior Characterization

Developing component models for MEMS represents a great challenge due to the complexity of modelling their behavior [Dewey, 1999], [Schleger, 2003]. Their behavior can not be considered a simple addition of separate mechanical (fluidic, etc.) and electrical behavior, because they are a simultaneous combination. New component modelling, analysis and design techniques are required to obtain it.

Generalized networks can be considered in MEMS modelling because many physical quantities are compared to flow or difference quantities and generalized Kirchhoff's laws can be applied. To obtain it, large systems can be interpreted as decomposition into network basic elements, as can be found in [Voigt, 1998]. This network concept is valid in many and different domains, like electrical, fluidic, mechanical, etc.. The network elements definition is based on generalized Kirchhoff's laws. These laws are basically two, the first is the mesh law and the second is the node law, see Fig 5.2.

Another way to obtain the model is using **order reduction**. Modelling strategy the real system can be described using partial differential equations for the entire system, producing reduced system matrices. The last way is to obtain behavioral models with black-box models, coming from simulation results in time or frequency domain. Once the model is developed and included in the complete system, the simulation and the optimization process must be considered. The simulation system must interact using an optimization/simulation algorithm thanks to the setting of parameters and adjustment of them.

A more general **MEMS' modelling approach** is based in the **creation of a model ready for simulation coming from a generic model**. The simula-

tion results will be obtained depending on simulation parameter sweep, always comparing the initial results with the desired results. Once the comparison is done, the initial parameters are changed to consider a more real model. There are many simulators that can be used, like the following ones. For circuit simulation: Saber, ELDO, Spice, for control systems there is Simulink/Matlab, for FEM there is ANSYS or FLOTHERM and for Math-Codes, there is Mathematica and C or Java routines.

All modeling tools that can be considered and multi-domain libraries are usually very incomplete. Standardized modeling languages like VHDL-AMS will be supported by many system simulators.

2.2 VHDL-AMS

Considering a higher level of abstraction in modeling at device level, is like equaling MEMS component modeling to Very Large Scale Integration (VLSI) component modeling. It will be the step from three-dimensional electrical and mechanical effects into a two-dimensional network of lumped parameter elements governed by a system of ordinary differential and algebraic equations (DAEs). **Circuit network topology defines the global structure of the microelectromechanical equations**.

To develop the modeling characterization including differential and algebraic equations a new modeling language can be used. **VHDL-AMS** is a new IEEE standard (IEEE 1076.1(1999) focused to the analogue and mixed-signal modeling and simulation. It appears as an extension of **VHDL** (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language), but enabling description of continuous time systems. The suffix AMS of the VHDL extension means Analog and Mixed Signal, considering that a mixed signal system presents coupled time-based or coupled energy-based models. It allows the description of non-electrical (like mechanical), or electrical signals and also the combination of discrete and continuous time language constructions and can also be used to describe a system that can be simulated at different levels of complexity. The commercial EDA tools permit also the co-simulation of VHDL-AMS, VHDL, SystemC, SPICE, ELDO,...

The main characteristics of this language are:

- VHDL-AMS provides a constructs for defining sets of **simultaneous ordinary differential and algebraic equations** (DAEs) at any level of abstraction and support both conservative and non-conservative systems. This implies that any physical system that exhibits strictly analog behavior or a mixture behavior can be modeled and simulated using VHDL-AMS. VHDL-AMS uses a declarative dynamical model of tightly-coupled simultaneous relations influencing each other's solution by linked unknowns.

- The unknowns are called **quantities** and are a new class of objects of VHDL and they take their values of solving the set of simultaneous ordinary differential and algebraic equations.

- These models considered to be in a real world can be described using different **domains**. Each domain is defined when are specified by 3 objects: **nature**, **across** and **through**. The possibility of defining new domains gives the opportunity of modelling new systems that can be described in a compact way.

3. Inertial MEMS

One MEMS-based design accelerometer solutions is used to develop the modelled microsystem. It is a **Piezorresistive accelerometer**, its associated output circuitry and the interface to a sensor's bus.

The accelerometer that is going to be used was manufactured in CNM [Collado, 2003], but presented some problems because of general **temperature dependence**. One of the proposed solutions was to develop a complete **"theoretical" study** to check what kind of sources provokes the problems, to propose solutions and implement them. Now, the first results obtained in this study will be presented in the following sections.

3.1 Accelerometer

The accelerometer is a **MEMS's stress-measurement based in the Piezorresistive effect**. One of the main features that presents is a large temperature dependence of the piezoelectric coefficients. The accelerometer can be described as cantilever design in a SOI wafer and it can be mechanically described as a second order system.

The value of the **main parameters** of the dynamic system is obtained from **technological parameters**, the static and the dynamic characterization. Due to the higher stress located in the lateral bridges, the piezoresistors are located there to obtain a greater output signal from the Wheatstone bridge. Even so, the signal obtained has an amplitude in the order of mV, fact that makes taking special care with the effects of noise in the final signal.

The strategy that was followed to obtain the final model of the accelerometer was to consider all the different kind of representations, see Fig 5.3. The **different representations** were the following ones:

A **Behavioral Description**, based on experimental measurement . These results present a linear relation between the applied acceleration and the output voltage obtained of the accelerometer. It doesn't consider any physical process, like the existence of the piezoresistance, which presents high temperature dependence. The development of this **"basic**

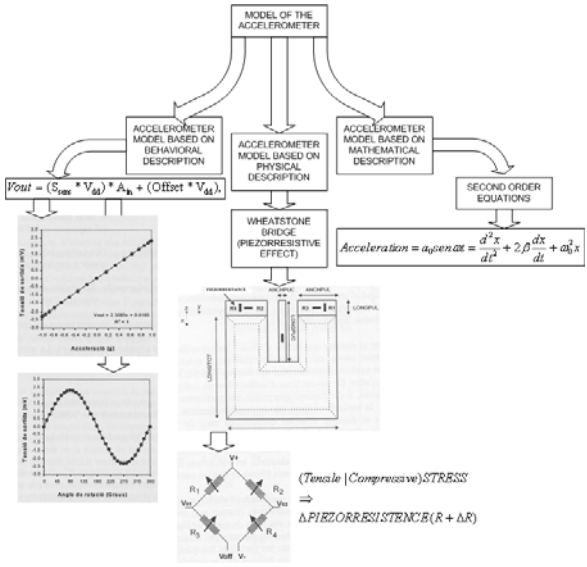


Figure 5.3. Different models developed for the accelerometer.

and simple" model is to test the complete output circuitry at an ambient temperature, without considering any variation at temperature (like self-heating, temperature distribution in the accelerometer, or different ambient temperature).

This model can also allow the develop of a **more complex system** like several sensors with a distributed architecture, due to the simplicity of the single element. One of the problems that this simulation presents is that can take too much time if a complex description of the components is used. So in order to simulate a more complex global system, it is more useful to work with the behavioral descriptions.

B **Physical Description**, based on Piezorresistive effect. It describes the model as a **Wheatstone bridge with 4 variable resistances**. These resistances are a region of a semiconductor type p over a type n. The main characteristic of a piezoresistance is that presents a variation on its value if it suffers a stress. The model includes the relation between acceleration and the variation of the resistance, and also the relation between the variation of the resistance and the final output value of the voltage. The physical description is showed in Fig 5.4. The system described here is excited with an acceleration, which is given as a sinusoidal wave. The next step is calculating the stress that piezoresistances suffer due to this acceleration. This calculation is done by applying a linear relation

```
library ieee;
library work;
library disciplines;
use ieee.math_real.all;
use disciplines.electromagnetic_system.all;
use disciplines.physical_constants.all;
use disciplines.thermal_system.all;
use disciplines.kinematic_system.all;
use disciplines.rotational_system.all;
use disciplines.fluidic_system.all;
use work.resist.all;
use work.accel.all;
use work.all;
-------- Modelling of the accelerometer D2U-2.5g. The experimental -----
-------- results used here have been obtained from the tesis "Un nou ---
-------- encapsulat multixip per a acceleròmetres piezoresistius"-------
entity wheatstone_bridge is
               generic
                              (r0            :real          :=1888.0);
               port
                              (terminal voltage1,voltage2: electrical;
                               terminal accelerat: acceler);
end wheatstone_bridge;
architecture total of wheatstone_bridge is
               terminal voltagedd: electrical;
               terminal deltar1,deltar2: res_v;
               terminal sigma_t,sigma_l: kinematic;
               quantity vdd across i_alim through voltagedd to electrical_ground;
               quantity vout across voltage2 to voltage1;
               quantity v1 across i1 through voltagedd to voltage1;
               quantity v2 across i2 through voltagedd to voltage2;
               quantity v3 across i3 through voltage1 to electrical_ground;
               quantity v4 across i4 through voltage2 to electrical_ground;
               quantity delta1 across deltar1 to gnd_r;
               quantity delta2 across deltar2 to gnd_r;
               begin
                              applied_acceleration:         entity acc_gen1(behav)
                                                            generic map(freq=>50.0,Ampl=>1.0,Delay=>0.0)
                                                            port map(accelerat);

                              applied_stress:entity accstress(total)
                                                            port map(sigma_t,sigma_l,accelerat);
                              rvariationA:                  entity accpiezo(total)
                                                            port map(kinematic_ground,sigma_l,deltar1);
                              rvariationB:                  entity accpiezo(total)
                                                            port map(sigma_t,kinematic_ground,deltar2);
                              vdd == 5.0;
                              v1 == i1*(r0+delta1);
                              v2 == i2*(r0+delta2);
                              v3 == i3*(r0+delta2);
                              v4 == i4*(r0+delta1);

end architecture total;
```

Figure 5.4. Physical Description of the accelerometer.

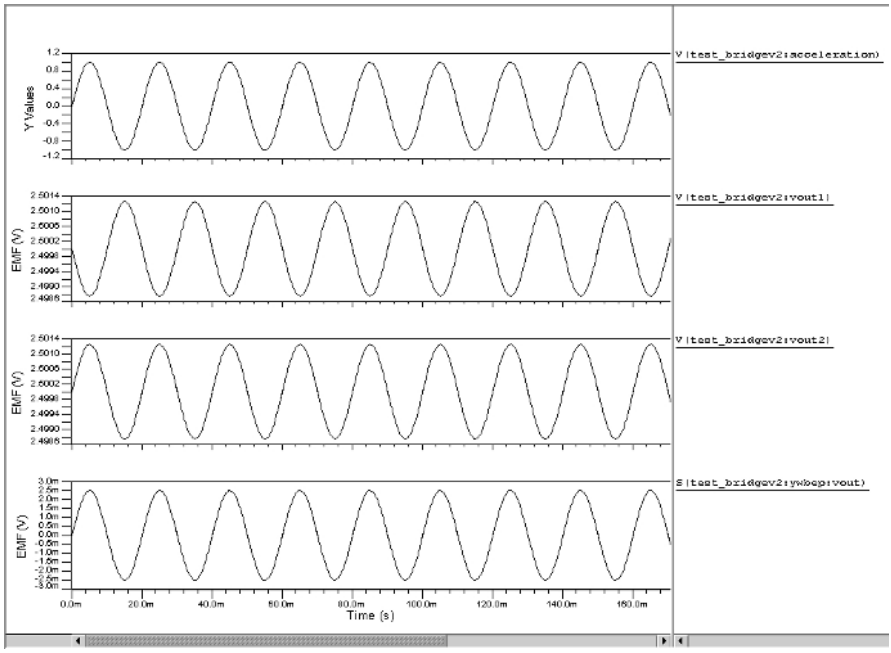


Figure 5.5. Simulation of the physical description of the accelerometer.

between acceleration and stress, which has been obtained thanks to experimental measures. **The longitudinal and transversal stress applied to the resistances is calculated given the acceleration**. In this configuration of Wheatstone bridge, there are two piezoresistances placed along the direction of the applied acceleration, and the other two piezoresistances placed transversally to this direction. So for a pair of parallel piezoresistances, only the stress in one direction will affect its value. Due to this fact, two different variations in the resistors value have been calculated depending on the direction of the applied stress. This feature can be seen in entities **accpiezo**. Finally, it is only needed to use the characteristic equations of the Wheatstone bridge electrical components to solve the system. The simulation of this model is showed in Fig 5.5.

C **Mathematical Description** based on the operation principle of the Piezoresistive accelerometers. The accelerometers, as an acceleration sensor, can be described as a system composed by a mass-spring. It can be described by the second Newton law including the damping force. The value of all constants included in the equations can be extracted from the physical values and the experimental measurement results. The capacity of VHDL-AMS to describe systems including derivatives and integrations allows the possibility of obtaining the value of the displacement due to an applied acceleration. This description is useful in the **dynamic characterization** of the accelerometer.

3.2 Output Circuitry: General Description and Models

A **general description** of the **output circuitry** is included in the intelligent MCM substrate. It presents three parts, the first one **corrects the offset in the Wheatstone bridge** and the second are some **temperature sensor** element incorporated to extrapolate the temperature of the sensor. The third part is composed by the **output circuitry** that should **amplify the signal** coming from the accelerometer.

Offset effects in a Wheatstone bridge are common because, although the resistances are generally located close, sometimes its value can be slightly different, and also there may be an initial stress on the system. The different value of the resistances is due to small inhomogeneities of impurities implantation process along the wafer. To correct this imperfection, the resistances are placed as close as possible without modify sensor features. About initial stresses, they are a consequence of the existence of different materials and the temperature phases in the fabrication process. These two facts provoke the undesirable offset in the final signal. To solve this problem, when the accelerometer was built, resistances were place parallel to the ones of the Wheatstone bridge. So, if there was an important difference with the value of one resistance of the

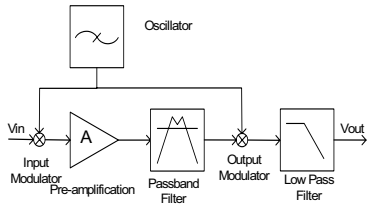


Figure 5.6. Schematic corresponding to the amplification circuit.

bridge, it could be used its parallel resistance to make up for **compensating its deviation just managing the connections**. This compensation has not been initially included because only one device will be described, although it will appear in any description including any high quantity of devices because it will highly affect the global performance of the system.

The next element contained in the intelligent MCM substrate was a **temperature sensor to extrapolate the response of the sensor at different temperatures**. These features have not been captured by the VHDL-AMS model yet.

The **models** developed for the the output circuitry consists in (see Fig 5.6) a **pre-amplification circuit based in Chopper Stabilization (CHS)**. This technique has been chosen due to the need of decrease the effect of noise in the readout circuitry. In this configuration the noise and offset is entering in the circuit after the first modulation. So, the frequency spectrum is displaced, and the noise, with lower frequency, does not affect so much the amplification.

The circuit is composed by two modulators (Input/Output, which are controlled by oscillator signal), a **pre-amplification**, a **pass band filter**, **oscillator** and a **low pass filter** (40dB/decade and 10 kHz Butterworth filter). All the circuitry that has been described in previous paragraph has been modeled using element models provided from ADVanceMS in the library CommLib and a general description of the elements can be:

- The **Oscillator** is composed by an astable multivibrator (square-wave generator) and a comparator. It was designed to obtain the filter resonance frequency and oscillator frequency as close as possible. The frequency of oscillation is defined by the relation between R1 and R2 and presents a value of 110 KHz.

  The modulator is composed by different logic gates (see Fig 5.7), and provides two square signals with a delay of 180 degrees and the frequency of the oscillator.

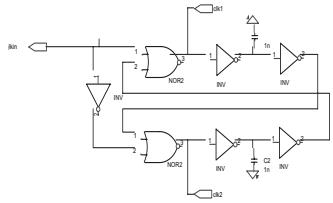


Figure 5.7. Schematic corresponding to the Modulator.

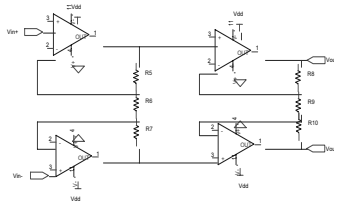


Figure 5.8. Schematic corresponding to the Pre-amplificator.

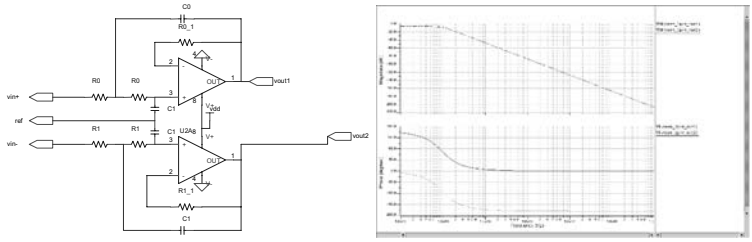


Figure 5.9. Schematic and simulation for the low pass filter.

- The **Pre-Amplificator** is composed by a differential amplifier with two amplification stages (see Fig 5.8). The gain that presents the operational amplifiers is 100dB.
- The **Low Pass filter** is a differential filter based on a 40 dB/decade Butterworth filter. The cut frequency of the filter is 10 KHz, and the model developed and the simulation results of the Filter can be seen in Fig 5.9a,5.9b.
- The **Band Pass Filter** is a differential filter based on a narrow band's band pass filter. The most important characteristics are the resonance

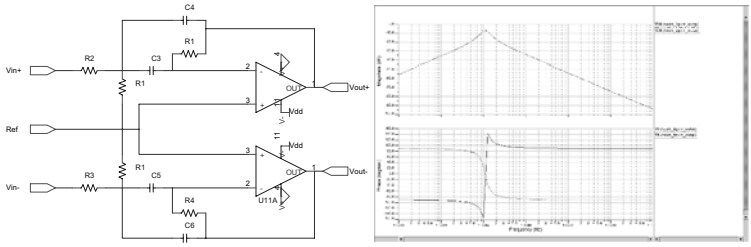


Figure 5.10. Schematic and simulation for the Band pass filter.

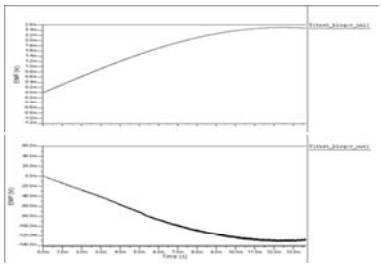


Figure 5.11. Simulation results obtained for the output circuitry.

frequency of 110 KHz and a 40 KHz bandwidth. These features can be seen in the results shown in Fig 5.10.

- The previous results have been obtained by running the simulation of the **complete output circuitry**. The excitation signal (see Fig 5.11) is a sinusoidal wave with the amplitude within the range expected by the accelometer. The output signal has a linear relation with the incoming signal, and amplifies it in two orders of magnitude.

3.3 Complete Circuit Model

The last part studied is the digital connection with a sensor's serial bus interface by using VHDL and allowing the possibility of modelling an intelligent sensor (Smart Sensor).

The models have been developed mainly using models given by the library of **ADVanceMS of Mentor**. The evolution of this models has been the following one: first use as many digital components as possible (like in the Modulator) to start changing all possible elements by analog models.

The addition of the VHDL-AMS model (MEMS and Output Circuitry) and the VHDL model (Sensor's bus), gives the possibility of obtaining previous

```
-------------------------------------------------------
library ieee;
library work;
library disciplines;
use ieee.math_real.all;
use disciplines.electromagnetic_system.all;
use disciplines.thermal_system.all;
use disciplines.kinematic_system.all;
use work.resist.all;
use work.all;
-------------------------------------------------------
entity wheatstone_bridge_etapa_ampl is

        generic
                (value_r:real           :=1000.0);
        port
                (terminal v_out: electrical;
                 terminal deltar: res_v);

end wheatstone_bridge_etapa_ampl;
-------------------------------------------------------
architecture total of wheatstone_bridge_etapa_ampl is

        terminal v_out1,v_out2: electrical;

        begin
        wb:     entity wheatstone_bridgev0(total)
                generic map(value_r)
                port map(v_out1,v_out2,deltar);
        etap:   entity amplifier_total(total)
                port map(v_out1,v_out2,v_out);
end architecture total;
-------------------------------------------------------------------------------------------
```

Figure 5.12. Behavioral description of the accelerometer.

simulation and modelling results before continuing the development of the microsystem and also giving the possibility of enlarge this model to include the thermal description.

A final model can be done by combining the accelerometer model based on the architecture of the wheatstone bridge and output circuitry model. Two different entities have been used in this model (see Fig 5.12). The first instantiation refers to the accelerometer described as a wheatstone bridge. The effect of piezoelectricity can be described by varying the resistances a value of increment of them. The second instantiation refers to the output circuitry explained in the previous section.

3.4 Results and Discussion

The models that have been used have to be improved to include a more detailed description of the system, and more precise results can be obtained. But several problems have appeared during the modelling and simulation of the circuits. The use of quantities across and through means in practice that problems related with non invertible matrix may appear, as well as convergence problems in the iterations of the simulations. Particularly, the modelling of the multivibrator in the output circuitry has involved difficulties to obtain correct values in frequency and amplitude in the output signal. However the simula-

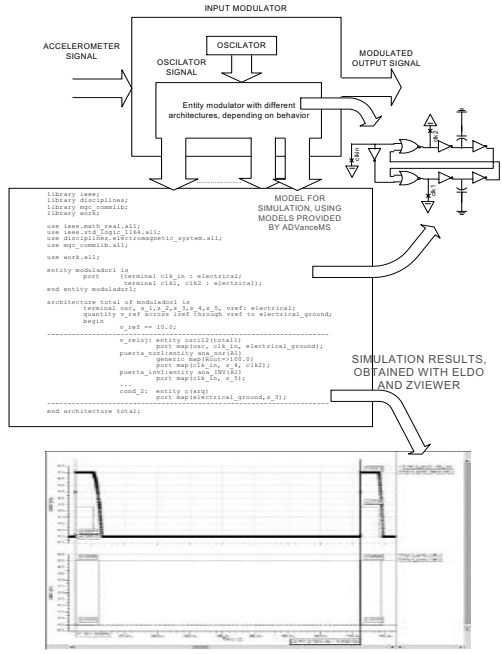


Figure 5.13. Example of results corresponding to a subsystem of the output circuit.

tions of the other modules of the output circuitry have obtained the expected results according to the schematic of the circuit (see Fig 5.6, 5.10, 5.13.

4. Conclusions

The modelling of the accelerometer Inertial MEMS show the great advantages of using VHDL-AMS in mixed signal/multiple domain modelling. It may help the design process permitting to simulate not just electrical features of the microelectronics applications in early phases of the design process. The work presented in this abstract is a description of a work in progress that show all the important features related with the using of new modelling languages.

This work will continue by improving the existing models of the accelerometer, including the physical processes that take place related with piezoelectric features. The models of the output circuitry will also be improved by developing different models of the active elements of the circuit, instead of using the ones provided by ADVanceMS.

Acknowledgments

Presented work has been founded by the Spanish CICYT project n° TIC-2002-01048.

References

A. Collado. *Un nou encapsulat multixip per a accelerometers piezoresistius* PhD. thesis, Universitat Autònoma de Barcelona January 2003.

A. Dewey, H. Dussault and J. Hanna, E. Christen, G. Fedder, B. Romanowicz, and M. Maher. *Energy-Based Characterization of Microelectromechanical Systems(MEMS) and Component Modeling Using VHDL-AMS*. MSM99, Technical Proceedings of the 1999 International Conference on Modelling and Simulation of Microsystems pp. 139-142, Puerto Rico, USA, April 19-21 1999.

C. Ferrer *Design & Development of semiconductors MOEMS and the requirement of International Standardization with a focus on Architecture and Design on Microsystems*. International Seminar MEMSTAND.pp. 118-129. Barcelona, 24-26 February, 2003.

J. Hanna and R. Hillman. *A Common Basis for Mixed-Technology Micro-System Modeling*.Technical Proceedings of the 1999 International Conference on Modeling and Simulation of Microsystems.

M. Schlegel, G. Herrman, D. Müller. *A system level model in VHDL-AMS for a micromechanic vibration sensor array*.First IEEE International Conference on Sensors, Proceedings of IEEE 2002.Vol. 2, pp. 1208-1213.

P. Schwarz, P. Schneider. *Model Library and Tool Support for MEMS Simulation*. International Symposium on Microelectronics and MEMS Technologies, SPIE Proceedings Series. Volume 4407, Edinburgh, Scotland, May-June 2001.

P. Voigt, G. Schrag, G. Wachutka. *Electrofluidic full-system modeling of a flap valve micropump based on Kirchhoffian network theory*. Sensors and Actuators A 66 (1988), pp. 9-14.

II

UML-BASED SYSTEM SPECIFICATION AND DESIGN

Introduction

Piet van der Putten

Department of Electrical Engineering
Technische Universiteit Eindhoven, Eindhoven, The Netherlands
p.h.a.v.d.putten@tue.nl

This part II of the book contains a selection of the most interesting work presented in the FDL'04 workshop on UML-based system specification and design. This workshop explores the use of the Unified Modeling Language (UML) in design methods for next generations of complex embedded systems and Systems-on-Chip.

UML is a very flexible notation for analysis and is necessarily less formal than required for automatic transformations. Adding semantic information is crucial for enabling automatic transformations. Chapter 6, "Metamodels and MDA Transformations for Embedded Systems", written by Lossan Bondé, Cédric Dumoulin and Jean-Luc Dekeyser, presents an MDA transformation engine, where transformations can be defined on the metamodel level, such that reuse is enabled. This chapter describes three successive transformations steps from UML to SystemC.

The Model Driven Approach is also a challenging research area because the path from Platform Independent Model (PIM), to Platform Specific Models (PSM) is far from paved. Chapter 7, "Model Based Testing and Refinement in MDA Based Development", written by Ian Oliver, brings structure in the MDA world by proposing a taxonomy for MDA mappings. The paper describes the relationships between mappings in the context of MDA and model based testing, such as development by refinement (vertical mappings), transformation mappings (horizontal mappings), Code generation.

The question what role can UML play for predictable design of real-time systems, led to research in more fundamental problems. Chapter 8, "Predictability in Real-time System Development" by Jinfeng Huang, Jeroen Voeten, Oana Florescu, Piet van der Putten, and Henk Corporaal, describes such research results. The deficiencies, w.r.t. predictable support in existing design approaches and tools clearly show the need for new approaches. This chapter focuses on compositionality and property preservation in the context of predictability. A case description shows how a new approach for predictable design is used for a real-time control system.

Chapter 9 focuses on both the MDA approach and design of real-time systems. "Timing Performances of Automatically Generated Code Using MDA Approaches" written by Mathieu Maranzana, Jean-François Ponsignon, Jean-Louis Sourrouille, and Franck Bernier, describes the effect on timing properties of the transformations in an MDA approach from UML to automatic code generation for embedded software. This chapter also contains a nice overview of

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 87–88.

some popular commercial tools and their properties w.r.t. the MDA approach. The actual work aimed mainly at two questions. To what extent are models, written during system development, platform independent, and secondly to what extent does automatic code generation reduce timing performance of applications?

Chapter 10 is the last one in this UML part of the book. "UML-Executable Functional Models of electronic systems in the VIPERS Virtual Prototyping Methodology" written by , P. Lister, V. Trignano, M. Bassett, P.L. Vatten, shows an interesting graphical validation environment that transforms UML specifications into virtual prototypes.

Chapter 6

METAMODELS AND MDA TRANSFORMATIONS FOR EMBEDDED SYSTEMS

Lossan Bondé, Cédric Dumoulin and Jean-Luc Dekeyser
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Villeneuve d'Ascq, France

Abstract Embedded system design needs to model together application and hardware architecture. For that a huge number of models are available, each one proposing its own abstraction level associated to its own software platform for simulation or synthesis. To produce a co-design framework, we are obviously obliged to support different models among all possible ones. Between these models we should produce automatic transformations. Each time a new model is included in the framework, we should develop a new transformation.

To improve transformation engine development, Model Driven Architecture (MDA) techniques are useful. This approach permits to define the transformations at the metamodel level. It guaranties to the framework the reuse of models and unifies the definition of the transformation rules.

We present the application of MDA in the context of Intensive Signal Processing (ISP) applications deployed on System on Chip (SoC) platforms. For that purpose, we have developed a new MDA Transformation engine: `ModTransf`. We apply this engine on UML profiles to generate SystemC Transaction Level Model dedicated to ISP. A particular rule will be presented to illustrate the interest of this approach in a multi model embedded system design environment.

Keywords: metamodel, MDA, Model transformations, Embedded Systems

1. Introduction

The MDA is based on models describing the systems to be built. A system description is made of numerous models, each model representing a different level of abstraction. The modeled system can be deployed on one or more

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 89–105.

platforms via model to model transformations. A key point of the MDA is the transformation between models. The transformations allow to go from one model at a given abstraction level to another model at another level, and to keep the different models synchronized. Related models are described by their metamodels, on which we can define some mapping rules describing how concepts from one metamodel are to be mapped on the concepts of the other metamodel. From these mapping rules we deduce the transformations between any models conforming to the metamodels. The MDA model to model transformation is in a standardization process at the OMG [OMG, 2003]

The MDA is based on proven standards: UML for modeling and the MOF for metamodel expression. The new coming UML 2.0 [OMG, 03-07-06] is specifically designed to be used with the MDA. It removes some ambiguities found in its predecessors (UML 1.x), allows more precise descriptions and opens the road to automatic exploitation of models. The MOF (Meta Object Facilities [OMG, 00-04-03] is oriented to the metamodel specifications.

Our proposal is partially based upon the concepts of the `Y-chart` [Gajski and Kuhn, 1983]: application, hardware architecture and then association to map one application on one hardware architecture. The MDA contributes to express the model transformations which correspond to successive refinements between the abstraction levels, from PIM to PSM. In this paper we present the transformation of the association PIM to SystemC PSM for a System on Chip design. For this transformation we use the tool ModTransf developed in our research group in respect of the QVT proposal.

2. The Transformation Engine : ModTransf

Model to model transformations are at the heart of the MDA approach. Anyone whishing to use MDA in its projects is soon or later facing the question: how to perform the model transformations? There are not so much publicly and freely available tools, and the OMG QVT standardization process is not completed today. To fulfil our needs in model transformations, we have developed `ModTransf`, a simple but powerful transformation engine. `ModTransf` was developed based on the recommendations done after the review of the first QVT proposals [Gardner et al., 2003]. Based on these recommendations and on our needs, we have identified the following requirements for the transformation engine:

- Multi models as inputs and outputs
- Different kind of models: MOF and JMI based, XML with schema based, graph of objects
- Simple to use
- Easy modification of rules to follow metamodel changes

- Hybrids: Imperative and declarative rules
- Inheritance for the rules
- Reversible rules when possible
- Customizable, to do experimentations
- Free and Open-Sources.

The proposed solution fulfil all these needs: `ModTransf` is a rule based engine taking one or more models as inputs and producing one or more models as outputs. The rules can be expressed using an XML syntax and can be declarative as well as imperative. A transformation is done by submitting a concept to the engine. The engine then searches the more appropriate transformation rule for this concept and applies it to produce the corresponding result concept. The rule describes how properties of the input concept should be mapped, after a transformation, to the properties of the output concept.

2.1 Basic Principle

Transforming a model can be a very complex task. `ModTransf` helps to reduce this complexity by allowing the specification of a model transformation *rule by rule*. A rule specifies how to transform one or few input concepts to one or few output concepts. This *divide and conquer* approach allows focusing on simple transformations, improves the readability, open the road to rule inheritance and eases the maintenance. In the XML rule language, a rule specifies the source concepts it uses, the concepts it produces in the destination model, and which properties of the source concepts are used to populate the properties of the destination concepts. The rule does not specify how to transform these properties; it only specifies which properties must be transformed and where to store the result. It is the engine responsibility to search and execute the more appropriate rule.

This way of expressing the rules induces recursive calls to the engine, and provides a natural scan of the model to transform. By default a rule is called only once for a given set of inputs. Subsequent calls will return the same results than the first call. This allows breaking the recursivity and avoids multiple transformations of an object: if a source object is referenced by several properties, it will be transformed only once.

The transformation of a model or an object is performed by submitting it to the engine. The engine looks for the most appropriate rule which in turn call the engine to transform the child objects. Thus the entire graph of objects associated to the first object is transformed by using the most appropriate rule for each node of the graph.

The rules can be organized in *rule sets* used as search unit by the engine. It is then possible to specify which rule set should be used for a transformation. If no rule set is specified, the current one is used by default. Rule sets can be used to reduce the scope of a search or to provide several rules transforming the same input concepts, but used in different contexts. It is also possible to specify explicitly which rule should be used by the engine. In this case, the transformation is imperative and the engine uses directly the rule without performing any search.

The rules are searched in the order of their declarations in the rule set. By default, only the first matching rule is executed. This behaviour can be changed by specifying that all matching rules should be executed. Input and output models are submitted to the engine as graphs of objects. The engine and the rules access to the graph of a model through a well known API defining the basic methods they need: attribute access, concept creation, ...

The access API allows making use of different technologies to manipulate the models: JMI and its different implementations (MDR, NsUML, CIM, ModFact, ...); EMF; DOM representation of XML files; models generated from XML schema or DTD with tools like Jaxb, Castor; or any kind of object graph. An implementation of the API is linked to the particular technology used to represent models. Generally it should be developed only once for this technology.

The rules understood by the engine are very simple: they are made of only one guard and one action. The guard is evaluated to select the rule, and the action is executed when the rule is selected. More complicated rules can be built on top of this basic interface. Thus it is possible to write rules directly in Java, as well as in a dedicated high level language with a dedicated interpreter or compiler. One can develop its language and compiler or interpreter. To avoid the burden of such development, we propose a language written using XML, and allowing complex rules.

2.2 The XML Rules

Rules defined in XML use a concept of left and right instead of source and destination. This does not presuppose on the direction of the transformation, allowing writing transformation rules potentially reversible.

The left and right notions will be translated to sources and destinations according to the direction of the transformation. If the transformation flows from left to right, the left will become the source and the right the destination. If the transformation is performed in the other direction, right becomes the source and left the destination.

The reversibility of the rules is possible only in certain cases. The complete transformation is reversible only if all the rules are reversible. Actually, this

feature is only for experimentation. In the remaining of this paper we suppose that the transformation flows from left to right.

An XML rule is made of left conditions, right conditions, and actions. The conditions are used to describe the pattern of source concepts used by the rule, and the concepts that the rule should produce. The description of a condition is the same though it is used as source or as destination: it generally specifies the type of the concept, and optionally the conditions expected on some properties of the concepts or on any sub-properties. Simple conditions like type checking are easy to express: you just specify the expected type. To ease the transformation of models described in UML with an associated profile, some dedicated conditions are provided, like testing or setting a stereotype or a tagged value. More complex conditions like checking the presence or absence of an object with specified values in a collection can be expressed by using an expressions language similar to OCL [OMG, 03-10-14].

When the transformation is performed, the source condition becomes a guard testing the concept while the destination condition becomes an action creating the expected concept. The actions are used to populate the properties of the destinations concepts from the properties of the source concepts, with eventually a transformation. Two main actions are used: The first copy primitive types, with eventually a conversion between the types; the second specifies one or few properties of the source concept that should be transformed to one or few properties of the destination concept by calling the transformation engine to select the more appropriate rule.

Action arguments are specified by using *accessors* allowing to express the source and the destination arguments in exactly the same way. The more common *accessors* uses the expressions language.

A rule being invoked has a context holding various objects like the parameters, the metamodels and user defined variables. All this objects are visible to *accessors* through their declared names. The metamodels are used to query models for concept instances.

The same expression language based on OCL 2 is used in *conditions* and in *accessors*. It allows to describe simple properties access, nested properties access, literals (string, integer, real, Boolean), method calls, metamodel objects access, operation on collection (selection, union, iterate…), operation on expressions (and, or, +, -…)…

Expression	Meaning
src	Access to the attribute named src. A rule has a context holding various attributes like the parameters, the models and user defined variables.
src.name	Access to property name of the object src.
src.attribute	If attribute denote a collection, return this collection.
src.attribute.select (i | i.isDerived = true)	Select the object from attribute having the attribute isDerived set to true. Return a subcollection of matching object. The operation select(iteratorNames | expression) is available on all collection.
src.name +_+ x.name	Concatenation of string
src.method(src.name)	Method call

The XML language provides basic *conditions*, *actions* and *accessors* that should cover current needs in model transformations. Should you encounter a special need that is not cover yet, or should you want to propose a special behavior to simplify your transformation, you can provide your own *condition*, *action*, *accessor* or even complete *rule*. This is done by implementing a Java class providing the desired behavior. Likely all elements of the language accept customized behavior in place of the default behavior.

The `ModTransf` engine is an Open Source project available on the net [Dumoulin, 2004]. We will now see how it is used in our Embedded Application for Soc Design project.

3. PIM and PSM Metamodels for Embedded Systems

We propose a construction of metamodels to support a co-design methodology [Dumoulin et al., 2003]. This proposal is partially based on the concept of Y-chart. We have defined and formalized our concepts in MOF and these MOF specifications have been implemented in UML profiles using Tau G2.

In our approach we design a system starting from two initial models: the application part defines functions and services provided by the system and the hardware architecture part represents an abstraction of the hardware material on which the application will be executed. These two models are then mapped to make the association model. This latter expresses associations between the functional components and the hardware components. Each of these three models are instances of their corresponding metamodels, they are Platform Independent Models. To realize a simulation of this embedded system, we propose the Platform Specific Model for Transaction Level Simulation in

SystemC. `ModTransf` will produce the transformation from PIM association to PSM SystemC.

Our metamodels are too large to be exhaustively presented in this article. We will therefore give an overview of the main concepts, leaving out details.

3.1 The Application Metamodel

The application metamodel is defined in the ISP-UML profile. It is based on the Array-OL language (Array Oriented Language) designed by Thomson Marconi Sonar and dedicated to Intensive Signal Processing. Array-OL introduce the notions of local model and global model. In the ISP-UML metamodel, we propose a set of concepts to specify the application part of a system. An application is described by assembling component wich can be of three types: **CompoundComponent**, **DataParallelComponent**, and **ElementaryComponent**.

- A **CompoundComponent** can contains other sub-components. It expresses the global model of Array-OL and shows component interconnections.

- The **DataParallelComponent** is the heart of intensive signal processing (similar to the local model of Array-OL). It is made of a unique nested component (eventually an **ElementaryComponent**) and of one tiler component for each of its connections. The tilers are used to describe how the data are spraid among the instances of the nested component.

- The **ElementaryComponent** directly refers to an external implementation. It is a computation unit which has no further detailed structure. It gets its input data from input tilers and the result of the computation is carried out through an output tiler.

In the metamodel a **Tiler** instance is represented by an **AolTilerPart** which is introduced to add some tagged values necessary to specify the origin, the fitting and the paving vectors.

Ports represent proxies for data handled by components. They are used as endpoints of connections. A port specifies the type of data it carries, itself defined by an interface in the Object Oriented sense. In ISP UML, all AOL arrays are defined by inheriting from an interface called **AolArray** providing basic attributes: element type, number of dimensions and size of each dimension.

A broad description of the application metamodel in given in figure 6.1.

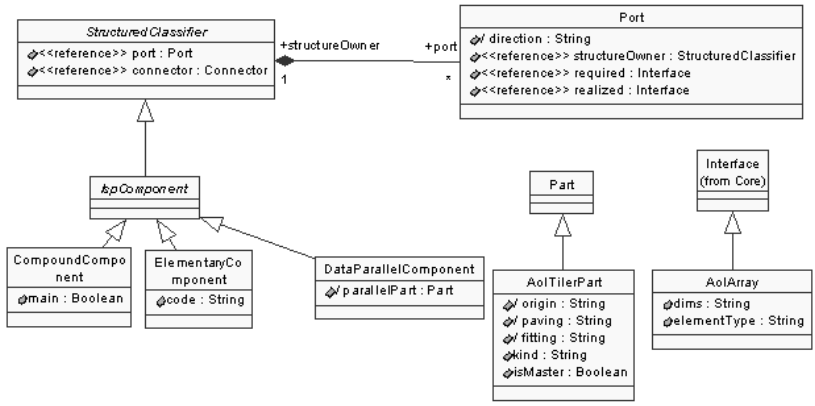


Figure 6.1. Overview of the application metamodel

3.2 The Hardware Architecture Metamodel

The architecture metamodel is similar to the ISP-UML metamodel: it provides components to describe a hardware architecture. The overview of the metamodel is shown on figure 6.2. The hardware component represents abstractions of physical hardware architecture elements. We propose to classify the resources according to two criteria: a functional and a structural (see figure 6.2).

- Structural concepts: **ElementaryHwComponent**, **CompoundHwComponent** and **RepetitionHwComponent** are used to describe the structural aspect of the architecture.

  - The **ElementaryHwComponent** is a component without an internal structural description. For example, it could be used in the case where we have an hardware IP for this component, or in the case where we don't want to model the component more finely.
  - The **CompoundHwComponent** is a component with an internal structure description. The interest in using such a concept is to provide a means for hiding details that are not necessary at a certain level of specification, and also the reuse of existing blocks in modelling other architectures.
  - The **RepetitionHwComponent** is a kind of particular **CompoundHwComponent**, which contains a repetition of the same hardware component. This kind of component is well suited to the modelling

of massively parallel architectures and is motivated by the recent introduction of such architectures in the design of SoC such as the picoChip PC101 and PC102 [picoChip, 2003].

- Functional concepts : **PassiveHwComponent** , **ActiveHwComponent** and **InterconnectHwComponent** are used to specify the functions of the architecture elements.
  - The **PassiveHwComponent** is a storage unit. It stores the data of the application. We typically use it as a representation of elements such as RAMs, ROMs, sensors, or actuators.
  - The **ActiveHwComponent** is a processing unit, it reads or writes into passive resources. This category includes CPUs, DMAs or SMP nodes inside a parallel machine.
  - The **InterconnectHwComponent** is a hardware unit used to specify connections between active and passive components or active and active in the case of a distributed memory architecture. This category includes elements such as bus or an interconnection network.

Each hardware component should be tagged with these two aspects, each one representing a different view on the component. This lead to 9 possible types of hardware components.

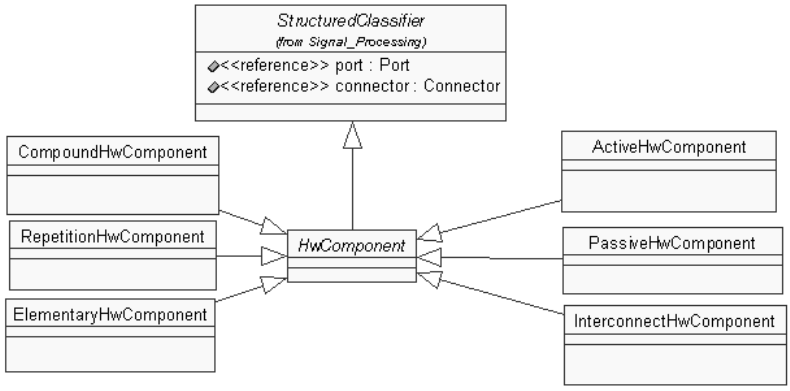


Figure 6.2. Overview of the architecture metamodel

3.3 The Association Metamodel

The aim of the association model is to point out where each software component will be executed, the location of the data used by the software, and the

channels used for the communication between the different hardware components. It is also intended to show how the different executable components are scheduled. This scheduling is static, and decided at the association level. In other words the association metamodel defines a mapping of the application specification and the architecture specification. Therefore the association metamodel imports the application and architecture metamodels and adds to them the following concepts: **TaskAllocation**, **DataAllocation** and **Scheduling**.

The overall view of the metamodel is given in figure 6.3. It is important to note that the **Part** concept stands for an instance of either a application component or an hardware component.

The **TaskAllocation** specifies on which hardware components the different software components are assigned. The *runnables* attribute references software components that are executed on the hardware component referenced by the *activeComponent* attribute which practically is a reference to a processor (or a group of processors).

The **DataAllocation** concept specifies where the application data (arrays) are placed in the architecture. It is mainly the specification of the mappings of the data (arrays) on the memory.

The **Scheduling** concept is used to define the order in which components are processed in the case where several application components are assigned to the same architecture unit. This scheduling is local to each hardware component.

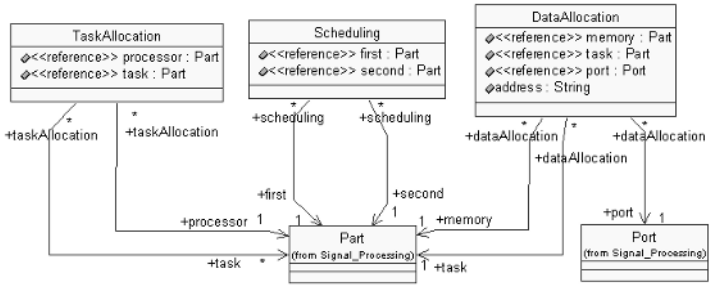


Figure 6.3. Association metamodel.

3.4 The SystemC based simulation Metamodel

SystemC is a C++ class library and a simulation kernel for hardware, software and system modeling. It is particularly suited to:

- propose a methodology for SoC designs consisting of DSPs, ASICs, IP-Cores, Interfaces, ...

- extend C/C++ by providing concurrency, timing, reactivity, communication, signal/data types,...
- simulate up to le level of cycle-accurate.

In our SystemC TLM metamodel, an application contains a *Main* concept which contains a set *processors*, *Memories*, *Interconnection* units and a set of *signals*. This metamodel is oriented towards SystemC code generation for the particular case of Intensive Signal Processing mapped on a SoC. It is designed to fit our particular needs, and should not be considered as a general SystemC metamodel. we intend to later use any standardized SystemC metamodel, for example the UML for SoC metamodel proposed in [Hasegawa, 2004].

The metamodel is provided in figures 6.4 and 6.5. its main concepts are : `Main`, `Processor`, `ComputationModule`, `Memory`, `Interconnect`, `Bridge` and `Signal`.

- The **Main** is the top level component of a SystemC specification. It holds a set of Processors, Interconnects, Memories, Bridges and signals to model communications between these instances.
- The **Processor** refers to a hardware computation unit. To Each processor is assigned a set of compution Modules, and signals for synchronization between the modules.
- The **ComputationModule** is a software unit.
- The **Memory** is a hardware storage unit.
- The **Interconnect** is a hardware unit used to link processors and memories; for example a **Bus**.
- The **Bridge** is a hardware unit used to link two **Interconnects** in complexe hardware architectures.
- The **Signal** is used for synchronisation between computation modules either running on the same processor or on differents processors.

4. PIM Transformation to PSM

The transformation of our PIM *association* metamodel to our PSM *SystemC TLM simulation* metamodel requires the development of a set of dedicated mapping rules. This development requires the identification of the main mapping rules, and then the detail of each of these mappings. In this section we will show the main rules of our transformation, and the details of one of these rules. Then we will explain the implementation of the selected rule with `ModTransf`.

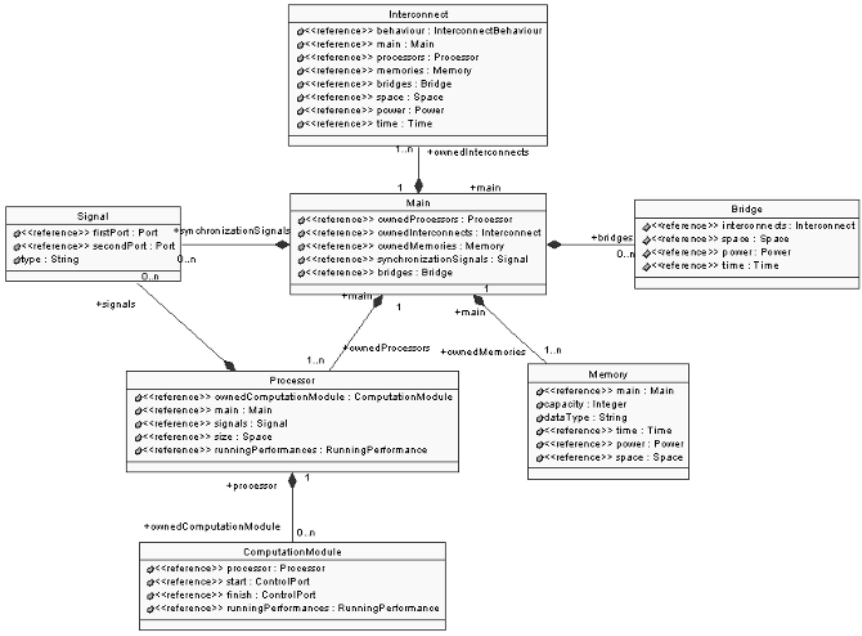


Figure 6.4. SystemC simulation metamodel : Overview

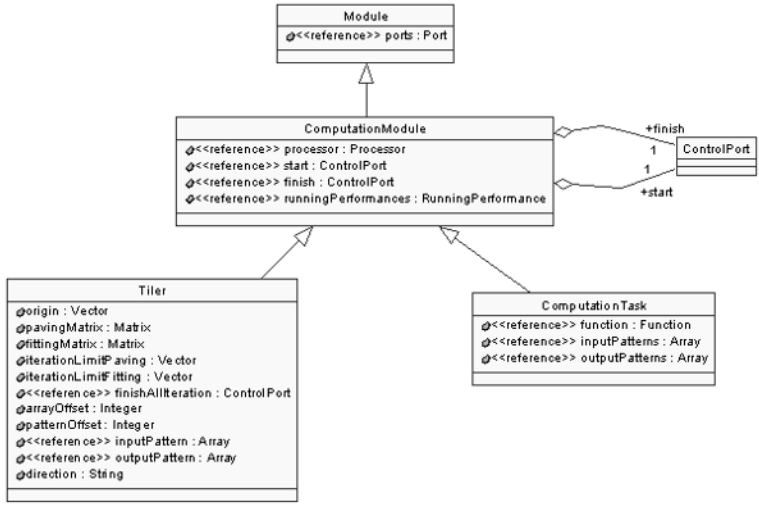


Figure 6.5. SystemC simulation metamodel : ComputationModule

4.1 Main Mapping Rules

The `ModTransf` tool allows a development rule by rule, where, ideally, each rule focus on a simple concept to concept transformation (even if the tool supports a one to one, one to many, and many to one mappings). Thus a complete transformation is made of basic rules. Our transformation main rules (mapping concepts from the association metamodel to concepts of our SystemC TLM metamodel) are given in the following table:

Rules	Input concepts	Output concepts
R1	ElementaryComponent (Ec) TaskAllocation (Ta) ElementaryHwComponent (Ehc)	ComputationTask Processor
R2	AolTilerPart (At) TaskAllocation (Ta) ElementaryHwComponent (Ehc)	Tiler Processor
R3	ElementaryComponent (Ec) DataAllocation (Da) ElementaryHwComponent (Ehc)	Memory DataPath
R4	ElementaryComponent (Ec1) Scheduling (Sc) ElementaryComponent (Ec2)	Signal
R5	ElementaryHwComponent (Ehc)	Interconnect
R6	Port (P)	ControlPort
R7	Port (P)	DataPort
R8	AolArray	Array

Given the input and output concepts mapping, we will now specify the conditions required on the input concepts.

Rules	Conditions
R1	Ehc is an ActiveHWComponent Ta.Task = Ec Ta.Processor = Ehc
R2	Ehc is an ActiveHWComponent Ta.task = At Ta.processor = Ehc
R3	Ehc is a PassiveHWComponent Da.Task = Ec
R4	Sc.first = Ec1 sc.second = Ec2
R5	Ehc is an InterconnectHWComponent
R6	P.ownerElement.Type = ispComponent
R7	P.ownerElement.Type = Memory
R8	none

In the above tables we have given the different elements in the **Y-model** that are used to create the TLM SystemC output metamodel and the conditions

under which the transformation rules are applied. This top level mapping is not sufficient. We need now to detail each rule to specify how properties of a source concept map to properties of the destination concept. Each attribute or feature will be either copied (for simple data types) or transformed from the source concept to the target one. We will now take as example the Rule R2 and give more details about that transformation.

Rule R2:
The `TaskAllocation` concept (class) contains two attributes : *task* which holds a reference to an `AolTilerPart` and the *processor* holding a reference to an `ElementaryHwComponent` which is instance of *ActiveHwComponent* (see figure 6.6).

- The `AolTilerPart` (At) is transformed into a `Tiler` (Tr),
- the `ElementaryHwcomponent` (Ehc) is transformed into a `Processor` (P),
- the `Processor` concept in the systemC TLM model has an attribute named *ownedComputationModule* which contains a collection of all the `ComputationTask` allocated to that `Processor`. In this rule, the `TaskAllocation` is used to add into that attribute of the created `Processor` instance, the `Tiler` (Tr) generated from the `AolTilerPart`.

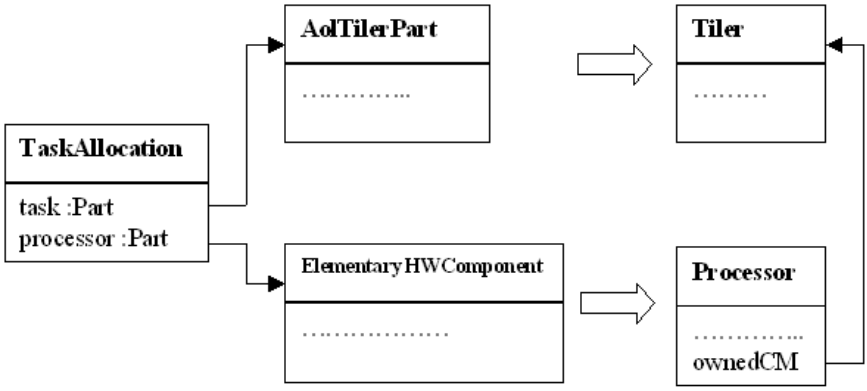


Figure 6.6. Example of Transformation

Once the concept mapping and the details of the mappings are defined, the next step is to implement them with `ModTransf`.

4.2 Transformation Rule in `ModTransf`

Here is the implementation of **Rule 2** using `ModTransf`. The xml Rule is as shown below:

```
1.  <rule ruleName="TaskAllocation" leftParam="src" rightParam="dst">
2.  <description> Transform a TaskAllocation </description>
3.  <leftConditions>
4.    <concept property="src" type="TaskAllocation" model="y-model">
5.      <concept property="task" type="AolTilerPart" model="y-model"/>
6.      <concept property="processor" type="Ehc" model="y-model">
7.        <conditionExpr expr = "function ='ACTIVE' "/>
8.      </concept>
9.    </concept>
10. </leftConditions>
11. <rightConditions>
12.   <concept property="dst" type="systemC.Processor" model="sysc"/>
13. </rightConditions>
14. <actions>
15.   <transform ruleName="EHwC2Processor"
16.     leftProperty ="src.processor"
17.     rightProperty beanName="dst"
18.   </transform>
19.   <transform ruleName="AolTilerPart2Tiler"
20.     leftproperty ="src.task"
21.     rightproperty ="dst.ownedComputationModule"
22.   </transform>
23. </actions>
24.  </rule>
```

In the `actions` part of this rule (lines 14 to 23), the two `transform` actions (line 15 and 19), call the rules named *EHwC2Processor* and *AolTilerPart2Tiler* to transform the `ElementaryHwComponent` into a `Processor` and the `AolTilerPart` into a `Tiler`.

Conclusion

In our co-design environment, the transformation from UML to SystemC is a flow of successive transformations. In order to help in understanding our transformations, we have provided an example of transformation. The complete process from UML to SystemC simulation code is a set of three steps:

- *From the application and the hardware architecture models to the association model.* The mapping of the application onto the architecture is performed automatically by refactoring of a default mapping to satisfy some constraits expressed by the designer. It is an in-built transformation. This transformation aims at generating the association model according to the association metamodel.

- *From the association model to SystemC simulation model.* This transformation takes as input the association model generated at the previous level, the rules for transformations are expressed using `ModTransf`. The transformation engine generates the SystemC simulation model. Each concept in the input model is transformed to its corresponding concept in the SystemC simulation metamodel. This part was studied in this paper.
- *From the SystemC simulation model to SystemC code.* This last transformation is rather a code generation process. The same `ModTransf` tool is used too. The transformation here takes as input the previous model, some rules and some code templates. Theses templates are called by the rules. They contains placeholders replaced by values of the concepts.

Model oriented co-design environment are widely used in the embedded system community. All the transformations between models could benefit of MDA techniques and `ModTransf` like tools. The reuse of models becomes a reality, add new models becomes feasible.

References

Cédric Dumoulin, *ModTransf: A Model to Model Transformation Engine*, 2004, `http://www.lifl.fr/west/modTransf`.

Cédric Dumoulin, Pierre Boulet, Jean-Luc Dekeyser and Philippe Marquet, *UML 2.0 Structure Diagram for Intensive Signal Processing Application Specification*, INRIA, 2003, `http://www.inria.fr/rrrt/rr-4766.html`.

D. D. Gajski and R. Kuhn, *Guest Editor Introduction: New VLSI-Tools*, IEEEC, 1983, vol.16, pages 11-14.

T.Gardner, C.Griffin, A. Koehler and R.Hauser, *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, OMG document 03-08-02, 2003, OMG document. Review of QVT proposals.

Takashi Hasegawa, *An Introduction to the UML for SoC Forum in Japan*, USOC'04@DAC2004, San Diego, California, 2004.

Object Management Group, Inc., *MOF : Meta Object Facility, Specification, Version 1.3*, Jan-2000, `http://www.omg.org/cgi-bin/doc?formal/00-04-03`.

Object Management Group, Inc., *MOF 2.0 Query/View/Transformations RFP*, 2003, `http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View_Transf._RFP.html`.

Object Management Group, Inc., *(UML 2.0): Superstructure Draft Adopted Specification*, Jul-2003, `http://www.omg.org/cgi-bin/doc?ptc/03-07-06/`.

Object Management Group, Inc., *UML 2.0 OCL Final Adopted specification, document ptc/03-10-14*, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.

picoChip, *PC101 and PC102 Datasheets*, 2003, http://www.picochip.com/technology/picoarray.

Chapter 7

MODEL BASED TESTING AND REFINEMENT IN MDA BASED DEVELOPMENT

Ian Oliver
Nokia Research Center
Helsinki
Finland

Abstract The Model Driven Architecture's key principle is that of the mapping. An algorithmic or otherwise mechanical way of generating new more platform specific models from platform independent models with respect to some platform. These mappings are always presented as devices driving the software development. However it is clear that there are a number of uses for mappings and that the idea can be extended to take into consideration not only software development but transformation between differing underlying representations.

Mappings also have a key rôle in the methodology and in the way tests are conducted. Development is coupled with the notion of refinement - that is a mathematically strict way of ensuring certain (critical) properties from the abstract to concrete models. To fully understand and utilise the mappings it is necessary to construct and formalise a framework for these mappings and their meanings (particularly in testing with refinement).

1. Introduction

Testing is probably the most critical issue with regards to software development but is one of the most lacking areas in terms of practise [Binder, 2000]. Technologies such as model based testing [Offutt and Abdurazik, 1999], refinement and so on, are all well known; integrating these together is a critical task for software engineering. There are a number of issues particularly when integrating refinement that need to be discussed.

Model Driven Architecture (MDA) is a proposal by the Object Management Group[1] for a development framework in which the logic of the system is separated from the logic of the underlying platform. The key points about the MDA is that it formalises the relationship between that of a model and that of the mapping between a pair or more of models by encoding algorith-

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 107–122.

mically methodological ideas and concepts. The idea, while arguably not revolutionary is now practical because of the existence of a standard, extensible modelling language (UML), domain specific meta-models and thus languages, a meta-modelling framework (MOF) and standardised model interchange formats (nominally based upon XML). This notwithstanding the development of sophisticated processes, methods and experience of the software engineer.

Model Based Testing (MBT) is a development concept where the validation and verification tests are generated directly from the models of the system under development. Refinement is a well known, formally defined method for ascertaining whether certain properties of a system are preserved across development. However refinement as seen in methods such as the B-Method is very strict and tied to one particular aspect of the model. Model based testing on the other hand deals with many aspects of a model.

In this paper we describe how model based testing, model driven architecture and the notion of refinement combine. We do not attempt to provide a full mathematical treatment of this composition but to outline a number of important issues when working with these technologies.

2. MDA Taxonomy

The MDA is a complex structure which takes into consideration many aspects of modelling such as the language, semantics and model management. In figure 7.1 we show a *simplified* representation of the MDA meta-model written using UML.

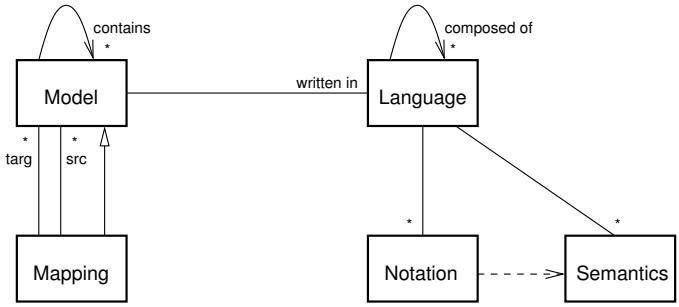


Figure 7.1. MDA Meta-Model

From this model we can clearly see the separation of concerns provided by the MDA and embodied in the technologies on which it is based. For example the UML [OMG, 2002b] makes the distinction between notation and semantics. The UML is supplied with a weak semantics enough to suit nearly

all development tasks and extensible enough for it to be customised to most domains.

The meta-model shown here we have reconstructed from our experiences in using the MDA and MDA-like approaches [Oliver, 2002b]. The primary issues are the creation of a structure of mappings and the explicit representation of the structure of a language.

2.1 Mapping Taxonomy

The mapping is the fundamental construct of MDA. The key point about the mappings in the MDA is that they are of semantic nature and not syntactic nature - that is they map the concepts in one language to the concepts in another preserving the meaning. This in unlike the traditional syntactic mappings found in many tools, for example, those that map UML classes to C++ or Java classes - this is of course fine *if* the semantic gap between the diagram and the code is almost non-existent.

The MDA as it stands does not define any taxonomy of mappings, this we feel leads to some confusions about what a mapping is and what can be performed by a mapping.

We therefore introduce a *simple* taxonomy[2] of MDA mappings based upon the idea that mappings can be broadly classified into three types: development, transformational and code-generation. These can be seen in figure 7.2 - taxonomy in black, MDA meta-model in grey.

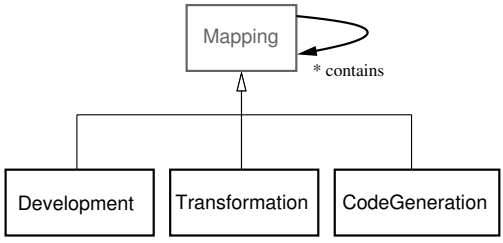


Figure 7.2. MDA Mapping Taxonomy

Mappings may contain other mappings - particularly in the case of development mappings which may utilise a number of transformational mappings to produce their result. This relationship also solves the problem (discussed below) of the transformation vs code-generation mapping - a transformation mapping may contain a (or many) code-generation mapping(s).

Of course there can be much discussion about the structure of this taxonomy and its classifications. One point certainly relates to code generation mappings: are they development, transformational *or* some subtype of a transformational

mapping? We do not discuss these issues here as classifications have too much of a philosophical nature.

Refactoring [Fowler, 1999] we consider the situation where we wish there to be no formal mathematical connection between the source and target models of the mappings. Refinement is the strongest of all the properties and insists that strict relationships between the models exist. Retrenchment is presented as a generalisation of refinement that deals with the situation where requirements may change but we wish to preserve the *ideas* of refinement.

In this paper we concentrate more on the notions of refinement and retrenchment and what it means within the context of an MDA development process that requires those properties to hold.

In [Oliver, 2002a] is a description of the space in which a model exists known as the model matrix shown in figure 7.3. Here we can clearly see that a model exists in a many dimensional space corresponding to various aspects of the state and meaning of that model at any particular time. We concentrate here on just the 'vertical' and 'horizontal' axes to which we give the names

- Development or Vertical Mappings (shown in the model axis)
- Transformation or Horizontal Mappings (shown on the support axis)

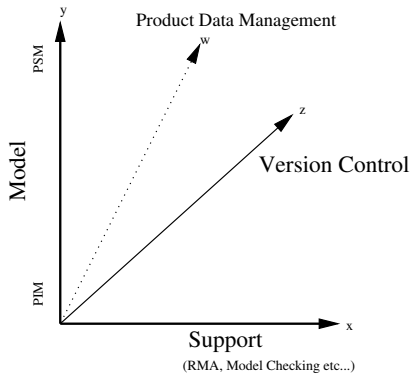


Figure 7.3. Model Matrix

We consider vertical mappings those that are conventionally in MDA parlance thought of as platform independent models (PIM) to platform specific models (PSM) mappings, that is, those that change the abstraction level. Horizontal mappings we consider not to change the abstraction level. Mappings into programming languages are discussed separately.

2.2 Development Mappings

Development mappings are those which change the abstraction level of the model. That is they map platform independent models to platform specific models. It is important to note that the terms 'platform independent' and 'platform specific' are *adjectives* - they describe the relationship between any pair of models in the development process rather than what a particular model is.

An important property of the development mappings is that they do not necessarily imply any change of language in which the models are written. It is conceivable that a model written using, for example UML 1.5 Core, will continue to be written in UML 1.5 Core complete with the same semantics throughout the development process - in this case the development mapping only adds more detailed information into the model.

Normally it is the case that the language - or at least the semantics of the language - changes to reflect the increasing concreteness of the model. For example very platform independent models talk of classes in the broad object oriented sense while platform specific models may introduce more concrete semantics such that the notion of a class becomes closer to that of a database table, VHDL process [Marchetti and Oliver, 2003] or C++ class [Stroustrup, 2000] for example. There is still much open research on development mappings and how they are implemented and what information is supplied. Broadly speaking they can be considered the mapping of the structure, behaviour and other aspects onto the architecture of system (at that level of abstraction) [Boulet et al., 2004, Siikarla et al., 2004].

2.3 Transformation Mappings and Model Based Testing

Transformation mappings are those which do not change the abstraction level of the model but rather extract information from the model. This is a separate issue from a development mapping where the semantics of the language remains relatively comparable (eg: UML to UML-RT [OMG, 2002b, OMG, 2002a]). In transformational cases we have the situation where the language change can be very great. We can show two interesting examples of this and both relate to the issues surrounding 'model based testing' [Offutt and Abdurazik, 1999] and aspect-orientation [Elrad et al., 2002].

In the first example we can map models written using UML-RT into schedulability analysis models [Oliver, 2003] which may be analysed by using a technique such as rate monotonic analysis [Klein et al., 1993]. Here the nature of the mapping is defined such that each unit of execution (method, transition etc) is mapped into an RMA task along with certain dependencies. This model then requires the presence of another source model detailing the deployment architecture of the UML system model. From this deployment model we can ascertain where the scheduling points are in the model.

In the second example we map UML to a different modelling formalism - the formal specification language B [Abrial, 1995]. B however is not object oriented and its major constructor is that of a 'machine'. This machine construct fits neither the concepts of class, object nor component directly. The transformation mapping between UML and B as described in [Snook et al., 2003]. An example of an application of this mapping can be seen below as B code and the class diagram in figure 7.4[3]

```
MACHINE DSP0
/*" U2B3.6.12 generated this component from Package DSP0 "*/
SETS
  DSP={thisDSP}; CELL; CHANNEL; DSP_STATE={boot,init,idle,traffic}
CONSTANTS
  threshold
PROPERTIES
  threshold : DSP --> INT
DEFINITIONS
  disjoint(f)==!(a1,a2).( a1:dom(f) & a2:dom(f) & a1/=a2 => f(a1) union (a2)=0 )
VARIABLES
  dsp_state, current,dspChannels,powerlevel,cellChannels, broadcasting
INVARIANT
  dsp_state : DSP --> DSP_STATE & current : DSP +-> CELL   ...
INITIALISATION
  dsp_state := DSP * {boot} || current :: DSP +-> CELL ||  ...
OPERATIONS
  gotoinit  =
  BEGIN
     SELECT dsp_state(thisDSP)=boot
        THEN   dsp_state(thisDSP):=init ||
          ANY xx WHERE xx:CELL THEN current(thisDSP):=xx END
        END;
   ...
END
```

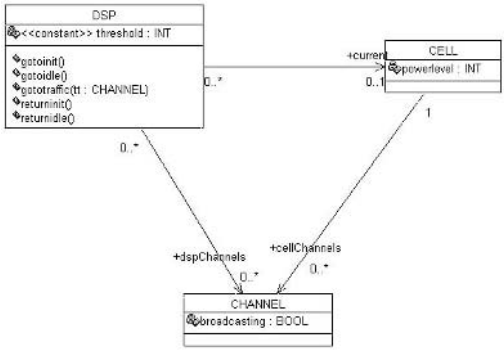


Figure 7.4. Simple Class Diagram

A *desirable* property of transformation mappings is the reversal of the mapping. While moving from one representation to another presents us with the opportunity to explore different aspects of the model, the relaying of the results in these models back to the original is necessary to allow full round-trip modelling between the two representations. In the examples given this means that the underlying semantics between the two representations is isomorphic with respect to the transformed aspects.

The ideas of model based testing can be clearly seen in the above example and how these ideas integrate. Each transformation mapping is a way of generating the test models from one or more aspects of the model. While utilising these models is generally simple for testing purposes, showing how these test models and their results fit together is more complex. This is where the ideas of refinement help in defining how the test models should be utilised with regards to the development of the system under test.

2.4 Programming Language Mappings

Programming language mappings present an interesting difficulty in this taxonomy as it is unclear exactly whether their are developmental or transformational in nature. If we take the case of a traditional UML modelling tool where class diagrams are annotated with methods, types, syntax and code that are of a given programming language, then certainly the mappings are transformational in nature. This is primarily because the mappings do not increase the concreteness of the model but just translate it in to a pure C++ or Java form - the model then is a just a graphical form of the programming language and the mapping purely syntactic in nature. A syntactic mapping places a fixed set of semantics on the nature of the relationship while a semantic mapping requires more information (this may be fixed of course) to resolve into what structure the relationship may be. In the case of Java this could mean Vector, Hashtable etc.

If the mapping requires information (from the architecture or platform definitions) to generate the more platform specific model in order to complete the mapping then the mapping is vertical in nature. Here the mapping is more semantic in nature usually.

3. Example of a Refinement Based Methodology

The PUSSEE project[4] constructed a methodology for the development of hardware systems using UML and formal development processes. The methodology discussed can be placed into an MDA context where the relationships between the models being produced are realised as MDA mappings. Pictorially the development process can be seen in figure 7.5

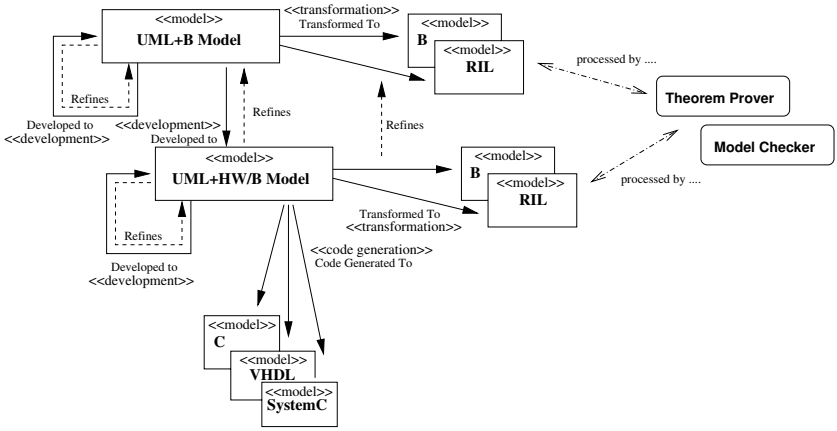


Figure 7.5. The PUSSEE Process

The formal aspects of the methodology are realised by utilising a profile of the UML [OMG, 2002b] known as UML-B [Snook et al., 2003] which integrates the B formal specification language [Abrial, 1995] into the UML as its action and constraint languages. These UML-B models are then verified for internal consistency and refinement properties [Morgan, 1990] by theorem proving mapping from the UML-B to a pure B form (which the user does not necessarily see). Subsequently this is then code generated into a target language such as SystemC, C or VHDL [Hallerstede, 2003]. In addition to this the models may also be mapped to the Raven Input Language (RIL) for model checking with Raven [Ruf, 2001]. Refinement is ensured between the B and RIL models. If both refine then this implies refinement of the UML-B model.

4. Refinement in MDA

Refinement [Morgan, 1990] is a property between models that states that one model is a) linked to a former, less-refined model and that b) the refined model reduces the state-space and non-determinism of the former model. Of course in reality the refinement link between any pair of models is more complex [Back, 1998].

Refinement therefore is a *property* of a relationship between two models (and not a development tactic as is sometimes believed). Referring back to the process described in figure 7.5 refinement is preserved if the following is true:

$$\forall m_1, m_2 : Model \mid m_2 \in DevelopedFrom(m_1) \bullet m_1 \sqsubseteq m_2$$

However this simple equation contains not enough detail - we must consider precisely **which aspects** of the model we wish to consider for refinement. This

is particularly necessary in the case of UML which has no in-built method nor refinement semantics. In the case of embedded systems, only a few critical aspects at certain stages of development are amenable to refinement. For an embedded system this might be just the schedulability characteristics [Klein et al., 1993] or performance constraints. In the methodology described earlier this might then be described:

As the PUSSEE-*method* is based upon the B-Method then the refinement semantics are take directly from the B-Method. Here the refinement operator takes its semantics directly[5] from B-Method (specifically defined between the B models).

Given a model M_1 in UML-B we transform that model into its corresponding B representation, that model we denote B_1. Usage of a suitable B theorem prover proves the consistency of the model (but not necessarily its validity). If we develop M_1 to M_2 with some development mapping d which has the property that its target model(s) must refine the source model we are stating that any B representation of M_2, ie: B_2 must refine B_1 in the sense defined by the B-Method. Therefore we can state that $M_1 \sqsubseteq M_2$ iff $B_1 \sqsubseteq B_2$.

The case is similar when we map to our second target aspects - RIL. Again the method is similar: M_1 is mapped to R_1, M_2 is mapped to R_2, and the development step d preserves refinement iff $R_1 \sqsubseteq R_2$.

However the refinement property of the development step d and thus $M_1 \sqsubseteq M_2$ is **only** true if we are only considering *single* aspects of the model. The notion of refinement therefore has to be extended to take into consideration the multiple-aspects that may be explored during the development of the model.

5. Model Refinement Generalisation

We have so far presented the semantics of refinement of a model based upon the refinement of particular models that are transformed via an MDA mapping from the source model. We can generalise this approach and thus define firstly a number of types of refinement, that is refinement of particular aspects, and then a more global idea of refinement based upon the whole set of aspects being modelled and investigated.

Given a model M at any point in time we have a set of transformations T that extract particular aspects from the model. Not all available transformations need be applied to a model at every level of abstraction; deciding which to apply is governed by the development method employed.

We define a development mapping $d : Model \rightarrow Model$ over which refinement is preserved,ie: $m \sqsubseteq d(m)$. We also define model M_n where n denotes some point in time. For simplicity here we assume that n is an integer greater than zero and that time is a set of discrete values 1,2,3 and so on: $M_{n+1} = d(M_n)$. At any point in time we extract using a transformation

mapping on a model a number of aspects of that model. For example given a model M where the set of transformation mappings T is $\{B, RIL\}$ we obtain A_B and A_{RIL} as models describing those particular aspects.

Our previous definition of refinement:

$$M_n \sqsubseteq M_{n+1} \Rightarrow \forall t : T \bullet t(M_1) \sqsubseteq t(M_2)$$

is too strong and can not take into account that not all transformations are applied. At any point in time we have a model M_n and a set of transformation applications A_n where $A_n \subseteq T$. Then refinement can be defined as:

$$(M_n, A_n) \sqsubseteq (M_{n+1}, A_{n+1}) \Rightarrow \\ A_n \subseteq A_{n+1} \quad \wedge \quad \forall a \in A_n \cdot a(M_n) \sqsubseteq a(M_{n+1})$$

If A_{n+1} contains more transformation mappings than A_n these can not be checked themselves for refinement as those members do not have any meaning in M_n and thus across the pair of models.

This definition also ensures that as more aspects are checked then their properties must refine across all subsequent model pairs ensuring the property that refinement is transitive.

This now gives us the definition of refinement across multiple aspects. Of course what refinement means for a particular aspect of the model still depends upon the methodology being employed for that aspect. In the case of B and B-method this is already defined. In the case for schedulability analysis [Klein et al., 1993] then as timing figures for a model decrease then refinement is preserved. Some aspects do not necessarily have a well defined semantics for refinement.

As we shall now see there are other considerations to make when working with refinement and the interaction between aspects when working with multiple aspects.

6. Other Considerations

Refinement as a property of development is highly desirable, however there are circumstances when the requirements do change and this then causes refinement to break. There are two particular tactics for dealing with this situation.

Also we have to consider the interaction between aspects. Aspects are often considered to be orthogonal in nature, however, there are situations where a change in the model which affects one aspect has repercussions for another aspect - this can be seen for example when working with schedulability and performance characteristics.

6.1 Requirements Volatility

One of the major problems with refinement is that it assumes that the models in question can be built in such a way that the final concrete model is a refinement of the first, most abstract model. While this approach has some very real advantages it is often the case that the initial set of requirements changes so that one model may not refine the previous source abstract model.

One solution to this is that of *retrenchment* [Poppleton and Groves, 2003, Poppleton and Banach, 2002] which introduces a relaxation of the rules of refinement to allow for the situation where the models do not refine. This situation can be easily catered for with MDA by providing information about the retrenchment through a separate model.

Retrenchment requires the use of the notion of concession which defines how the refinement relationship is weakened. In [Popplpeton and Banach, 2004] a semantics for retrenchment is given such that the retrenchment can be expressed in terms of a refinement with respect to some universal model in which the concessions are expressed. We can utilise this by creating an addition model which contains the concessions. This approach though is still experimental and support for dealing with the concession model does not exist at this time.

Another approach is to rebuild the models by layering the features that have been specified in the model [Back, 2002, Back, 2003]. A situation where the refinement property of the development step can not be attained then the model must be refactored [Fowler, 1999]. These techniques represent methods by which one can overcome certain refinement restrictions. It may be possible in the case of retrenchment to factor in the retrenchment ideas into the MDA structure and refinement definition given in this paper. At present we have not explored this in detail but the ideas are presented here to outline the future areas of research.

6.2 Aspect Interaction

Our definition of refinement so far assumes that the aspects of the model are orthogonal in nature. Some pairs (or more) of aspects may interact in such a way that a change in one aspect may cause a failure of the refinement obligation in another, seemingly, unrelated aspect.

This can trivially be seen if we are dealing with extracting memory usage information where an change of functional specification may cause the amount of memory consumed to increase, thus breaking the trivial memory refinement property.

Aspect interaction is one thing that makes model based testing and the approach discussed here more difficult. It is always necessary to examine the interactions. Categorically [Barr and Wells, 1990] this can be achieved by for-

mulating a universal structure that describes the interaction between the two aspects. This is done by constructing a forgetful functor to each of the aspects to build the model of interaction. This is analogous to the retrenchment universal model in [Popplpeton and Banach, 2004].

In most situations we can avoid the necessity of constructing such models by ordering the transformation mappings for testing in such a way that certain aspects are only meaningful for refinement (of the particular system under development) after certain amounts of development have already been completed.

For example, for schedulability analysis or memory usage analysis, these aspects may need to be kept until later stages of development before refinement can be utilised.

7. Methodology

MDA places much more emphasis on method rather than just notation or process. The current MDA releases do not discuss in detail[6] the relationship between method and mappings which in our opinion is critical to the uptake and use of MDA principles.

We are developing a model of this relationship which is used to help in the development of methods which support MDA. The outline of this modelling framework can be seen in figure 7.6.

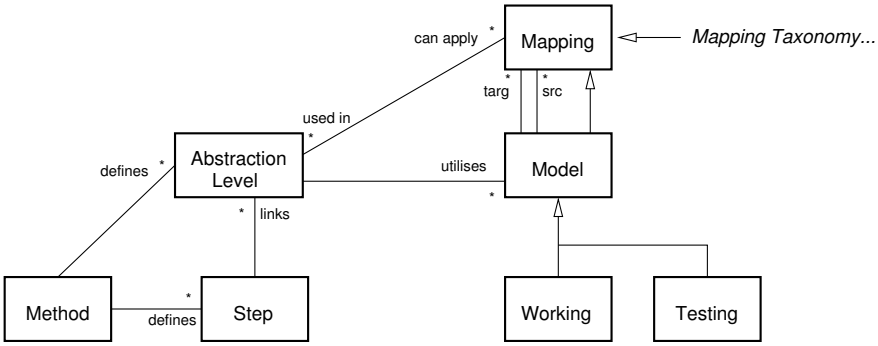


Figure 7.6. Method Meta-Model

We consider a method to consist of a number of method steps which act upon various abstraction levels (at least one or two). At each abstraction level there are a number of mappings that can be *sensibly* applied in that context. This is the first step towards a 'component based methodology' - an idea where meth-

ods are composed of a number of discrete techniques that can be composed together [Ambler, 2003].

We divide the concept of model into two parts, one for models of the system being built and one for models that facilitate some kind of testing. These basically correspond to those models generated by development mappings and those generated by transformation mappings respectively and thus demonstrating the relationship between the two basic types of mapping and the types of model produced.

The development properties related to formality becomes more interesting as we can assert on the sets of mappings for each abstraction level properties about how strict the mappings have to be with regard to more abstract models. It can be seen that weaker mappings (refactorings mainly) are best placed at very high abstraction levels where prototyping and very exploratory work can be carried out while stronger mappings such as those implying refinement are used at more concrete levels.

8. Conclusions

We have presented here the outline of how one may utilise and add the concept of refinement into MDA mappings to construct a formal development method and described a simple taxonomy of MDA mappings, their basic properties and their relationship to methodology. Model Driven Architecture is a very young and immature field although the concepts behind MDA can be traced back to the CASE idea of the 1970/80s. One of the reasons behind this immaturity is a large number of preconceived and unworked ideas regarding what a model is and what a mapping is.

Two areas of particular interest at the moment are extending the taxonomy to take into consideration operations [Alanen and Porres, 2003] such as model union, intersection and so on.

The secondary area of interest is the semantics of (a) method. In figure 7.6 we only define a structure state desirable properties of commutativity across the models produced with respect to the refinement and development relationships [Barr and Wells, 1990]. As the number of methods that do exist are plentiful the material to draw from here is large. One area of contention is that of the idea of a method step and a mapping - it is not clear whether these terms are actually different or whether they are isomorphic in nature.

Refinement offers much advantages when developing systems (usually in software) with regards to the ensured preservation of necessary requirements of the system being modelled. The addition of this concept into the MDA framework provides a placeholder to introduce the ideas of formal development in an MDA context. This we feel makes the idea more acceptable to the engineer who can still work with familiar notations and if truth be said in a de-

sired rigorous way. Techniques such as retrenchment can be applied similarly and offer a weaker approach when required.

The problem of feature interaction can be overcome in two ways, one by letting the methodology take care of things and the other more rigorously through defining what is the interaction between two aspects. Both approaches we believe are complimentary and still require work before advances can be made.

For the developer of a method a placeholder for the semantics of the refinement required in a given context provides the chance to introduce formal development into a given MDA based method.

9. Acknowledgements

This work is partially supported by the EU Project PUSSEE - IST 2000-30103. Many thanks to Jean-Raymond Abrial for some useful comments regarding refinement and methodology.

Notes

1. `http://www.omg.org/mda`
2. This by no means is a complete or "one true" taxonomy - many variations do exist but have yet to be either documented or demonstrated. We defer the argument about what constitute a good or correct taxonomy here to concentrate on the basic framework rather than a philosophical discussion
3. NB: B code listing is shown only partially for space reasons and the ellipses (...) show the missing code):
4. EU Project: IST-2000-30103,Paradigm Unifying System Specification Environments for proven Electronic design, `http://www.keesda.com`
5. actually only in part but we shall discuss this later
6. and nor should they possibly

References

Abrial, J-R (1995). *The B-Book - Assigning programs to Meanings*. Cambridge University Press. 0-521-49619-5.

Alanen, Marcus and Porres, Ivan (2003). Difference and union of models. In *Lecture Notes in Computer Science 2863: <<UML>> 2003 Conference*. Springer. October 20-24.

Ambler, Scott W. (2003). The right tool for the job. *Software Development*, 11(12):50–52.

Back, Ralph (1998). *Refinement Calculus: a Systematic Introduction*. Springer-Verlag. 0387984178.

Back, Ralph-Johan (2002). SFI: A refinement based layered software architecture. In George, C and Miao, H, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25. Lecture Notes in Computer Science 2495*. Springer.

Back, Ralph-Johan (2003). Software construction by stepwise feature introduction. In Bert, D., Bowen, J.P., Henson, M.C., and Robinson, K., editors, *ZB 2002: Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users, Grenoble, France, January 23-25. Lectures Notes in Computer Science 2272*. Springer.

Barr, Michael and Wells, Charles (1990). *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall. 0-13-120486-6.

Binder, Robert V (2000). *Testing Object-Oriented Systems - Models, Patters and Tools*. Addison-Wesley. 0-201-80938-9.

Boulet, P., Cuccuru, A., Dekeyser, J.-L., Dumoulin, C., Marquet, Ph., Samyn, M., Simone, R. De, Siegel, G., and Saunier, Th. (2004). MDA for SoC design: UML to SystemC experiment. In Müller, Wolfgang and Martin, Grant, editors, *Proceedings of UML-SoC 2004, DAC2004, San Diego, California, June 6*.

Elrad, Tzilla, Aldawud, Omar, and Bader, Atef (2002). Aspect-oriented modeling: Bridging the gap between implementation and design. In Batory, D, Consel, C, and Taha, W, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002. Lectures Notes in Computer Science 2487*, pages 189–201. Springer.

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley. 0201485672.

Hallerstede, Stefan (2003). Parallel hardware design in B. In Bert, Didier, Bowen, Jonathan P, King, Steve, and Waldén, Marin, editors, *Proceedings of ZB2003: Formal Specification and Development in Z and B. Lecture Notes in Computer Science 2651. Third International Conference of B and Z Users, Turku, Finland, June 2003*, pages 101–102. Springer.

Klein, Mark H., Ralya, Thomas, Pollak, Bill, Obenza, Ray, and Harbour, Michael González (1993). *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers.

Marchetti, Michele and Oliver, Ian (2003). Towards a conceptual framework for UML to hardware description language mappings. In *Proceedings of FDL03, Frankfurt, Germany, Sept 2002.*

Morgan, Carroll (1990). *Programming from Specifications*. Prentice-Hall.

Offutt, Jeff and Abdurazik, Aynur (1999). Generating tests from UML specifications. In *The Second International Conference on The Unified Modeling Language, Fort Collins, Colorado, USA, October 28-30*. Springer.

Oliver, Ian (2002a). Experiences of model driven architecture in real-time embedded systems. In *Proceedings of FDL02, Marseille, France, Sept 2002.*

Oliver, Ian (2002b). Model driven embedded systems. In Lilius, Johan, Balarin, Felice, and Machado, Ricardo J., editors, *Proceedings of Third International Conference on Application of Concurrency to System Design ACSD2003, Guimarães, Portugal*. IEEE Computer Society.

Oliver, Ian (2003). A UML profile for real-time system modelling with rate monotonic analysis. In Villar, Eugenio and Mermet, Jean, editors, *System Specification and Design Languages*. Kluwer Academic Publishers. 1-4020-7414-X.

OMG (2002a). *Response to the OMG RFP for Schedulability, Performance and Time*. Object Management Group, revised submission edition.

OMG (2002b). *Unified Modelling Language Specification)*. Object Management Group, version 1.5 edition. OMG Document Number ad/02-09-02.

Poppleton, Michael and Banach, Richard (2002). Controlling control systems: an application of evolving retrenchment. In *Lecture Notes in Computer Science 2272*. Springer-Verlag.

Poppleton, Michael and Groves, Lindsay (2003). Software evolution with refinement and retrenchment. In *Refinement of Critical Systems Workshop, RCS03*. Department of Computer Science, Åbo Akademi University, Turku, Finland.

Popplpeton, M R and Banach, R N (2004). Requirements validation by lifting retrenchments in B. In *Proceedings of ICECCS2004: IEEE International Conference on Engineering of Complex Computer Systems, Florence, Italy.*

Ruf, J. (2001). RAVEN: Real-Time Analyzing and Verification Environment. *Journal of Universal Computer Science*, 7(1):89–104.

Siikarla, Mika, Koskimies, Kai, and Systä, Tarja (2004). Open MDA using transformational patterns. In *Model Driven Architecture: Foundations and Applications MDAFA 2004, June 10-11 Linköping, Sweden*.

Snook, Colin, Butler, Michael, and Oliver, Ian (2003). Towards a UML profile for UML-B. Technical Report 8351, University of Southampton.

Stroustrup, Bjarne (2000). *The C++ Programming Language - Special Edition*. Addison-Wesley. 0-201-70073-5.

Chapter 8

PREDICTABILITY IN REAL-TIME SYSTEM DEVELOPMENT

Jinfeng Huang, Jeroen Voeten, Oana Florescu, Piet van der Putten and Henk Corporaal

Faculty of Electrical Engineering Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
J.Huang@tue.nl

Abstract The large gap existing between requirements and realizations has been a pertinacious problem in complex system design. This holds in particular for real-time systems with strict timing constraints and critical-safety requirements. Designers have to rely on a multi-step design process, where design decisions are made at different modelling levels. To ensure the effectiveness of this design process, predictability should be well-supported by design approaches, allowing designers to predict properties of future design outcomes based on existing design results. In this chapter, we first discuss the role of the semantics of design languages and investigated how they can support a predictable design process. Then, the deficiencies, w.r.t. predictability support, of existing design approaches for real-time systems are illustrated by an example. Finally, a predictable design approach for real-time systems is introduced to overcome this problem.

Keywords: Real-time, predictability, semantics, compositionality, composability

Introduction

The aim of real-time system design is to fill the gap between requirements and the realization. However, due to the continuous increase of the functional complexity of real-time systems, and because of stringent timing requirements they have to satisfy, the design gap has increased tremendously. Since traditional code-centric design approaches are obviously not capable of coping with this increasing complexity, designers have to resort to a multi-step design process, where the system is specified and analyzed at different levels of abstractions (see Figure 8.1). This design process usually involves *requirement*

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 123–139.

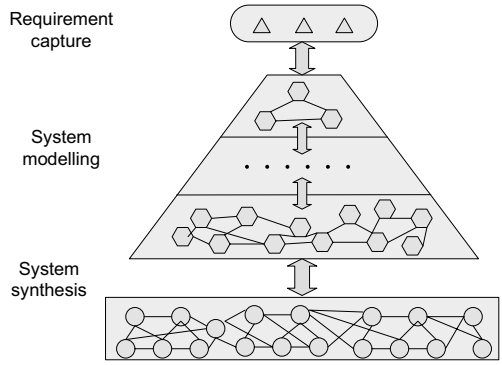


Figure 8.1. The multi-step design

capture, *system modelling*, and *system synthesis*. During *requirement capture*, the system is specified at the most abstract level, which defines the needs and constraints of the system. During *system modelling*, designers explore the design space at different abstraction levels, make design decisions through successive design steps and finally propose a proper design solution, which serves as a blueprint to synthesize a realization. During *system synthesis*, a model is transformed into a realization, which is expected to meet desired properties.

To smoothen the design process and improve productivity, consistency between design outcomes has to be maintained during each design step. In other words, predictability should be well-supported by design approaches, allowing designers to predict properties of future design outcomes based on existing design results.

The remainder of the paper is organized in four sections. In Section 1, We show that semantics of a design language plays an important role for the multi-step design process and has a direct impact on the support for predictability. In Section 2, we will briefly explain the deficiencies of existing approaches in supporting predictability during the design of real-time systems. To solve the problem presented in Section 2, we introduce a predictable design approach for real-time systems in Section 3. Section 4 concludes this chapter.

1. Semantics of design languages

Semantics of design languages has a direct impact on the thinking pattern of developers and the meaning of design outcomes. According to the different abstraction levels of design thoughts, three categories of design languages, requirement, modelling and implementation languages, are involved in the design process.

1.1 Requirement languages

Requirements express the needs and constraints that are put upon a system, each of which is a property that must be present in realizations in order to satisfy the specific needs of some real-world application [Kotonya and Sommerville, 1998]. Usually, requirements are written in natural languages. However, due to the ambiguity of natural languages, complex concepts are usually very difficult to specify precisely. This can result in errors and iterations during the design process. Formal semantics is proposed as a solution to solve the ambiguity problem. It is embedded in requirement languages to promote understandability of requirement specification, to facilitate the automatic checking of requirement consistency and completeness, and to improve the traceability of requirements during the multi-step design process [1].

1.2 Modelling languages

System modelling is the most challenging and creative activity of the design process. During system modelling, designers need first to understand thoroughly the requirements, carefully explore the design space and finally devise a design solution (model). The design model serves as the basis for later system synthesis, the success of which depends to a large extent on the model itself.

Due to the potential complexity of real-time systems, the modelling of such a system is often accomplished by taking a number of steps. Each step only considers a part of the system that is relevant to address some specific design problems. In addition to possessing adequate expressive power to assist designers to specify desired aspects of the system and to analyze the system behavior of interest at each design step, the semantics of a modelling language should also support effective model transformations, which preserves properties of interest during the multi-step design process.

Model transformations: abstraction and refinement. Abstraction and refinement are two elementary transformations performed during the design process (as shown in Figure 8.2). Abstraction is the activity that tries to remove (or hide) irrelevant information, which improves the comprehensibility of existing design models and facilitates the evaluation of different design solutions. The major goal of the abstraction activities is to improve the understandability of the design, enabling design decisions to be made. Refinement is the activity that adds more implementation details to models, thereby reducing the gap between models and realizations. The major goal of refinement activities is the implementability. Intuitively speaking, abstraction activities intend to clarify what the system (component) can do, while refinement activities intend to clarify how the functionality of the system (component) can be achieved.

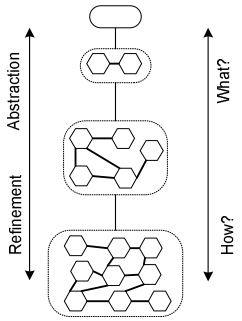


Figure 8.2. Basic design activities

A design process can be considered as a set of abstraction/refinement activities, which intends to fill the gap between the desired properties (what the system should be) and the realization (how the system functions). The effectiveness of model transformations can significantly affect the required design time and cost. This holds in particular for large-scale systems.

In practice, compositionality (or composability) is often regarded as an important characteristic that the semantics of a modelling language should possess, in order to facilitate model transformations for complex systems, where model transformations can be carried out on its subsystems.

Compositionality. The well-known principle of compositionality [Partee et al., 1990] states that *the meaning of a design description is a function of the meanings of its parts and of the syntactic rules by which they are combined.* It is originally proposed to guide the association of the semantics and the syntax of a design language and to assist designers in understanding the meaning of a complex design description in a structured way.

Consider that a system (or subsystem) is represented by a tree structure, where each leaf is a syntactic primitive and other nodes are combination rules. Compositionality ensures that each syntax sub-tree can be understand independently without the consideration of other parts of the tree. Due to the potential complexity of the syntax tree, the semantic interpretation of a complex design description can be far from simple. We can easily foresee that the interpretation of a syntax tree with hundreds of levels, which is not unusual for a complex design description, could easily grow beyond human's understanding. Therefore, compositionality alone does not promise that the meaning of a recursively composed syntax tree can be understood easily.

However, when compositionality is applied to model transformations (abstraction/refinement), it offers many benefits to reduce design complexity and to improve design efficiency. Compositional semantics divides a complex sys-

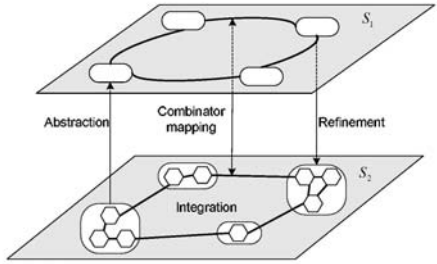


Figure 8.3. Abstraction/refinement based on the compositional semantics

tem into a set of semantic components and ensures the semantical independency of each component in the system. Thus, abstraction/refinement concerning of the whole system can be achieved by local abstraction/refinement for each component and the mapping of combinators to corresponding ones in the other abstraction/refinement level (see Figure 8.3). Furthermore, the correctness of the abstraction/refinement activities can also be verified locally.

One example of design languages equipped with compositionality semantics is CCS (Calculus of Communicating Systems) [Milner, 1989]. Based on the compositional semantics of CCS, observation equivalence is defined, which states that two models are observational equivalent if and only if both models exhibit the same communication behavior to the external observer. The semantic equivalence relation provides the theoretical basis for transformational design approaches, where components of a high-level model are iteratively refined into equivalent components with more details. In this way, observation equivalence can effectively assist the abstraction/refinement of a design description. More detailed discussion about transformational design approaches can be found in [van der Putten and Voeten, 1997], [Koomen, 1991]. Example 8.1 illustrated how abstraction/refinement activities can be carried out in CCS.

EXAMPLE 8.1 *Suppose that system S consists of two components P and Q, which are depicted by:*

$$P \equiv (a.b \parallel \overline{b})\backslash b,\ Q \equiv (c.d \parallel \overline{c})\backslash c \text{ and } S \equiv P \parallel Q. \tag{8.1}$$

The semantics of CCS allows designers to consider the abstraction of P and Q independently from each other. In other words, no matter in what context that P or Q are embedded, P can always be abstracted as $P' \equiv a$ and Q as $Q' \equiv d$. An abstraction of S can be $S' \equiv a \parallel d$. Conversely, P, Q and S are possible refinements of P', Q' and S' respectively.

In summary, suppose $S \equiv P_1 \oplus P_2 ... \oplus P_n$ is a system expressed by a language with a compositional semantics, where P_1, $P_2 ...$ P_n are components of

S and where $\oplus$ is a combinator of components. The compositional semantics guarantees that abstraction or refinement P'_1, P'_2... P'_n of P_1, P_2... P_n can be carried out independently. Therefore an abstraction or refinement of S can be $S' \equiv P'_1 \odot P'_2 ... \odot P'_n$, where $\odot$ is the corresponding mapping of combinator $\oplus$ in S. In Example 8.1, $\oplus$ and $\odot$ are both the parallel compositional combinator $\|$. In practice, S' can be expressed in the same language as that used for S or in a totally different language. For example, properties of a system written in a requirement language can be abstractions of a system written in a modelling language (see the next subsection *composability*).

Composability. The concept of compositionality is intuitively useful in achieving effective abstraction/refinement during the design of complex systems. However, in practice, it is not always as effective as expected. An important reason is that it does not put any restrictions on the assignment of meaning to combinators. As a consequence, semantical independency can always be achieved by assigning trivial semantics to combinators [Zadrozny, 1994]. In practice, the combinator semantics for both abstractions and refinements should be simple enough. For example, the semantics of the combinators $\|$ and $+$ in CCS is defined in a natural way and can be understood easily. The abstraction/refinement of sub-processes in CCS also retains the original combinators.

In the context of concurrent systems, a more restricted "version" of compositionality is sometimes called *composability*. Composability states that properties satisfied by individual components of a system should be satisfied by their parallel compositions [Sifakis, 2001]. For example, assume reactive system S consisting of two parallel components P and Q has a timing response property φ, which states that every environmental stimulus p must be followed by a response q within 3 seconds. If P satisfies φ and the design language supports composability, then $S \equiv P \parallel Q$ should also satisfy φ.

More generally, consider a system $S \equiv P_1 \parallel P_2 ... \parallel P_n$ expressed by a language supporting composability, where P_1, P_2... P_n are components of S and $\|$ is the parallel combinator. Assume each component P_i satisfies property φ_i respectively. Composability of a design language states that S satisfies the simple logical conjunction of these individual properties ($\varphi_1 \wedge \varphi_2 ... \wedge \varphi_n$). We can see that only the parallel operator ($\|$) and the logic conjunction ($\wedge$) are used in composability and their semantics are defined independently from the semantics of composed components.

1.3 Implementation languages

System synthesis is an activity that converts a model into a complete system implementation while preserving the correctness of the model. During this stage, the system is often depicted by an implementation language (such

as Java, C and C++), the semantics of which is usually related with and constrained by the target platform. Due to the different notions and assumptions made at the modelling stage and at the implementation stage, it is not always straightforward to correctly transform a model into a realization. As a consequence, it is difficult to guarantee the validity of the realization w.r.t. the satisfaction of the desired properties, which have been verified in the model.

The difficulty of maintaining correctness between a model and its realization is attributed to several reasons. First, during the modelling stage, certain assumptions are often made about the semantics of modelling languages in order to effectively explore the design space. These assumptions are valid at certain abstraction levels, but they do not always hold for the semantics of implementation languages. For example, to facilitate the analysis of the timing behavior of a model, it is often assumed that actions are instantaneous. However, every action does take a certain amount of execution time in every implementation language. Without carefully considering this difference during system synthesis, the realization may exhibit an entirely different behavior than the model does. Second, some primitives and operations defined in modelling languages do not have direct correspondences in implementation languages. For example, during system synthesis, parallel operations in the model is often implemented by means of a specific thread mechanism offered by the target operating system, which semantics is not always consistent with that of the modelling language.

In most existing design approaches for real-time systems, system synthesis is achieved mainly by a syntactic mapping, instead of by a semantic mapping between the modelling and the implementation language. As a result, the synthesized realization may exhibit a different system behavior than the design model does. A more detailed investigation of real-time system synthesis will be presented in Section 2.

2. Real-time system design approaches

In this section, we are going to evaluate whether existing design approaches have adequate semantic support for real-time systems. We classify existing approaches into two categories, platform-dependent approaches and platform-independent approaches, based on the different timing concepts adopted. This is a justifiable classification because approaches adopting the same concept of timing often provide similar predictability support during the design process. Briefly speaking, platform-independent approaches use a system variable to represent time (denoted as the *virtual time*), while platform-dependent approaches adopt the *machine time* to represent time progress. This implies that the timing behavior of a system depends on the underlying computing platform.

2.1 Platform-dependent design approaches: ineffective model transformations

Platform-dependent approaches take platform computation constraints into considerations at the modelling stage, and use the machine time to specify the timing behavior in their modelling languages. Examples of these languages are Rose-Rt [RoseRT], [Selic et al., 1994] and SDL-96. One major advantage of using the machine time is that no extra (or new) timing concepts are introduced other than those adopted in imperative languages such as C and Java. These approaches are readily accepted by designers, who are familiar with imperative languages. However, this timing semantics is often too ambiguous to support model transformations. This is illustrated by the following example.

EXAMPLE 8.2 ***Two synchronized processes*** P ***and*** Q***:*** *Consider a simple real-time system (shown in Figure 8.4) consisting of two parallel processes P and Q ($P \parallel Q$), each of which comprises an iterative code segment involving timed actions. At the beginning of each iteration, P and Q synchronize with each other. Then process P sets a timer with a 3-second delay and process Q sets a timer with a 2.999-second delay. After the timer of Q expires, Q sends a "rpl-sig" message to P. For process P, there are two possibilities: 1)P receives the timer expiration message and outputs the message "wrong"; 2)P receives the reply message from Q, resets its own timer and outputs the message "correct".*

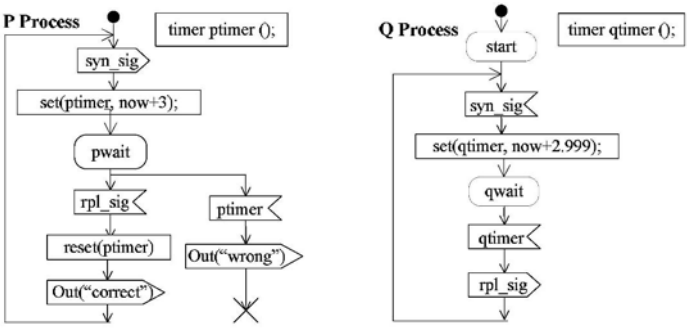


Figure 8.4. A system with two parallel processes P and Q

Here, we use a graphical modelling language based on SDL-96 to describe the system (shown in Figure 8.4). In SDL-96, the timing semantics is given in such a way that each action takes an undefined amount of physical time [2] [Graf, 2002] and the interpretation of timing expressions (such as timers) relies on an asynchronous timer mechanism provided by underlying platforms [Leue, 1996].

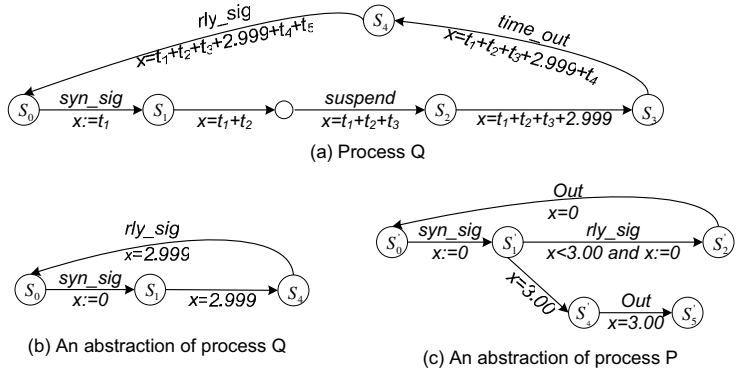


Figure 8.5. The semantics of process P and Q

Suppose two processes P and Q are designed separately, which is often the case in complex system design. Now, let us first look at the timing semantics of process Q depicted in Figure 8.5(a), where x is a clock used to express timing constraints on actions and $x := 0$ represents the setting of clock x to zero[3]. The process first receives a "syn-sig" message, which takes time duration t_1. Before the next statement ($set(qtimer, now+2.999)$) is executed, the operating system might switch to other processes taking a total amount of time t_2, before it switches back to process Q. Then the timer is set and the process is suspended (taking time t_3) to wait for the timer expiration message. Between the time that the timer expires and the time that process Q responds to the $time\_out$ message, again the operating system might take a total amount of time t_4 for the execution of other processes.

Execution times t_1, t_3 and t_5 are neglectable w.r.t. most of real-time properties of interest in a modern computing platform. In the case that process Q is the only active process running on the platform, t_2 and t_4 are zero. As a consequence, Figure 8.5(b) can be considered to be a proper abstraction of process Q. In a similar way, an abstraction for process P can be obtained, which is depicted in Figure 8.5(c). In design practice, it is often assumed that compositionality (or composability) is well supported. That is, the integration of parallel processes can preserve the properties of the integration of their abstractions. Therefore, the integrated system ($P \parallel Q$) is often reasoned about through the abstractions. This would indicate that P should never output the "wrong" message when it is integrated with process Q.

However, in certain circumstances, the platform-dependent semantics of both processes does allow process P to output the "wrong" message in the integrated system. For example, in Figure 8.5(a), when process Q is in state S_1, the underlying operating system can first make P the active process, then P can set the timer and suspends itself, after which the operating system switches

back to Q, which sets a timer with an expiration of 2.999 seconds. If one context switch, one timer setting, suspending one process together with the other necessary scheduling execution take more than 0.001 seconds in total [4], the timer of process P might expire before that of process Q. As a result, P outputs the “wrong" message.

From the above example, we can see that the abstraction of the integration of a set of components cannot always be correctly reasoned about from the abstraction of its components. To eliminate the unexpected behavior, designers have to rely on ad-hoc way to tune the behavior of each component, involving a tremendous number of design details of other components to be considered. As a result, the design process is often time consuming and prone to errors.

Real-time scheduling is often adopted in practice to alleviate the problems mentioned above for platform-dependent design approaches.

Real-time scheduling:. In the research domain of real-time scheduling, a system is viewed as a set of concurrent tasks. A scheduler is used to manage the activation and execution of tasks concurrently running in the system. It assigns the computation time by giving different priorities to tasks. In general, the task with a higher priority is scheduled before those with lower priorities. The goal of real-time scheduling is to devise a priority assignment scheme to ensure that every task can be accomplished in time. In principle, a feasible schedule can eliminate unwanted interferences from other tasks, reducing the ambiguity of the timing semantics of each task. However, real-time scheduling lacks a consistent framework to integrate functionality and timing [Liu and Joseph, 2001], and it is only suitable for a particular set of real-time systems, such as periodic systems. Hence they are not a general solution to the design of complex real-time systems. Especially interaction-intensive real-time systems are difficult to design with scheduling theory.

2.2 Platform-independent design: ineffective system synthesis

Contrary to platform-dependent design approaches, platform-independent design approaches often adopt a virtual timing concept, which is independent of any underlying execution platforms. Furthermore, the semantics of their modelling languages often treats the time progress and the action execution in an orthogonal way [Nicollin and Sifakis, 1991], which can reduce the ambiguity of the timing semantics and improve the understandability of design descriptions. In this semantic framework, system actions (such as communications and data computations) are timeless (taking zero time) and time passes without any action being performed. On one hand, such semantics can provide sufficient expressive power to describe the timing behavior of a system.

On the other hand, compositionality (or composability) is supported by the semantics of their modelling languages and effective abstraction/refinement can be supported during the design process. Furthermore, since actions are instantaneous, additional analysis code does not take up time, keeping the original timing behavior of the system unchanged. A typical modelling language based on this semantic framework is SDL-2000 [Z.100, 2000], which is supported by the TAU Generation 2 tool (TAU G2 in short) released by Telelogic [TAU G2]. Other examples often used in academic contexts are timed automata or process algebra, such as timed CCS.

The timing semantics of the design descriptions in Example 8.2 can also be given in platform-independent semantic frameworks. In these frameworks, t_1 till t_5 are all zero and we can always consider the semantics depicted in Figure 8.5(b) for process Q to be a proper abstraction of that in Figure 8.5(a), and the same holds for the abstraction of process P depicted in Figure 8.5(c). Consequently, the abstraction of the combined system $P \parallel Q$ can be captured by combination of Figure 8.5(b) and Figure 8.5(c), in which process P should never output the "wrong" message. We made the same model in TAU G2, and the behavior of the system ($P \parallel Q$) was indeed as expected.

Although most platform-independent approaches provide sufficient support in their modelling languages for predictable design, bridging the large semantic gap between these modelling languages and implementation languages is still not solved adequately (see Section 1, *Implementation languages*).

Automatic transformation of design models to realizations is a superceding technique to manual transformation, the latter of which is inefficient and prone to errors. In current practice, the automatic transformation is achieved mainly by the syntactic mapping of syntax primitives and constructs between two design languages, instead of by a semantic mapping. As a result, inconsistencies can be observed between the design model and the realization. For example, actions are usually assumed to be instantaneous in the model, while they do take a certain amount of physical time in the realization. Without careful considerations of this semantic difference, the realization can exhibit faulty behavior. Although the model in Example 8.2 made in TAU G2 is proven to be correct, errors are observed in the automatically synthesized realization (see Figure 8.6).

3. A predictable design approach

In the previous section, we have investigated the deficiency of the existing design approaches in supporting predictability for real-time systems. In this section we introduce a design approach which can overcome this problem. This approach has two distinct characteristics. First, the POOSL language is adopted at the modelling stage, which is a platform-independent modelling

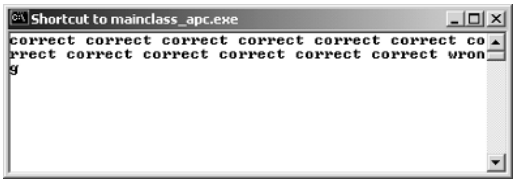


Figure 8.6. The output of a realization of two parallel processes P and Q

language. Second, the Rotalumis tool is used to automatically synthesis realizations (C++) from POOSL models. Most importantly, the synthesis procedure is based on a formal linkage between between the semantics of the design language (POOSL) and that of the implementation language (C++), which guarantees property-preservation between models and realizations.

3.1 The design language POOSL

In this section, we give a brief overview of the POOSL language (Parallel Object-Oriented Specification Language), which is employed in the SHESim tool and developed at the Eindhoven University of Technology. POOSL language integrates a process part based on a timing and probability extension of CCS and a data part based on a traditional object-oriented language [Voeten et al., 1998]. For example, the system in Example 1 can be modelled by the POOSL code shown in Figure 8.7(a). The expressive power of POOSL enables designers to describe concurrency, distribution, communication, real-time and complex functionality of a system using a single executable model. We have successfully applied it to the modelling and analysis of many industrial systems such as a network processor [Theelen et al., 2003], a microchip manufacture device [Huang et al., 2002] and a multimedia application [van Wijk et al., 2002].

Similar to some other recent design languages equipped with adequate predictability support, the semantics of the POOSL language [van der Putten and Voeten, 1997], [van Bokhoven, 2002] is also based on a two-phase execution model, which guarantees the predictability support during system modelling.

The implementation of the two-phase execution model in simulation tool SHESim is achieved by adopting so-called process execution trees (PETs). The state of each process is represented by a tree structure where each leaf is a statement or a recursively defined process method (an example is the PET of $P \parallel Q$ shown in Figure 8.7(b)). During the evolution of the system, each PET provides its candidate actions to the PET scheduler and dynamically adjusts its state according to the choice made by the PET scheduler. More details about PET can be found in [van Bokhoven, 2002]. The correctness of PETs with

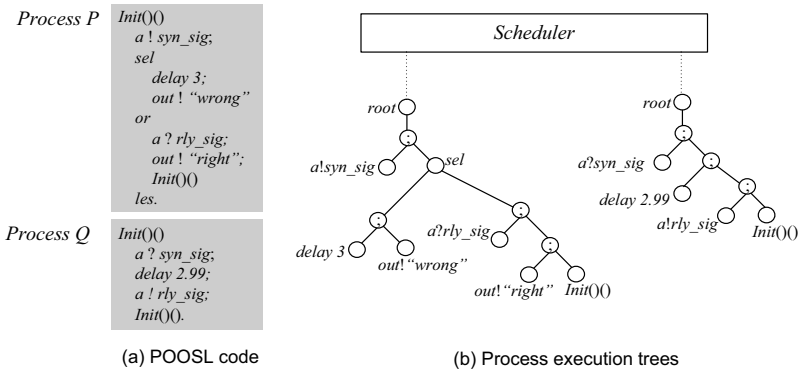


Figure 8.7. The $P \parallel Q$ system in POOSL

respect to the semantics of the POOSL language is formally proven in [Geilen, 2002].

3.2 Rotalumis

The generation tool Rotalumis takes the POOSL model acquired during the modelling stage as its input and automatically generates the executable code for the target platform. To ensure property-preservation during the transformation, a formal linkage between two semantic domains of the modelling and implementation languages is built based on the ϵ-hypothesis [Huang et al., 2003]. The ϵ-hypothesis requires that:

1 A model and its realization should have the same observable execution sequence.

2 Time deviations between activations of corresponding actions in the model and the realization should be less than ϵ seconds.

In the case that the ϵ-hypothesis is complied with during the transformation, we could predict properties of the realization from those of the model. More specifically, if the model satisfies a property P formally specified by MITL (Metric Interval Temporal Logic)[Alur et al., 1991], we know that the realization satisfies a 2ϵ relaxed property $R^{2\epsilon}(P)$ of P [Huang et al., 2003]. For example, a typical response property that "every input p must be followed by a response q between 3 and 5 time units" is defined by formula $\Box(p \rightarrow \Diamond_{[3,5]}q)$. Its 2ϵ relaxed property is $\Box(p \rightarrow \Diamond_{[3-2\epsilon,5+2\epsilon]}q)$. In case an upper bound of the time deviation between the realization and the model is 0.01 seconds and $\Box(p \rightarrow \Diamond_{[3,5]}q)$ is satisfied in the model, we can conclude that property $\Box(p \rightarrow \Diamond_{[2.98,5.02]}q)$ holds in the realization.

The Rotalumis tool tries to satisfy the hypothesis by applying the following techniques:

Process execution trees. POOSL language provides ample facilities to describe system characteristics such as parallelism, nondeterministic choice, delay and communication that are not directly supported by C++ or other implementation languages. In order to provide a correct and smooth mapping from a POOSL model to a C++ realization, PETs are used to bridge the semantic gap between two languages. The data part of a POOSL model is directly translated into corresponding C++ expressions since no large gap exists between their semantics. The process part of a POOSL model is interpreted as a C++ tree structure whose behavior is the same as the PET implemented in SHESim. As a result, the synthesized realization exhibits exactly the same behavior as that in the model, if we interpret it in the virtual time domain.

On the other hand, the realization of a system needs to interact with the outside world and its behavior has to be interpreted in the physical time domain. Since the progress of the virtual time is monotonically increasing, which is consistent with the progress of the physical time, the event order observed in the virtual time domain should be consistent with that in the physical time domain. That is, the PET scheduler ensures that the realization always has the same event order as observed in the POOSL model. Therefore, any qualitative timing property (such as safety and liveness) satisfied in the model also holds in the realization.

Synchronization between virtual time and physical time. To obtain the same (or similar) quantitative timing behavior in the physical time domain as in the virtual time domain, the PET scheduler tries to synchronize the virtual time and the physical time during execution. This ensures that the execution of the realization is always as close as possible to a trace in the model with regard to the distance between timed state sequences[5].

Due to the physical limitations of the platform, the scheduler may fail to guarantee that the realization is ϵ-close to the model (for some fixed ϵ value). In this case, designers can get the information about the missed actions from the scheduler. Correspondingly, they can either change the model and reduce the computation load at a certain virtual time moment, or replace the target platform with a platform of better performance.

With the aid of the Rotalumis tool, a property-preserving realization of Example 1.2 can be automatically synthesized from a POOSL model.

4. Conclusions

To smoothen the system design process and improve design productivity, the semantics of design languages should provide sufficient support for predictable

design. More precisely, two aspects should be supported by the semantics of design languages. 1) The semantics of design languages should support compositionality (and composability), thereby facilitating the design of complex systems. 2) A formal linkage between the semantics of modelling and implementation languages is necessary, which can serve as a basis for automatic system synthesis.

In this paper, we investigate the support of the existing design approaches for the above two aspects. The investigation is carried out in two categories of real-time design approaches: platform-dependent approaches and platform-independent approaches. Platform-dependent approaches adopt the physical time as their basic timing concept, often lack sufficient support to model and analyze complex real-time systems, and predictability is not well supported during system modelling. On the other hand, platform-independent approaches adopt the virtual time as their basic timing concept, which improves predictability during system modelling. But they are often ineffective in system synthesis, due to the large semantic gap between modelling and implementation languages.

To cope with the problems of existing design approaches, a predictable approach is proposed, which has two distinct characteristics. First, the POOSL language is adopted during the modelling stage, the semantics of which provides adequate predictability support for real-time system modelling. Second, the Rotalumis tool is used to automatically synthesis realizations (C++) from POOSL models. Most importantly, the synthesis procedure complies with the ϵ-hypothesis, which ensures that realizations keep the same qualitative and quantitative (up to 2ϵ) timing properties as models. In paper [Huang et al., 2004], a rail-road crossing system is presented, which is designed by applying this approach. The analysis of property-preservation between the model of the rail-road system and its realization is presented in [Florescu et al., 2004].

Notes

1. Although it is often unrealistic to formalize all the requirements of the desired system in practice, we believe that critical timing and safety requirements should be precisely specified.

2. Physical time can be considered as machine time here.

3. Since the timing semantics of process Q is influenced by the underlying platform and other processes running on the same platform, in general it is too ambiguous and (almost) impossible to be accurately illustrated by state diagrams. Figure 8.5(a) only shows a part of the semantics of Q, which is already sufficient to show the deficiencies of platform-dependent semantics.

4. In a complex concurrent real-time (software) system, the cost can far exceed 0.001 seconds due to frequent context switches between many processes.

5. A timed state sequence is an execution of a system, in which a time interval is attached to every state. If two timed state sequences are ϵ-neighbouring, they have the same state sequence and the least upper bound of the absolute difference between the left-end points of corresponding intervals is less than or equal to ϵ. For more information, see [Huang et al., 2003].

References

Alur, R. Feder, T. and Henzinger, T.A. (1991). The benefits of relaxing punctuality. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 139–152. ACM Press.

Florescu, O. Voeten, J.M.P. Huang, J. and Corporaal, H. (2004). Error estimation in model-driven development for real-time software. In *In Proceedings of Forum on specification and Design Language, FDL'04*, Lille, France.

Geilen, M.C.W (2002). *Formal Techniques for Verification of Complex Real-time Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands.

Graf, S. (2002). Expression of time and duration constraints in sdl. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, LNCS.

Huang, J. Voeten, J.M.P. and Geilen, M.C.W. (2003). Real-time Property Preservation in Approximations of Timed Systems. In *Proceedings of First ACM & IEEE International Conference on Formal Methods and Models for Codesign*, Mont Saint-Michel, France. IEEE Computer Society Press.

Huang, J. Voeten, J. van der Putten, P.H.A. and Ventevogel, A. (2004). Predictability in real-time system development (2) a case study. In *In Proceedings of Forum on specification and Design Language, FDL'04*, Lille, France.

Huang, J. Voeten, J.P.M. van der Putten, P.H.A. Ventevogel, A. Niesten, R. and van de Maaden, W. (2002). Performance evaluation of complex real-time systems, a case study. In *Proceedings of 3rd workshop on embedded systems*, pages 77–82, Utrecht, the Netherlands.

Koomen, C. J. (1991). *The design of communication systems*, volume 147 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston; London; Dordrecht.

Kotonya, G. and Sommerville, I. (1998). *Requirement Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York.

Leue, S. (1996). Specifying real-time requirements for sdl specifications - a temporal logic-based approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 19–34. Chapman & Hall.

Liu, Z. and Joseph, M. (2001). Verification, refinement and scheduling of real-time programs. *Theoretical Computer Science*, 253(1):119–152.

Milner, Robin (1989). *Communication and Concurrency*. Prentice Hall. ISBN 0-13-114984-9 (Hard) 0-13-115007-3 (Pbk).

Nicollin, X. and Sifakis, J. (1991). An overview and synthesis on timed process algebras. In K. G. Larsen, A.Skou, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification, LNCS 575*, pages 376–398, Alborg, Denmark. Springer-Verlag.

Partee, B., ter Meulen, A., and Wall, R. (1990). *Mathematical Methods in Linguistic*. Kluwer Academic Publishers.

Rational Rose RealTime. `http://www.rational.com/tryit/rosert/index.jsp`.

Selic, B., Gullekson, G., and Ward, P.T. (1994). *Real-time object-oriented modeling*. John Wiley & Sons, Inc.

Sifakis, J. (2001). Modeling real-time systems-challenges and work directions. In *Proceedings of the First International Workshop on Embedded Software*, pages 373–389. Springer-Verlag.

TAU Generation 2. `http://www.taug2.com/`.

Theelen, B.D., Voeten, J.P.M., and Kramer, R.D.J. (2003). Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks, Special Issue on Network Processors*, 41(5):667–684.

van Bokhoven, L.J. (2002). *Constructive Tool Design for Formal Languages from semantics to executing models*. PhD thesis, Eindhoven University of Technology, The Netherlands.

van der Putten, P.H.A. and Voeten, J.P.M. (1997). *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands.

van Wijk, F.N., Voeten, J.P.M., and ten Berg, A.J.W.M. (2002). An abstract modeling approach towards system-level design-space exploration. In *Proceedings of the Forum on specification and Design Language*, Marseille, France.

Voeten, J.P.M., van der Putten, P.H.A., Geilen, M.C.W., and Stevens, M.P.J. (1998). System Level Modelling for Hardware/Software Systems. In *Proceedings of EUROMICRO'98*, pages 154–161, Los Alamitos, California. IEEE Computer Society Press.

Zadrozny, W. (1994). From compositional to systematic semantics. *Linguistics and Philosophy*, 17:329–342.

Z.100 Annex F1: Formal Description Techniques (FDT)–Specification and Description Language (SDL) (2000). Telecommunication standardization sector of ITU.

Chapter 9

TIMING PERFORMANCES OF AUTOMATICALLY GENERATED CODE USING MDA APPROACHES*

Mathieu Maranzana,[1] Jean-Francois Ponsignon,[1] Jean-Louis Sourrouille,[1] and Franck Bernier[2]

[1]*INSA Lyon*
Bât. B. Pascal, Lab. PRISMa
F-69621 Villeurbanne Cedex, France
{Mathieu.Marazana, Jean-Francois.Ponsignon, Jean-Louis.Sourrouille}@insa-lyon.fr

[2]*Schneider Electric, A2 plant*
Electronic and Software Research Division
F-38050 Grenoble Cedex 9, France
Franck.Bernier@mail.schneider.fr

Abstract

To improve usual software development in the C programming language, a model-driven approach seems very promising. Writing generic models, independent of any platform, is attractive and will reduce the work to implement software on multiple platforms. Another very attractive feature of the MDA approach is complete and automatic code generation. Within the very concrete context of an industrial example, this article aims to answer two questions: "*To what extent models written during development are platform independent?*" and "*To what extent does automatic code generation reduce timing performances of applications?*". To answer these questions, the study focuses on the sequence of models needed to transform a system specification into an implementation, and then on the timing performances of the code generated by CASE tools supplying different level of services.

Keywords: MDA, Timing Performance, Automatic Code Generation, Embedded systems.

*This work was partly supported by *Schneider Electric* under grant 008686.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 141–159.

1. Introduction

Most embedded systems software are still developed in the C language while application complexity increases continually (100 000 lines and several processors for a middle range circuit breaker at Schneider Electric). To reduce development effort, a promising way is to adopt a MDA (Model Driven Architecture) approach with two great advantages: writing generic models, independent of any platform, will reduce the work to implement software on multiple platforms, and automatic code generation using CASE tools (Computer-Aided Software Engineering) will reduce the implementation effort. The additional time required by modeling will be greatly counterbalanced by the reduction of the implementation effort.

The advantages of such an approach are well known, but a main obstacle in the area of embedded systems is the assumed decrease of performances (time, space, etc.). Thus a first issue is: "*To what extent does automatic code generation deteriorate timing performances?*". To answer this question, a comparison of the performances of the code generated by several tools has been achieved, in partnership with *Schneider Electric*, using an industrial example. The reference is the corresponding program directly written in C. The tested CASE tools all are based on the UML (Unified Modeling Language [UML 1.5, 2003]), and supply at least behavior modeling and automatic generation of the associated code. However they are very different regarding their approach, synchronous or asynchronous, and the offered services, for instance the management of timing constraints.

A promise of MDA is to reuse models in different platform implementation, but the meaning of *platform independent* must be made clear: for instance, may a platform independent model depend on the target programming language? Thus the second issue of this work is: "*To what extent models written during development are platform independent?*". Of course, software development based on modeling has numerous other major interesting features, but within the concrete context of our industrial experiment, the scope of the work is limited to these issues.

The rest of the article is organized as follows. Section 2 first examines the refinement of models from specification to an automatic code generation using a MDA approach; the section 3 presents the tools used to model the industrial experiment, and gives a classification based on service levels; in section 4, the industrial example is described and timing performances are discussed.

2. MDA approach

The MDA approach [MDA, 2001] for software development is *model-centered*, focusing on models instead of programs, while the code is generated from the model. We assume that code generation is automatic and complete

therefore the code should not be modified directly and manually. That way the contribution of the MDA approach is the most meaningful.

A model is a representation of a system in a given formalism. In the sequel we call model the set of points of view on the system, e.g., requirement view, dynamic view. Each view may be based on several UML diagrams such as use case diagram, activity diagram and sequence diagram for the requirement view.

In broad outline, the core of the MDA approach is a specification using a PIM (Platform Independent Model), then the transformation of this PIM into a PSM (Platform dependant model), and finally the translation of the PSM into code. This is a well-known and long established way to deal with software development. According to tools, to application domains, and to platforms, the benefit induced by MDA varies. This section studies the use of MDA in the context of our experimentation.

2.1 PIM and PSM

The really promise of the MDA approach is kept only when the code is generated automatically and completely for different execution contexts, OS and/or platforms. The ideal would be to hide context and platform specific details using a virtual machine, but except in some particular cases this situation is not very realistic.

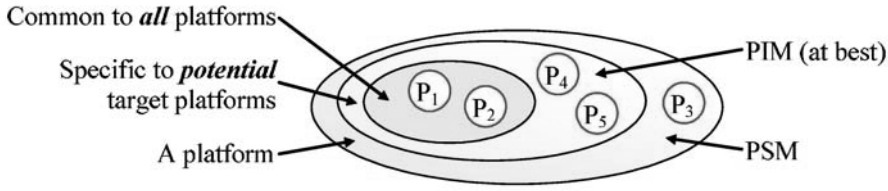


Figure 9.1. Properties used by models.

The truly nature of a PIM is not to be completely platform independent but to be as independent as possible. A PIM becomes a PSM when it relies on properties subjectively viewed as platform specific. This position is justified from the fact that a PIM may require properties (e.g., priority levels, timer) or even a programming language that some platforms do not provide (P_1 and P_2 excluded on Figure 9.1). When these properties are necessary to specify the system, no model can be strictly platform independent. From a practical point of view, a model that includes all the common aspects of the potential target execution contexts (P_3, P_4, P_5 on Figure 9.1) could be seen as a PIM, while the specialization for a platform is a PSM (P_3 on Figure 9.1).

2.2 From PIM to PSM

For automatic code generation to be possible, the model should be precise enough and include all information related to the platform described in the Platform Model (PM). The ideal solution would be to merge the PIM and the PM to build the PSM, thus keeping a great independency. However, the simple following example shows that additional steps are required.

Let assume a model in which the execution time of the operations should be provided. The association of the operation with its duration directly within the PIM would transform immediately the PIM into a PSM. If the duration of the operation op_i is 10ms on a platform and 15ms on another one, a most generic solution is achieved marking in the PIM the operation op_i with the name $durationOp_i$, which should be interpreted as " op_i has a duration of $durationOp_i$ " (*Marking* in [MDA, 2003]). During the transformation of the PIM into a PSM, $durationOp_i$ will be replaced by the value given in the PM of the target context. As a result, the link between op_i declared in the PIM (whose name may be changed) and its duration defined in the PM (e.g., $durationOp_i$ = 10) should be kept continuously and automatically: to make again this link at each transformation of the PIM into a PSM is not acceptable..

A model becomes a PSM when adding details such as an OS call to change thread priority. The sequence of models on Figure 9.2 is in line with usual software development using refinement:

- The PIM specifies the application, including system constraints (e.g. deadlines) without hypotheses about the platform, and takes logical decisions only (for instance, the choice between thin and heavy client).
- Marking means to add adornments that define precisely the nature of notions, e.g. a *Port*, the name of a variable for a duration, and also a tag to force the generation of a constructor.
- When platform specific information is added to the PIM it becomes a PSM: at least the target platform characteristics, but other examples are calls to primitives of the target OS, the use of processor or clock properties. When a tool provides a powerful and complete virtual machine, the PSM is more platform independent, but in return the timing performances of the generated code may be reduced.
- When the marked PSM is translated into code, marks are interpreted according to the target PM.

Remarks:

- The order given Figure 9.2 is the most normal course of events, but marking is not mandatory to go from PIM to PSM.

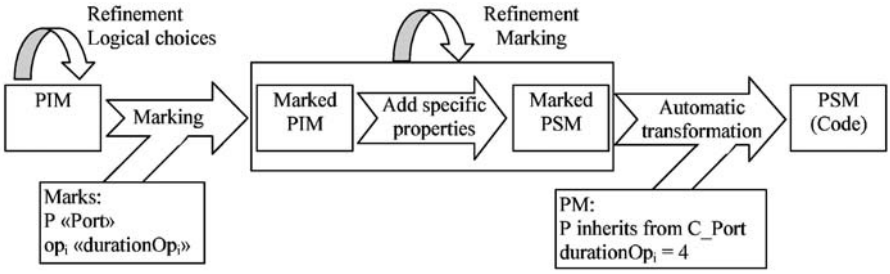


Figure 9.2. Development cycle and models sequence.

- To avoid platform dependency, the developer may build its own piece of virtual machine. Thus, the priority change of a thread remains generic, while a library is implemented on each target.

2.3 Interpretation of UML expressions

Automatic translation into code implies that the PSM provides all the needed information. Although the PM gives additional details (e.g. OS services calls), there is a gap between the level of abstraction of models and the level of details required to generate code (Figure 9.3). This gap can be explained in terms of interpretation.

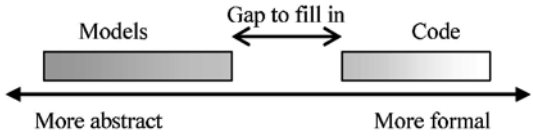


Figure 9.3. Gap between models and code.

Usually, language constructions are associated with notions in a semantic domain. In this semantic domain, an expression (i.e., any sentence built using the words of the language) can be false even when its syntax is valid (e.g., $t[i]$ has no meaning when i is out of the range of t). The semantic domain from which the UML takes its meaning is the modeling domain (not to confuse with the modeled system domain in which a model takes its meaning). Valid expressions should have a meaning in the modeling domain. The result of the mechanism by which sense is given to a model is commonly called an *interpretation*. In a formal language, an expression is associated with a known set (very often one) of licit interpretations in the semantic domain (*a model is inconsistent when it has no interpretation*). In the UML, the semantics of the notions is often not precise enough and incomplete. Hence, several in-

terpretations can be associated with expressions. Moreover, the UML claims that it is a universal language implementation-independent, and interpretations are intentionally kept open [UML 1.5, 2003]: "*Although the intent is to define the semantics of state machines very precisely, there are a number of semantic variation points to allow for different semantic interpretations that might be required in different domains of application*" (open interpretations are called *Semantic Variation Points* in [UML 2.0, 2004]).

The many interpretations are an advantage for a modeling approach. First, the developer delimits a set of a priori acceptable interpretations, and then, as far as his/her knowledge of the system increases, details are added and interpretations are removed. The abstraction level should remain high not to be overburdened with details and to reduce the cost of changes at the development beginning.

2.4 Code generation

A model implemented in a programming language (PL) has only one interpretation, thus the many interpretations are a drawback for automatic code generation. For instance, here is the way events are processed in a UML 1.5 state machine: "*Events are dispatched and processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority based schemes*". Each tool has to define its interpretation among all the licit ones.

In some case, an interpretation is frozen, for instance "links implementing 1 to 1 associations are created automatically". But, very often, users need to choose the interpretation, and the UML extension capabilities such as *Stereotypes* and *taggedValues* become indispensable. For instance a class marked with a stereotype *Capsule* will be interpreted as an active object (object + thread of control) with additional properties such as a message queue, an automatic transition at creation time, etc. These extensions allow adding details, but also increase the expressive power adding notions that the UML does not supply such as *Port* or mechanisms such as broadcast.

The Figure 9.4 summarizes the different ways to translate UML into a programming language. All UML expressions are not translated into code, for instance the tested tools ignore the sequence diagrams (area a on Figure 9.4). In Figure 9.4, the area b varies according to the target programming language but above all according to the UML extensions provided. To avoid each tool to choose its own extensions, the UML profiles play an essential part since they standardized these extensions (e.g., *Schedulability, Performance, and Time* [SPT, 2002]).

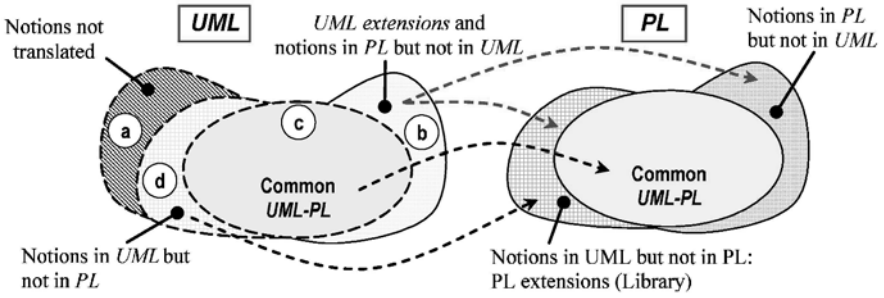


Figure 9.4. Translation of UML into PL (Programming Language).

3. CASE tools classification based on service levels

In order to develop an application, at least an IDE (Integrated Development Environment) supplying tools to assist programming is needed. From this basic level, it is not possible to detail the numerous additional services that a COTS (Commercial Off-The-Shelf) CASE tool and a model-centered approach may provide. However, we have considered useful to classify the different tools used for our study, according to the supplied services. The list (Table 9.1) gives a sample of COTS tools, among the best known and covering a wide solution domain, and which has been dictated by the industrial partner (except ARTO). Obviously, dozens of other tools (Artisan, Objecteering, Tau, etc.) could have been added to extend our comparative study. The discussed services are related to development in the area of embedded systems, subject to performance constraints, which explain the choice of the C language as a basis for comparing timing performances of each solution.

3.1 UML extensions

Compared with a development in *C* (Table 9.1a), object-oriented approaches enforce an organization of data and operations that masks implementation and favors the discovering of independent components.

At the second level (Table 9.1b), classes and their relationships are described in the UML, and the corresponding code, limited to the structure, is generated automatically. This step represents an improvement, but there are still drawbacks because it is difficult to maintain the consistency between the model and the code (two different points of view of the same system).

At the next level (Table 9.1c), the implementation of the state diagram is done automatically. In comparison to the previous level the progress is important since active objects are introduced and asynchronous events are processed using queues. As a natural extension, behavior simulation of state machines

Table 9.1. Classes of COTS CASE tools.[a]

Level	*[Class] COTS Tool*	*Description*
a	[IDE] Microsoft Visual C++	A well-known IDE for a "usual" development with editors, debugger, browser, etc.
b	[Standard UML]	Any visual modeling tool based on the UML. The developer builds the class diagram (structure description) which is used to undertake a partial code generation (headers).
c	[Extended UML] Rose-RT [IBM-Rational, 2002] Rhapsody [I-Logix, 2002]	From the only UML state diagrams, the tools provide a complete and automatic code generation, often in C or C++, as well as the simulation of the associated state machine. This code generation can be done for several platforms, according to parameters (virtual OS). A comparison between actual traces and specified sequence diagrams can take place.
d	[Synchronous] Esterel [Esterel-Technologies, 2002]	Esterel is a synchronous language very convenient for the specification and the programming of reactive systems. Its strong points are the static checking and the proof of properties.
e	[QoS] ARTO [Contreras et al., 2001]	In addition to the services of the *Extended UML* class, it supplies QoS management and behavior adaptation when the system is overloaded. The code is automatically generated from a description in the UML (Rational Rose© with Add-in) adorned with QoS data and degradation policies. Messages between objects convey the QoS requirements, and are scheduled dynamically aiming to avoid temporal faults, if needed degrading the system behavior.

[a] All these tools are available on the market except *ARTO* (framework for Adaptable Real-Time Objects), which is a research prototype.

brings a higher level of abstraction than usual debuggers, and allows users to focus on logical aspects rather than implementation ones.

The UML is used as a modeling language throughout the software development process. This process should be, at least, described at level 3 (*defined*) or more in the CMM (Capability Maturity Model [Paulk et al., 1993]), for instance using USDP (Unified Software Development Process [Jacobson et al., 1999]). Such a process allows checking of model inconsistencies, hence increases the quality, although current UML COTS tools do not make all the needed checks [Kuzniarz et al., 2002]. Increasing automation requires to go beyond the UML limits and to extend it. As the UML provides all the ele-

ments needed to describe a language, the extensions are described into UML *profiles*.

3.2 Synchronous approach

It is difficult to ensure behavior determinism for asynchronous systems, i.e., to prove that the same input event sequence always leads to the same result. In reactive systems, i.e., that react to stimuli coming from their environment, an alternative is to use synchronous languages such as *Esterel* [Berry, 2000] or *Lustre* [Halbwachs et al., 1991] (Table 9.1d). From the *Esterel* point of view, the time is a logical notion composed of a sequence of instants. At each instant the program reads the inputs and builds the outputs. *Esterel* allows static checking of behavior consistency and behavioral properties, for instance to prove that the lift cannot move while its door is open. *Esterel* is not actually a UML extension since transition firing is based on *SyncCharts* [André et al., 2002]: there is no notion of *run to completion* and within the same instant all the enabled transitions fire, while in the UML this instant corresponds to a sequence of interactions. For example, let $s_1 \xrightarrow{e/o'.e'} s_2$ be a transition in o with e' that triggers $s'_1 \xrightarrow{e'} s'_2$ in o': the two transitions will fire at the same instant in *Esterel* while in the UML the first one will be completed before starting the second one. To analyse *Esterel* temporal behavior is reasonably easy insofar as it only requires measuring the maximum duration of an instant.

3.3 QoS management

The temporal behavior of asynchronous programs is hard to control. A solution is to manage dynamically timing constraints, and more generally the QoS (Quality of Service) directly within the application (Table 9.1e). The needed QoS (e.g., deadline or result accuracy) and the QoS characteristics (e.g., operation duration) are first described, for instance using the *Schedulability, Performance and Time* profile [SPT, 2002]. Then, at run time, a middleware schedules the tasks according to the required QoS in order to maximize the quality of the supplied services. A multithreaded layer increases reactivity and avoids low importance activities in progress to block critical activities. A static temporal analysis should prove that timing constraints are met (out of the scope of this study).

Within a system, not all timing constraints are necessarily *hard*. When all timing constraints cannot be met, a solution is to degrade the behavior in a controlled manner to find, for all the running applications, a satisfactory working point. In addition to QoS needs, the accepted degradation policies and the application degree of freedom must be specified, for instance alternative operations or operations that may be cancelled when the system becomes over-

loaded. According to this knowledge, the system automatically adapts its behavior to its execution context. Temporal analysis is required only to verify the remaining hard timing constraints. As the QoS optimization problem is NP-Hard [Lee et al., 1998], most of the current proposals aim to find a good solution based on feedback adaptation or heuristics (numerous works, e.g., [Cardei et al., 2000, Abdelzaher et al., 2000]).

3.4 MDA and tools

To ease model construction, tools increase the expressive power of UML adding notions (e.g., for Rose-RT, *Port*, *Protocol*) and mechanisms (e.g., for Rose-RT, message broadcast and priorities, periodic timer, etc.). New properties are also added to models to achieve a complete code generation (e.g., a logical thread is associated to a physical thread). Moreover, many tools relying on UML choose directly the target platform language, for instance C++ or Java, as action language. Does this choice convert irremediably the PIM into a PSM? In fact, these languages are often target independent, but as all the targets do not provide all those languages, the potential targets are immediately limited (Figure 9.1). Furthermore, there is a good reason to choose very early the target language: to avoid using constructions with no equivalence in the target language.

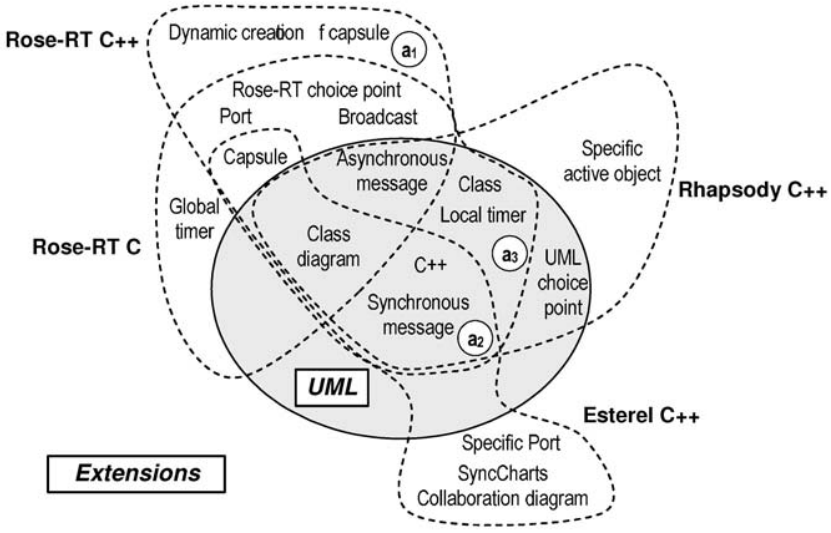


Figure 9.5. Model translation between tools: examples from our experiments.

Starting from the same initial formalism (UML), Figure 9.5 clearly shows that tools, using extensions and choosing the target platform language as ac-

tion language, give rise to significant differences. These differences make the change difficult from one tool to another one or from one target language to another one in the same tool. For instance, to change from Rose-RT C++ to Rose-RT C, all the areas marked a_i on Figure 9.5 require a transformation, e.g., all synchronous messages (a_2) must be changed into asynchronous ones. In addition, the C++ code for all the actions must be translated into C code (no class notion any more). Changing from Rose-RT C++ to Rhapsody C++ may be equally awkward due to Rose-RT specific extensions, such as *Port* and message broadcast. Another problem comes from the interpretation of the UML extensions, for instance, Rose-RT and *Esterel* share the same *Port* notion but with a different semantic. Of course, *Esterel*, based on the synchronous approach, is far away from the other tools used for our case study: no timer, no thread, no asynchronous message, etc. Therefore, changing a non *Esterel* based model to a model relying on *Esterel* requires a great effort from the developer.

3.5 Discussion

Does the MDA approach really allow writing platform independent models? We have assessed this approach in the context of an industrial example, and written models with several COTS CASE tools. These experiments allow drawing some conclusions.

Most of CASE tools provide a virtual machine quite independent from the target OS and hardware. Thus, very little system calls are necessary to complete the models, and a quick and easy configuration in the deployment diagram allows changing the implementation. From this point of view, platform independence seems satisfactory.

On the other hand, choice of the used CASE tool is very important: each tool brings its set of extensions and constraints, and early forces an action language. Since models rely on a tool they are de facto less generic. Finally, models appear more dependent on CASE tools than on target platforms.

4. Experiments

The MDA approach aims to automatically generate code from the model. However, and this is especially true in the embedded field, the developers often express several reasons to refuse this new approach:

- They do not trust the code generator and the quality of its code;
- They think that the generated code is inefficient (time and space);
- They do not master exactly the code architecture to undertake a debugging session at the code level etc.

Using concrete measures on an industrial example (the software development of protection functions for the mains supply at *Schneider Electric*), we tried to answer the following question: "*Does a full automatic code generation deteriorate the timing performances*?". Obviously, other criteria could have been taken into account such as the readability of the generated code, its size, its maintainability, its safety (e.g., ability to statically ensure the respect of timing constraints, essential in an embedded context), but they are clearly out of the scope of our article. In order to refine our timing performance measures, a set of additional experiments, in particular based on a multithreaded execution, has been developed. Each experiment deals with a specific point of view (such as asynchronous or synchronous messages in a mono or multithread context, etc.), and a representative one is examined in detail in section 4.2.

4.1 Industrial example

Short description of the circuit breaker. The modeled system includes four main actors: (i) the mains electricity; (ii) the safety device made up of N separate protection functions; (iii) the circuit breaker; (iv) and the end-user.

The mains supply provides the safety device with its two typical outputs, the voltage and the strength of the current. Periodically, the safety device reads these two values and compares them to predefined thresholds. In case of overrun, the safety device starts to be faulty. If this fault persists until the validation timeout expires, it is a genuine fault, and the circuit breaker must be triggered. Afterwards, a user intervention may be required to reset the safety device. If it was a transient fault (no more thresholds overrun for the voltage and the strength as the timeout expires), the safety device goes back to usual waking state.

Measure explanation. The initial model was made up of 17 classes, firing fifty or so transitions. To carry out the measures, classes have been added to instrument the model (timing aspects), and to simulate and manage failures all along the experiment: no user's intervention is required during the complete sequence of measures.

The sequence diagram (Figure 9.6) describes the scenario from the *timeout_t1* expiration, which confirms the fault (asynchronous message, stick arrowhead on Figure 9.6), to the circuit breaker triggering (asynchronous message *trigger*). After the validation timeout expiration, the safety device requests an update of the current configuration (*update* then *scanState*, which is spread to the N protection functions). When a protection function is enabled, it initiates the synchronous reading of the new state (*readState*, filled solid arrowhead on Figure 9.6), and propagates the fault (*propagate*). The defect treatment goes on with a release request (*triggerRequest*) and the physical release com-

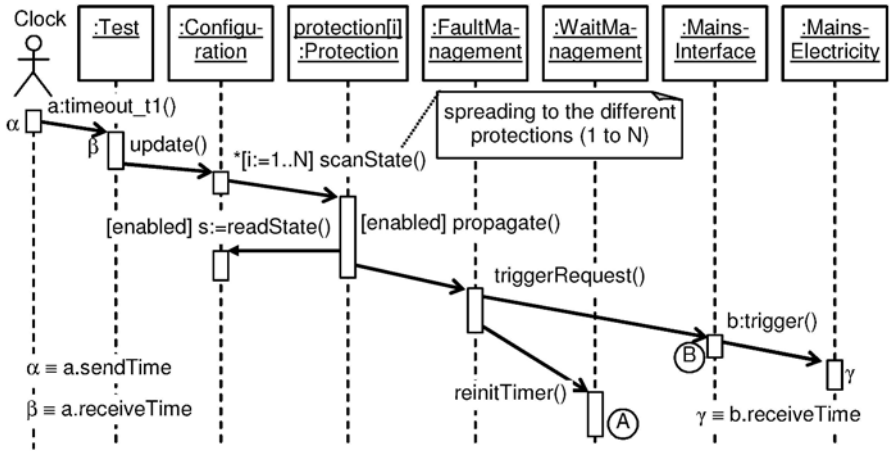


Figure 9.6. UML sequence diagram for the measure.

mand of the circuit breaker (*trigger*). Finally, a timer is reset (*reinitTimer*). The execution order of the sequences A and B (Figure 9.6) is respectively:

- Defined by the priority of both messages, *triggerRequest* and *reinitTimer*, for *Rose-RT* and *Rhapsody*;
- Unspecified for *Esterel*;
- Defined by the sequence of calls for *Visual C*;
- Defined by the deadline of messages for *ARTO*.

The required time to release the circuit breaker from the expiration of the timer *timeout_t1* (fault confirmation) is measured (Table 9.2). For the *Extended UML* class, the time $\gamma - \beta$ is considered, since the instant α is inaccessible. For all the other classes, the time $\gamma - \alpha$ is measured.

Inherently, the circuit breaker example does not require concurrency. We implement it in a single thread whenever it was possible: *C*, *Esterel C++/C* and *Rose-RT C*. The timer needed in the model is simulated with an infinite loop in *C* and *Esterel*, and managed in the same thread in *Rose-RT C*. *Rose-RT C++* always uses a separate thread to implement the timer, therefore the generated code is multithreaded even if it was designed to run in a single thread. All the *ARTO* implementations are multithreaded since each active object owns at least two threads, one for the controller and one to execute methods. This essential difference explains the gap between the results Table 9.2. Additional measures are given in section 4.2 to deal with the multithreaded context.

Table 9.2. Time for the triggering.

Class	*Measure*	*Tool/Language*	*Average in μs*	*Standard Deviation*
IDE	γ - α	Visual/C	2.4	0.2
Synchronous	γ - α	Esterel/C	12.7	0.7
Synchronous	γ - α	Esterel/C++	14.6	0.2
Ext. UML	γ - β	Rose RT/C	45.6	1.2
Ext. UML	γ - β	Rose RT/C++	103.6	4.2
QoS	γ - α	ARTO/C++	299.0	5.0

Comments. The measures, shown in Table 9.2, were carried on a Pentium II 333MHz, under *Windows NT SP5*, running with the highest priority class. All the programs were compiled using *Visual C++ 6.0* and the compiler option *maximize speed*. As the UML class tools do not provide a complete code generation, measures concerning that class were not carried out. No special fine tuning were done for the measures. The models were never adapted to the best possible advantage of each specific tool. Therefore, the figures do not represent a best case, and are average results achieved without any optimization effort.

For the *Extended UML* class, we only mentioned one specific tool, namely *Rose-RT*, but with a code generation in both C and $C++$. It is worthwhile to notice that the figures for *Rhapsody* are of the same order. As expected, the measures clearly indicate that the number of provided services impacts on the overall temporal performance: increasing services consumes time. However, it is worthy of note that *Esterel*'s performances are outstanding, and yet *Esterel* provides a high level of services.

4.2 Sender / receiver example

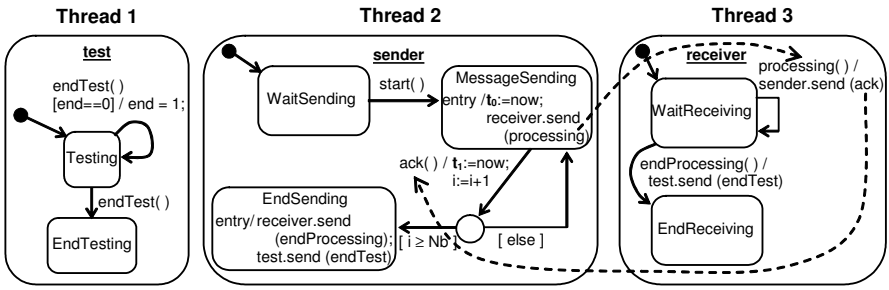


Figure 9.7. UML state diagrams of the sender / receiver example.

To analyze and explain the previous results, additional single measures have been achieved on ad hoc examples, each example underlining a special point of view (message broadcast, hierarchical states, timer, etc.). Of course, all these different examples have been executed in the same environment (processor, operating system, and compiler) as the industrial example. As the different tools carry out various optimizations when applications are monothreaded (e.g., removal of critical sections), it is interesting to undertake measures in a multithreaded execution context, in order to reduce the influence of these uncontrolled optimizations. Among all these examples, we have chosen the sender / receiver example (Figure 9.7), which brings out the accurate sequence of messages (Figure 9.8) and the overhead due to the messaging system in a multithreaded execution context.

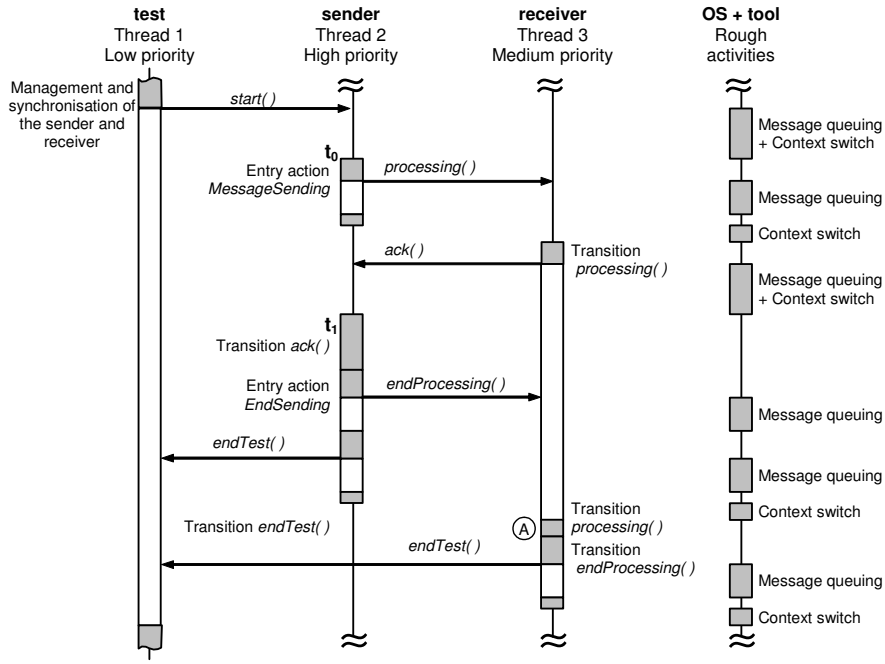


Figure 9.8. Sequence of activities when receiver's priority less than sender's priority (RPS).

Measurement of a full cycle. This example measures the time required for a full cycle ($t_1 - t_0$ on Figure 9.7): sending the message *processing()* to the *receiver*, handling this message in the *receiver* and sending back to the *sender* a reply message *ack()*. In order to comply with a multithreaded context, the *sender* and the *receiver* are each executed in their own thread, and the two following possibilities are considered: (i) the receiver's priority is smaller

than the sender's one (*RPS*), and (ii) the receiver's priority is greater than the sender's one (*RPG*). A third thread, called *test*, is introduced to manage and synchronize both the *sender* and the *receiver* threads. As this last thread has the lowest priority among all the threads, it does not upset the measured sequence. In the first case (i), the *sender* goes on with its execution once the message *processing()* was sent to the *receiver* (Figure 9.8). On the contrary, as soon as the *ack()* message is sent by the *receiver*, a context switch is performed, and the *sender* begins immediately its execution. The conclusion of the *processing()* transition is executed later (part labeled A on Figure 9.8) by the *receiver*. A similar sequence diagram can be done for the *RPG* configuration.

Comments. (Table 9.3). Each measure $t_1 - t_0$ is an average over *Nb* cycles, which explains the choice point on Figure 9.7. The measures for the *Synchronous* class are not given, since multithread programming does not fulfill the basic principles of the synchronous approach: all inputs and outputs can not be evaluated within the same instant.

Table 9.3. Duration for a full cycle in a multithreaded context: measure $t_1 - t_0$.

Class	*Tool/Language*	*Priority*	*Average in μs*	*Standard Deviation*
IDE	Visual/C	RPS	12.7	0.25
Ext. UML	Rose RT/C	RPS	38.8	0.4
Ext. UML	Rose RT/C++	RPS	44.5	0.4
QoS	ARTO/C++	RPS	61.2	0.8

For the *IDE* class (*Visual C*), we made several implementations to enlarge the comparison:

- Using the standard *Windows messaging* to communicate between the sender and the receiver: the average time for the complete cycle is of 22.0μs for the *RPS* configuration;
- Using a homemade message queue whose concurrent access were protected by a *Windows mutex*: the average time for the complete cycle is of 30.9μs for the *RPS* configuration;
- Using a homemade queue whose concurrent access were protected by a critical section: the average time for the complete cycle is of 12.7μs for the *RPS* configuration.

We choose to mention the critical section figures in Table 9.3 because both *Rose-RT* and *ARTO* use the same kind of protection to manage their own mes-

sage queue. In fact, these few figures clearly show that technical choices have a great incidence on the timing performances.

Table 9.3 provides only the figures for the *RPS* configuration. The results are quite similar for the *RPG* configuration since the only thing to do in the final part, labeled A on 9.8, is to assign a value to an array, which leads to a marginal temporal costs. This final part is measured in a *RPG* execution because the treatment of the *processing()* message is not split.

4.3 Discussion

It is not worth focusing on the precise timing values, but simply on their rough estimate, because they are slight modification-sensitive. The use of system calls during the interval of measure, the compiler's options, the choice of libraries etc. change the results, but fortunately preserve the rough estimate.

Measures related to the full cycle duration are unambiguous, but it is not advisable to hastily generalize from the industrial example measures. As transitions hold few instructions, the whole CPU time is consumed by the messaging system. A detailed analysis of message sending shows that in fact, CPU time is spent running system calls, e.g., to manage the critical section needed to left messages in a shared queue (see comments above). That explains the best results for both classes *IDE* and *Synchronous* in the industrial example, which does not require concurrency. This remark is confirmed by the following: the *IDE* class is twenty times quicker than the *Extended UML* class (*Rose-RT C*) for the industrial example, but only three times quicker for the full cycle measure of the sender / receiver example, which takes place in a multithreaded execution context. These results are congruent with [Becker et al., 2001]. It is worth to notice that, in practice, an overhead of 30μs can be negligible with treatments covering milliseconds. The *QoS* class is always slower due to additional system calls (multithreading, events, data sharing for scheduling, etc.), but also to CPU time spent providing services such as scheduling and adaptability. Due to the difference in the provided services, it may be advised to use COTS CASE tools for the development of new software, since the timing performances of the automatically generated code is not as awful as the embedded developers often claim it, in particular in a multithreaded context.

5. Conclusion

This work mainly was aiming to deal with two issues. First, "*To what extent models written during development are platform independent?*". In fact, models are dependent on tools due to their specific extensions, and moreover they are dependent on the target programming language. On the other hand, the virtual machine supplied by tools is not (cannot be) perfect but hides most platform specific details. Of course, when a service is not supplied the devel-

oper has to add a library or some additional code. Therefore, in addition to the initial question, tool dependency should be taken into account when starting a MDA approach and may be more important than platform dependency.

The second issue was: "*To what extent does automatic code generation reduce timing performances of applications?*" The answer depends on the context. The automatically generated code for programs running in a single thread is meaningfully slower. For multithreaded programs, the gap cuts down, and the advantages of a model-centered approach are to take into account. Beyond performances, the positive aspects of object-oriented approaches and tools are well known. On the one hand it would be very expensive to implement, for a specific application, the services supplied by tools such as property proof, graceful degradation or dynamic scheduling. On the other hand the benefits of modeling are widely acknowledged.

Finally, as expected there is no best choice. First the tool is to choose according to the available platforms. Then the emerging choice criteria based on performance are:

- To choose a tool according to the required services, when CPU resource is not a restrictive factor, but also when messaging system is insignificant compared with operation execution time (according to the execution context).

- To choose, when resources are critical, either the *IDE* or *Synchronous* classes for monothreaded applications or the *IDE* class for multithreaded applications.

References

T. F. Abdelzaher, E. M. Atkins, K. G. Shin: *QoS Negotiation in Real-Time Systems and its application to Automated Flight Control*, IEEE Trans. on Computers, Vol. 49(11), pp.1170-1183; 2000.

C. André, M.-A. Peraldi-Frati, J.-P. Rigault: *Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems*. UML 2002, LNCS 2460, (p.163-178).

L. B. Becker, M. Gergeleit, E. Nett: *Approach for Implementing O-O RT Models on Top of Embedded Targets*. OMER-2 - Workshop on O-O Modeling of Embedded RT Systems; 2001.

G. Berry: *The Foundations of Esterel*. in Proof, Language and Interaction: Essays in Honour of Robin Milner, ed. G. Plotkin, C. Stirling and M. Tofte, MIT Press; 2000.

I. Cardei, R. Jha, M. Cardei, A. Pavan: *Hierarchical Architecture for Real-Time Adaptive Resource Management*. Middleware 2000, LNCS 1795, pp. 415-434; 2000.

J.L. Contreras, J.L. Sourrouille: *A Framework for QoS Management*. TOOLS'39, USA, IEEE press (pp.183-193); 2001.

Esterel-Technologies: *Esterel studio Version 3.1.6*. `http://www.esterel-technologies.com`.

N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud: *The synchronous dataflow programming language Lustre*. Proceedings of the IEEE, 79(9): 1305-1320; 1991.

I-Logix: *Rhapsody Version 4.0.1*. `http://www.ilogix.com`.

IBM-Rational: *Rose Real-Time Version 2002.05.01*. `http://www.rational.com`.

I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Addison-Wesley; 1999.

L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar (eds.). Worshop on Consistency Problems in UML-based Software Development. RR 2002:06 Blekinge Institute Of Technology; 2002.

C. Lee, D. Siewiorek: *An Approach for QoS Management*. CMU-CS-98-165; 1998.

MDA: *Model Driven Architecture*. Document number ormsc/2001-07-01, Architecture Board ORMSC; 2001.

MDA: *MDA Guide Version 1.0.1*. Document number omg/2003-06-01; 2003.

M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber: *Capability Maturity Model for Software*. Version 1.1,CMU/SEI-93-TR-24, DTIC Number ADA263403; 1993.

SPT: *Profile "Schedulability, Performance and Time"*. Final adopted specification available at OMG: `http://www.omg.org`; 2002.

UML: *OMG Unified Modeling Language Specification*. Version 1.5; 2003.

UML: *Unified Modeling Language (UML) Specification*. Version 2.0; 2004.

Chapter 10

UML-EXECUTABLE FUNCTIONAL MODELS OF ELECTRONIC SYSTEMS IN THE VIPERS VIRTUAL PROTOTYPING METHODOLOGY

P.F. Lister, V. Trignano, M.C. Bassett and P.L. Watten
Centre of VLSI and Computer Graphics,
University of Sussex,
Brighton, BN1 9QH, UK
Tel: +44 1273 678050
Fax: +44 1273 678030
P.F.Lister@sussex.ac.uk

Abstract

This paper presents the use of UML-Executable Functional Models (UML-EFM) in the context of the ViPERS virtual prototyping methodology [Lister et al., 2004a, Lister et al., 2004b] for System-on-Chip design. The concepts, the implementation and the experiments presented in this paper were developed at the University of Sussex (UoS) in the Centre of VLSI and Computer Graphics as part of an EU project [VIPERS]. The ViPERS methodology and its employment of the executable functional models have been developed to face the contemporary challenges of System-On-Chips by integrating key design methodologies with the graphical and interactive features of virtual prototyping. The fast evolution in silicon technology and its consequences on the market of hand held electronic products, is making the adoption of new design methodologies mandatory, with modern techniques for the design, development and manufacturing of consumer electronics. Executable functional models provide a means to simulate the target device in different phases of the design flow and analyse its requirements (behaviours, interfaces, etc), architecture (HW/SW partitioning) and finally its digital implementation. A key contribution includes the combination of an interactive 2D photorealistic model with its functional executable model implemented as a UML state machine; the experiment is applied to an RF home-based remote control used to control a cooking stack.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 161–178.

Keywords: Virtual Prototyping; UML; SystemC; executable specification; handheld devices; SoC modelling; ViPERS methodology.

1. Introduction

A close look at the market of consumer electronics reveals that nowadays a significant slice of it is occupied by hand-held devices. The rapid advance in silicon technology is enabling a substantial increase in the number of transistors per chip [ITRS, 2003]. This growth in complexity is parallel to other phenomena, for example the shortened time to market, and the high competition among manufacturing companies. Designers and engineers are therefore facing the dilemma of having to produce highly technological and complex systems in a limited time. To reduce the gap between complexity and time to market new design methodologies are being proposed. The ViPERS methodology links key trends in SoC design with modern interactive and graphical features of virtual prototyping. At the heart of this methodology is the desire to test virtual prototypes of electronic products at different stages of the design, development and manufacturing processes.

The first step in the ViPERS methodology is the analysis phase and the consequent derivation of an UML-executable functional specification. It is clear from the research in the field of requirements and specification development [RUP] that the specification work is unlikely to be confined to the period before implementation begins. Determining accurate product requirements and specifications is a vital stage in the development of a commercially viable device and executable functional models can help extend the value and meaning of the requirements and specification phase to further ensure the validity of this work prior to implementation [Kimura and Verlag, 2002]. Hence there is a need to rapidly feed changes in the requirements into the implementation tool chain in an evolutionary way. It is common for a design house to be given a written specification for a prototype device. Often the specification is not complete enough for the first resultant prototype to be satisfactory to the client, resulting in some design iterations. If the design house were to build a virtual prototype or even several alternative schemes, the client can clarify the functional specification before any hardware or software is built. The virtual prototype is a form of communication and reference in addition to the functional specification and any other requirements of the design [Preece et al., 2002]. A key aspect being highlighted is to ensure that effort spent in the early product definition phase should be reused as much as possible in the later implementation and test of the device. Several methods of requirements gathering have been explored including traditional written reports and UML based tooling. UML [OMG, 2003] provides the means to document detailed requirements which can lead, with the aid of software tools such as Rational Rose RealTime and

the use of state machines, to the production of the first behavioural model of the electronic device. Rational Rose RealTime is built on the UML-RT profile [Selic and Rumbaugh, 1998], which, due to its limited architecture and performance modelling capabilities, should be considered complimentary to the UML Profile for Schedulability, Performance and Time [OMG, 2002] (also called the Real-Time UML Profile) standardised by the Object Management Group (OMG). Rational Rose RealTime was chosen upon other UML real-time software tools because of the intention by Rational Rose to implement a SystemC profile [Sardini, 2002], which would simplify the route to hardware for the ViPERS methodology.

If the graphical model has been implemented at this stage then the requirements can be explored through the connection of the graphical model to its behavioural correspondent in UML as shown in Figure 1 and feedback from the stakeholders can be gathered.

Figure 1 shows the refinement steps related to the ViPERS methodology and the consequent creation of virtual prototypes that result from the four main phases (analysis, design, implementation and test). Virtual prototypes are distinguished based on which phase of the design flow they are generated from, and therefore which features of the target device they incorporate; each virtual prototype implements an EFM. The virtual prototypes are:

1 Functional Prototype; this is a product of the analysis phase, where the requirements and specification of the target product are analysed and defined.

2 Architectural Prototype; this is a product of the design phase, where architectural design takes place and hardware/software partitioning is defined.

3 Digital Prototype; this is a product of the implementation and test phases, where all the hardware and software blocks that constitute the target electronic device are implemented and then tested.

To explore the use of UML-EFMs in the context of the ViPERS methodology this paper presents the combination of an interactive 2D photorealistic model with its functional executable model implemented as a state machine in UML with the support of Rational Rose RealTime. The SxUMLSocket Package is employed to link the state machine of the EFM to its correspondent graphical model. Communication between the functional and the graphical model conforms to a XML-like communication protocol which defines the criteria that models need to be consistent with, in order to establish a connection with each other and communicate.

Traditional UML techniques are used to explore the requirements of the target electronic product. Once the system has been specified purely by means of

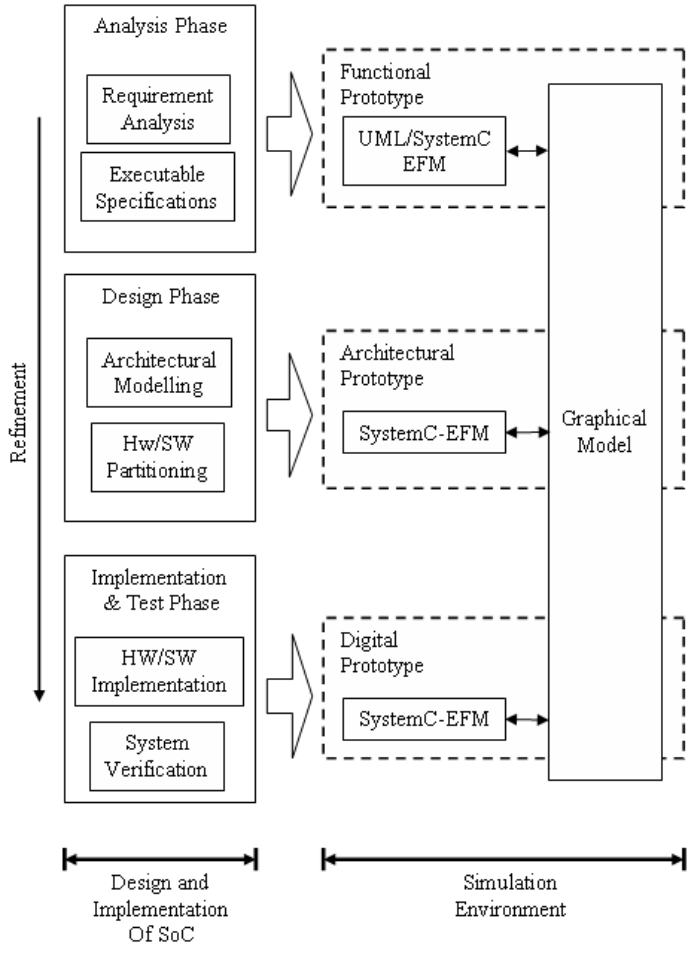


Figure 10.1. EFMs allocation in the ViPERS methodology.

UML diagrams (class, sequence, use-case, activity, etc), Rational Rose Real-Time is used to implement a state machine that describes the functionality of the system. The ViPERS methodology suggests the use of UML state machines for the creation of the functional model in this analysis phase. The model can optionally be described in SystemC [SystemC], using the freely downloadable toolkit or taking advantage of the graphical interface and automatic code generation that software tools as CoCentric System Studio from Synopsis offer.

The advantages offered by the use of UML-EFMs are clear when the simulation environment is running and the functional model is linked to the graphical model; the first analysis of the functionality of the target product can then commence. Designers as well as investors, hardware engineers as well as end-users with no technical background can test the usability of the product; interaction is achieved, for example, by pressing a button or moving a slider on the graphical model and viewing in real-time the changes on the display or other output means. The designer can also view the progress of the simulation through the graphical state machine at run time, and track the changes between states that result from the interaction with the graphical model implemented in VDM [Lister et al., 2004a] (virtual device model). VDM is an integrated development environment which enables designers to create photorealistic models of the target electronic product and define its interactivity through the scripting of the graphical objects that constitute the model.

The interaction of the executable functional model with the graphical model is a very effective approach to test features such as the graphical user interface and user interactivity issues of the device prior to any implementation. The two elements that make up the virtual prototype (graphical and functional model) can be packaged together in order to be distributed between stakeholders or end-users for feedback.

2. Executable Functional Model (EFM)

For the purpose of this paper we define the Executable Functional Models as descriptions of a number of properties associated with the functionality of an electronic system. They are independent executables which contain the means to communicate with an external application for simulation purposes. The nucleus of the ViPERS prototyping environment is VDM. Figure 1 shows that executable functional models connected to a VDM graphical model constitute the simulation environment of the ViPERS methodology.

The aim of the executable functional models is to simulate the behaviour of the virtual prototype (functional, architectural and digital) through the design and implementation stages. Based on a modified version of the ROPES process [Douglas, 1999], as shown in Figure 2, the ViPERS methodology with

its environment provides the specific tools, libraries, packages, and services needed to connect EFMs to the target graphical model.

EFMs are developed to test various features of the target electronic product; features include: Itemized lists:

- Functional properties,
- Graphical User Interfaces and User Interactivity properties,
- Architectural issues such as HW/SW partitioning,
- Digital properties, explored through the verification of the completed hardware/software implementation.

The ViPERS methodology currently provides support for two types of EFMs, these are:

1 UML-EFM, which use Rational Rose RealTime and Visual Studio version 6.0.

2 SystemC-EFM, which uses either:

   (a) The SystemC free-toolkit and Visual Studio version 6.0 (Windows-based),and

   (b) CoCentric System Studio from Synopsys (Linux-based)

The first type of SystemC-EFMs was demonstrated in [Lister et al., 2004b] and uses a SystemC-container application, developed by the ViPERS team, for the communication of the SystemC free toolkit with the VDM model. The second type uses a SystemC library for CoCentric, SxSockets Library [Trigano et al., 2003] and a Linux-based application server - the local communication control service (LCCS). Drawbacks and advantages of the different approaches in different stages of the methodology will be described in the following section of this paper.

The communication framework which allows real-time simulations provides a fast interaction mechanism between the functional and graphical model; the framework relies on the passing of small tagged messages. Further considerations on time-related issues for the simulation are explored later in this paper.

2.1 Executable Functional Models in UML (UML-EFM)

UML-EFMs are allocated in the analysis phase of the ViPERS methodology to take advantage of the many aspects that a description language such as UML. With its vast tooling support, it can bring to the definition of a system when details of the implementation have yet to be defined [Douglass, 1999]. The models are developed as state machines using Rational Rose RealTime,

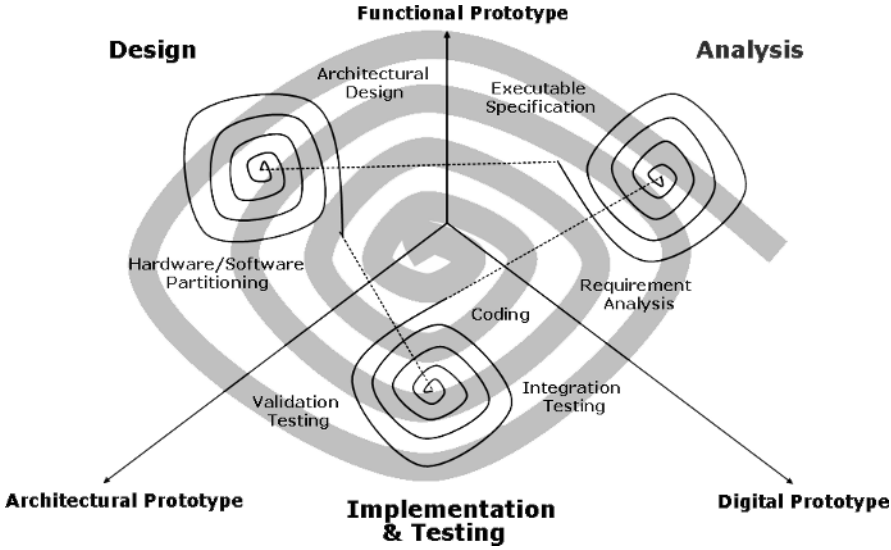


Figure 10.2. ViPERS modification of the ROPES process.

which provides a familiar framework for windows programmers. State machines are defined graphically and C++ code is added to the various states and transitions. The model created represents a very high level description of the system, the state machine in fact is supported by various diagrams (class, structure, sequence, etc), as well as communication protocols to communicate with other threads. UML-RT profile also introduces the use of capsules, which are differentiated from classes by their dynamic behaviour; capsules are active objects that represent system components, their internal behaviour is defined by state machines and they communicate with each other through stereotyped objects called ports which implement interfaces. The suggested approach for the development of a functional description is to create the various classes and diagrams which represent the initial high level description of the system. Once the first design of the system is achieved, it can be refined by adding attributes and operations that will be eventually used for the behavioural description of the model. A capsule is then developed to incorporate the classes and give dynamic support to the system; the behaviour is described with the use of state machines. The communication protocol of the capsule needs to be defined so that the designer can commence testing the capsule by injecting signals at simulation time. As soon as the model functions as expected the SxUMLSocket package is integrated into the system to provide the means to communicate with outside applications. The integration of the two capsules and the structure of a basic UML-EFM are shown in Figure 3.

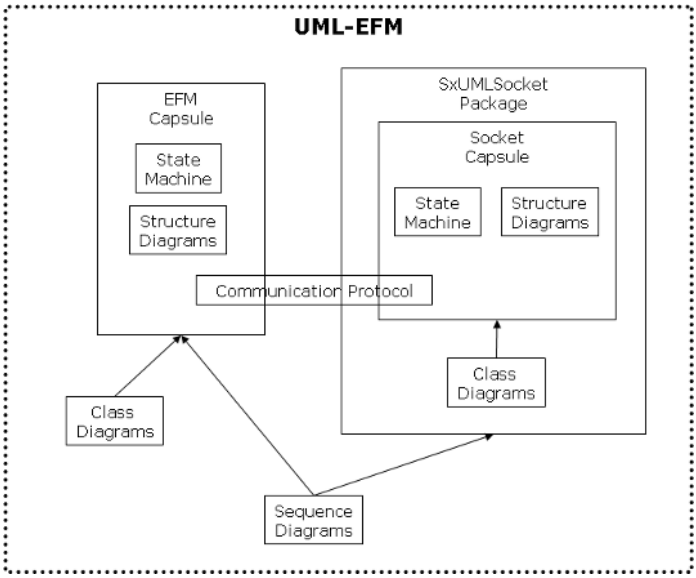


Figure 10.3. UML-EFM structure.

At this stage the functional model is an independent executable which is ready to communicate with the VDM graphical model. The state machine can be tested through enhanced visual means allowing users to interact with the graphical model of the electronic model and view how the state machine processes the input and output. The designer no longer needs to follow the simulation through a visual representation of the state machine, which requires a technical knowledge, but can actually interact with the virtual product as the end user would.

Changes on the display of the graphical model or other outputs that the model might possess are now the means by which the designer tests the functionality of the target product.

Some of the benefits of using a UML state machine are:

- Easy to implement
- Fast to simulate
- Can be used as an independent executable
- Parts of the state machine can be reused in later stages of the implementation

The power of this approach during the analysis phase of the methodology is substantial. UML-EFMs provide a very high level description of the target device, plus Rational Rose RealTime provides various tools to help the designer

in the development and debugging stages. Designers are provided with all the means to test the usability of the target system, graphical user interfaces and interactivity issues; the model can easily be modified or new ones can be created to test different possibilities or to fix unexpected behaviours. Ideally designers would develop a set of possible candidate solutions for the target product, from which one would be chosen as the final design. The life of UML-EFMs in the methodology is not over at this stage and there is a good possibility that the designer will need to come back to it when technical constraints or unexpected bugs make mandatory the redesign of certain features. The nature of the model provides fast and ease means to achieve this goal. UML-EFMs have another two possible major uses in the ViPERS methodology, these are:

1 Automatic translation to SystemC-EFM; issues related with the translation mechanism are outlined in the final section of this paper.

2 Reuse of state-machines parts in later stages of the design flow as embedded code; this is highly dependent on the implementation of the state machine of the EFM, however Rational Rose RealTime provides the means to create code for both platform-specific models (PSMs) and platform-independent models (PIMs).

SxUMLSocket Package. SxUMLSocket Package is a UML Rational Rose RealTime package that provides the classes, the communication protocol, and the capsule needed to link the state machine describing the behaviour of the system to an external application. The package was developed using Rational Rose RealTime and Visual Studio version 6.0, and the generated code is C++. The package includes the class diagram, the structure diagram, the sequence diagram, and the state diagram that visually describe static and dynamic features of the socket capsule.

Figure 4 shows three type of UML diagrams:

1 Class diagrams; which show the classes and their relationship, with relative attributes and operations, that are used in the socket capsule to implement the server and its functionality. The class diagram contains all the classes needed to support the socket capsule plus other classes to support new developments. The class diagram includes:

- Socket Capsule Stereotype. It is a stereotype capsule which represents the active object of the TCP/IP socket and therefore implements its behaviour.
- Sock Class. It is the base class for both the server and the client class.
- Server and Client Classes. They define the functions and attributes that are specific for the server and the client implementation.

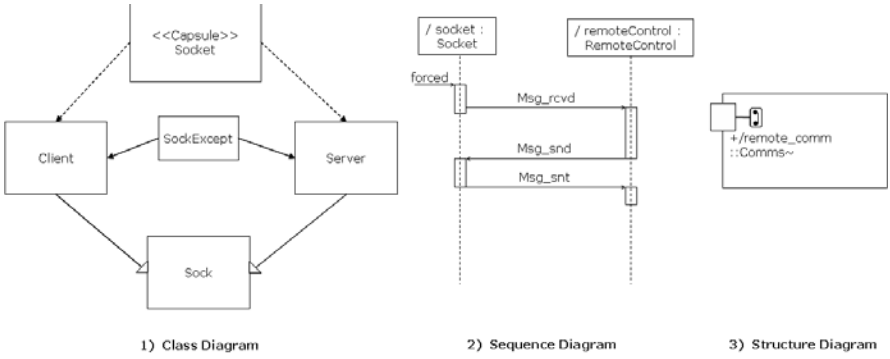


Figure 10.4. SxUMLSocket Package, sequence and structure diagrams.

- SockExcept Class. This class defines the exception cases for the socket.

2 Sequence diagrams; these show the sequence of events, by the passing of signals, between the socket capsule and the functional capsule (i.e. the capsule that describes the behaviour of the target device)

3 Structure diagrams; which show the structure of the socket capsule. The diagram shows the presence of a conjugate wired end port (*remotecomm*) which represents the means for this capsule to communicate with other capsules.

The socket capsule is implemented as a non-blocking server thread and the dynamic features of it are described in its state machine, visually shown in the state diagram of the capsule. Figure 5 shows that the state diagram consists of one state and two transitions; the "waiting for messages" state, the "initial" transition, and the "sending message" transition.

The operation of the socket capsule was purposely kept simple to allow designers to change features when needed. After the initial transition, determined by the initialisation of the socket capsule, the state machine enters the state and waits through a non-blocking receive function call for messages from the client model (VDM).

The non-blocking feature allows the capsule to perform other operations while waiting for the client to send a message. Once the socket capsule receives a message, it sends it to the EFM capsule to be processed by its state machine. The sate machine will output a signal carrying the message to be sent back to the VDM model, this signal will trigger the socket capsule to go through the "sending message" transition, which will force the socket capsule to send

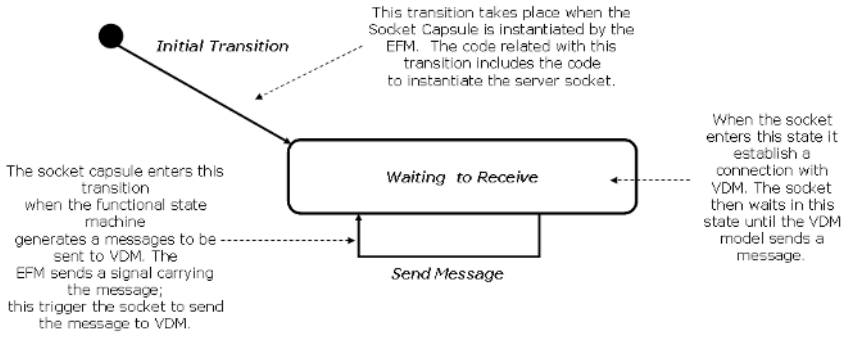


Figure 10.5. SxUMLSocket Package, state diagram.

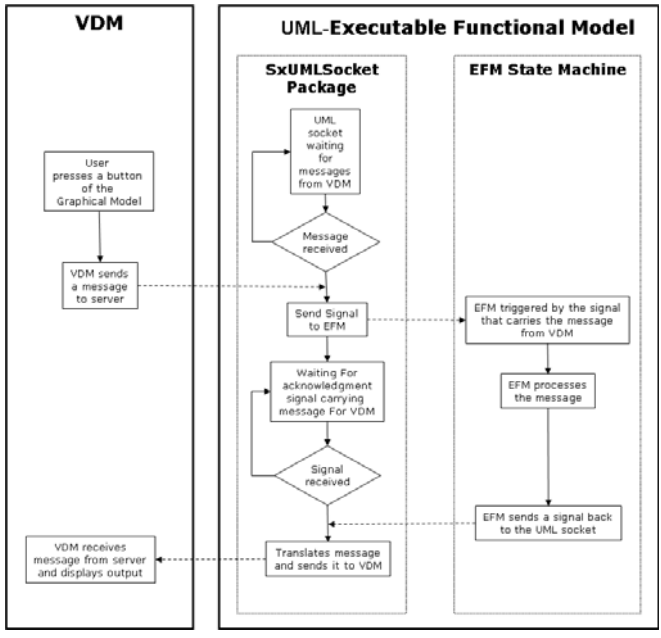


Figure 10.6. Interaction between UML-EFM and VDM.

the message to the graphical model. Details of the sequence of events and interaction that take place during a simulation between the two capsules (socket and EFM), and the VDM model are shown in Figure 6.

3. Comparing EFMs

This section compares the use of the UML-EFMs previously described, with the SystemC-EFMs in the context of the ViPERS virtual prototyping methodology. The aim of this comparison is to justify the specific allocation of each model in the methodology and to reason a suggested approach over another one in different phases. At present, the only SystemC-EFM that have been built and tested by the ViPERS team and therefore supported in the ViPERS prototyping environment are implemented in behavioural SystemC.

Previous work included the implementation of executable models in SystemC [Vanderperren et al., 2002] with the aid of UML diagrams. However, UML was not used to construct executable models, but only for architectural modelling and therefore to demonstrate that the design could meet the requirements. One of the drawbacks of that approach was that engineers needed a UML background in order to understand the design. In our approach the UML model serves two purposes; first it can be used to graphically describe the system, in which case a UML knowledge is needed, and secondly it can be used in conjunction with a graphical model to execute its functionality in a real time simulation. In the second case no knowledge is needed; the user simply interacts with a photorealistic representation of the target product and studies its behaviour.

UML-EFMs present substantial differences with SystemC-EFMs. The development and use of UML-EFMs is almost entirely devoted to the first phase of the methodology, the analysis, while the SystemC model can be refined to RTL level within the same environment. A SystemC model offers the advantage that it can be implemented at very different levels of abstraction, and each level can be allocated in the methodology as part of the refinement process. Another advantage (a consequence of the previous one) is the reusability of the SystemC-EFMs; these models can ideally be reused from their first high level implementation down to the timed (cycle accurate) models, by the refinement process shown in the design flow for SoCs in Figure 1. But UML-EFMs have two major advantages over SystemC-EFMs, which drove the ViPERS team to explore their use in the analysis phase. These advantages are:

1 Ease of implementation. UML-EFMs are very easy to implement and to modify which makes it a much better candidate when, in the analysis phase, designers need to quickly put together and test the device and possibly implement modified version of it.

2 Simulation speed. UML-EFMs provide real-time simulation speed, which can ideally be achieved only by a high level SystemC description; that would strongly depend on the framework used (free-toolkit, CoCentric, etc) and on the implementation style.

3 Industrial Standard. UML, as well as SystemC, descriptions conform to an industrial standard and therefore are not tied to any particular proprietary tool.

The ViPERS team encountered considerable speed limitation when trying to simulate a behavioural SystemC-EFM implemented in CoCentric; considering the issues related with the different platform (Linux), the model was similarly re-implemented using the SystemC free-toolkit and the container application [Lister et al., 2004b]. The new model performed better giving a near to real-time simulation performance, but still not comparable with the simulation speed of a corresponding UML-EFM.

From these comparison considerations we concluded that the best possible solution for the ViPERS prototyping methodology was to employ both models in different phases in order to maximise their qualities and minimise their weakness. However different routes from this are possible.

4. UML-EFMs and ViPERS Virtual Prototyping Methodology

The final aim of EFMs is to be used in the context of a graphical simulation, where the user does not need to know any detail about the underlying processing of the EFM. At this stage of the development the benefits that virtual prototyping can bring to standard SoC design methodology can be appreciated. In the experiment presented here we linked together a UML-EFM of an RF remote control to its graphical counterpart; the simulation with both VDM and Rational Rose RealTime are shown in Figure 7. The experiment started with a design team writing a requirement document for the remote control. The document was simply a written document that attempted to describe the functionality of the device and all the possible action the user could undertake. It was clear from the start that an imagination gap was created between the understanding of the reader and the conceiver; another phenomenon was the fact that some scenarios were accidentally missed by the author of the document when trying to imagine all the possible scenarios.

Taking into consideration the popularity of UML in the field of requirements and specifications we produced a UML-type description document of the system; the description was more detailed and some ambiguity that natural languages can easily introduce was eliminated but the imagination gap was still not bridged. As described earlier in this paper, the adoption of UML to describe a system has an advantage in the fact that it is an industry standard

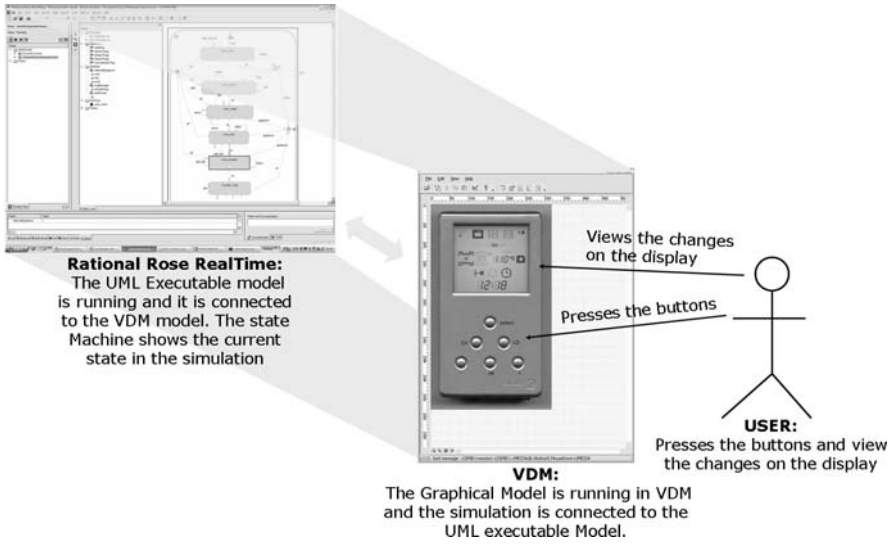


Figure 10.7. A Virtual Prototyping Experiment.

but a drawback since a UML knowledge is needed in order to understand the system. For this reason we decided to create a UML description that could be executed, so that the system could be seen running while interacting with a visual model, which represents the input/output means of the UML model. The UML description was then used as the basis for producing the UML-EFM of the device, while graphics designers were developing the photorealistic model of the remote control and determining its interactivity means.

The two models were then simulated, producing a fully functional virtual prototype, which is referred to as a 'functional prototype' in the context of the ViPERS methodology (to be distinguished from 'architectural and digital' prototypes shown in Figure 1). The team was able to test the behaviour of the device by interacting with the graphical prototype, pressing buttons and viewing the display updating. Considerations were made on the functionality and interaction means of the remote, and the imagination gap was eliminated. Figure 7 shows a use-case type diagram, where a user is interacting with the remote control. At this stage of the simulation anyone can test the functionality of the remote control since the mean is simply an interaction with the photorealistic model. Figure 7 also shows that the graphical representation of the remote control is linked to a state machine. The state diagram shown in the figure is running on Rational Rose RealTime and provides the functionality to the graphical model. Users with a knowledge of UML are also able to follow the simulation from the state changes and transitions in the state machine of the remote control.

The practical experiment illustrated in this paper was conducted at the University of Sussex. Both the functional and the graphical model were built and simulated on a windows-based PC with a 2GHz processor. The models communicate through TCP/IP sockets and therefore they can run on separate machines on a network to take advantage of the processing power; however the experiment showed that in this particular case this was not necessary due to the simple nature of the functional model.

5. Conclusions and Future Work

The experiment demonstrated the benefits that UML-EFMs can bring in the analysis phase of an electronic product. The advantages of using an industry standard such as UML were realised from the early stages of implementation. The standard diagrams provided the means for communication and understanding between the members of the team when trying to establish the requirements of the product. The major benefit though was introduced by the simulation of the EFM in conjunction with the VDM graphical model. The simulation environment provided all the means for the team to test the behaviour, the graphical interface and the input/output means of the target product. In the experiment only the final virtual prototype is showed. This was the result of many changes and iteration of both the graphical and the functional features of the initial prototype. Comparisons with the SystemC models enabled the team to establish the strength and the weakness of each approach and therefore locate their use in a specific stage of the methodology.

The requirement for virtual prototyping to become part of a wide ranging stakeholder evaluation process has driven the development of the ViPERS methodology. Early feedback from the application example suggests significant improvements can be made once stakeholder interaction is possible. The visual realism of the remote has allowed discussion beyond the engineering domain to more diverse stakeholders such as marketing and users. Although more rigour is required to evaluate the benefits, it seems the narrowing of the leap of imagination required to visualise the final product is a positive feature of this approach.

Inevitably the ViPERS environment contains similar features to existing virtual prototyping tools [Cybelius Software, Alita, RAPID], but our focus is on:

- The nature of the graphics (photorealism)
  - Alpha blending
  - Sub-pixel investigation
  - Special rendering
- Route to hardware

Our longer term research will be to apply the virtual prototyping of embedded systems to the entire system environment.

The ViPERS environment successfully demonstrated the viability of linking virtual prototyping to a SoC methodology. At present the linking can be applied at every level but simulation might fail to be in real-time for low level implementations. Further research will focus on the simulation of complex systems and RTL implementations, and therefore deal with concurrency, time constraints, scheduling and performance issues; emulation might be used to speed simulation with low level models.

The ViPERS environment and the use of UML-EFMs successfully increased the capability of separating user interfaces from behaviour; this allows testing different interfaces without having to modify the functional description of the prototype.

It is intended by the ViPERS team to further explore the link between UML and SystemC for platform based designs. Companies such as Rational are predicting an imminent development of a UML.SystemC profile [Sardini, 2002]. It is included in the future work the development and design of other EFMs at different abstraction levels. In particular the ViPERS team will implement Transaction Level models (TLM) and Register Transfer Level (RTL) models, perhaps with mixed HDL languages. TLMs will be produced to prove how a substantially detailed model can still produce fast simulation in the ViPERS environment, while RTL models will prove the link of the methodology down to synthesisable code.

Future work includes the integration of the simulation in virtual environments [Lister et al., 2002] in order to enhance the experience and obtain a more natural interaction of the user with pervasive electronic devices. Graphical models (2 and 3 dimensional) will be placed in their natural environments, where users will be able to navigate through and interact with the devices and view the changes not only through device outputs but also through the changes in the virtual environment (e.g. the light in the oven being switched on).

Acknowledgments

This work was funded as part of the ESPRIT framework 5 VIPERS project IST-2000-30023. We are grateful to our project colleagues for their constructive interaction, particularly Javier Mendigutxia of IKERLAN SA. The cooperation of our colleagues Teresa Riesgo and Eduardo de la Torre of Universidad Politécnica de Madrid and Sabastian Pantoja of Celestica Valencia is acknowledged. System TM is a trademark of the Open SystemC Initiative. CoCentric® System Studio is a registered trademark of Synopsys, Inc. Windows XP®, Visual Studio® and Visual Basic® are a registered trademark of Microsoft Cor-

poration. Rational® Rose Real Time Studio is a product of IBM® Kylix TM is a trademark of Borland® Software Corporation

References

Altia, Inc., `http://www.altia.com` [last accessed 29/07/04]

Cybelius Software, `http://www.cybelius.com` [last accessed 29/07/04]

Douglass, B.P. *Doing Hard Time: Developing Real-Time Systems using UML, Objects, Frameworks and Patterns*, Addison-Wesley, 0201498375, 1999.

Douglass, B.P.*Real-Time UML Second Edition, Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 0201657848, 1999.

International Technology Roadmap for Semiconductors (ITRS), 2003 Edition.

Kimura, I. and Verlag, S. *Product Development with Mathematical Modeling, Rapid Prototyping, and Virtual Prototyping*, ISBN 3-8322-0896-8, Chapter 1, June 2002.

Lister, P.F. Newbury, P.F. Watten, P.L. Senkoro, L. Dountsis, A. Midha, M. Banerjee, I. Trignano, I. and White, M.*Virtual Reality in Electronic Systems*, Proceedings of 5th International Conference on Business Information Systems, Poznan, Poland, April 2002. Pp. 390-394.

Lister, P.F. Watten, P.L. Lewis, M.R. Newbury, P.F. White, M. Bassett,M.C. Jackson, B.J.C. and Trignano, V. *Electronic Simulation for Virtual Reality: Virtual Prototyping*, Theory and Practice of Computer Graphics 2004 (TPCG04), Southampton, UK, June 2004.

Lister, P.F. Watten, P.L. Newbury, P.F. Bassett, M.C. Jackson, B.J.C. and Trignano, V. *Virtual Reality for Electronic Product Development of Hand Held Devices*, Design Automation and Test in Europe (DATE'04), Paris, February 2004.

Object Management Group, *UML profile for Schedulability, Performance, and Time*, OMG document ptc/03-02-03, Needham MA, 2002.

Object Management Group, *Unified Modelling Language (UML)* – Version 1.5, OMG document formal/2003-03-01, Needham MA, 2003.

Open SystemC Initiative. See `http://www.systemc.org/` [last accessed 29/07/04]

Preece, J. Rogers, Y. and Sharp, H. *Interaction Design, beyond human-computer interaction*, John Wiley and Sons, Inc. ISBN 0-471-49278-7, 2002.

RAPID virtual prototyping tools, e-SIM LTD, `http://www.e-sim.com/` [last accessed 29/07/04]

Rational Unified Process® for Systems Engineering, `http://www.rational.com/` [last accessed 29/07/04]

Sardini, A. *SoC Design with UML and SystemC*, European SystemC, 6.Users Group Meeting, Lago Maggiore, October 2002.

Selic, B. and Rumbaugh, J. *Using UML for modelling Complex Real-Time Systems*, white paper, rational (Object Time), march 1998.

Trignano, V. Bassett, M.C. Watten, P.L. and Lister, P.F. *Extending SystemC for high-level multi-platform SoC simulations*, IEE Postgraduate Colloquium on System-on-Chip Design, Test and Technology, September 2, 2003, Cardiff University.

Vanderperren, Y. Sonck, G. van Oostende, P. Pauwels, M. Dehaene, W. and Moore, T. *A Design Methodology for the Development of a Complex System-on-Chip using UML and Executable System Models*, Forum on Specification and Design Languages (FDL'02), Marseille, France, September 2002.

VIPERS Project references and web pages, `http://www.upmdie.upm.es/projects/vipers/` [last accessed 29/06/04]

III

C/C++-BASED SYSTEM DESIGN

Introduction

Eugenio Villar
E.T.S.I.Industriales y Telecom
University of Cantabria, Santander, Spain
villar@teisa.unican.es

Increasing system complexity demands system-level languages supporting specification and design methods above RT-level. Among the different languages proposed, SystemC is currently the language with a wider acceptance from the design community. SystemC provides the hardware and software, that is, the system development team with an executable specification of the system that can be used to quickly simulate, explore various algorithms, validate and optimize the design. System modeling, simulation and verification at the different abstraction levels are currently the most important applications of the language. Transaction-Level Modeling (TLM) is becoming a wide accepted industrial standard for system modeling and design verification.

Nevertheless, SystemC is committed to provide the modeling framework for a complete design flow from specification to implementation. Achievement of this goal is still the objective of an active research activity. Although SystemC could be the dominant design language in this design methodology, interoperability with other languages would be required.

FDL served again as the main European forum for technical presentations and discussion on system-level languages and their application to electronic systems design. As in previous events, a selection of the best contributions to the CSD workshop have been selected for the FDL'04 book in the series. The five selected papers cover some of the most important research topics to-day.

So, Chapters 11 and 15 address the design of reconfigurable systems. In the former, an extension to SystemC called OSSS is proposed. The objective is to support polymorphism avoiding a pointer-based mechanism. Pointers are problematic to synthesize efficiently and there is no synthesis tool supporting them. A library is provided to simulate the proposed polymorphic objects and a high-level synthesis tool to translate OSSS into synthesizable SystemC. The design methodology allows the programmer to implement a reconfigurable system using a well-known concept such as polymorphism. In the latter, requirements to C++-based languages supporting co-design methodologies for reconfigurable SoC are defined. Extensions to SystemC and OCAPI-XL are proposed supporting those requirements. The use of these languages in three reconfigurable scenarios is presented.

In its current version, SystemC is based on a strict-timed, discrete-event computational model including δ-time transactions. Nevertheless, there is a need for heterogeneous specification under several, more abstract models of computation. Chapter 12 proposes SystemC to model different computational

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 181–182.

models. This is made possible by exploiting the abstraction capabilities of C++. Border processes and channels are proposed to serve as interfaces among different subsystems described using different models of computation.

RT-level description and design using VHDL is a mature, industrial design methodology. Nevertheless, there are still areas for further improvement of current practices. So, in Chapter 13, xHDL, a meta-language improving VHDL providing flexible mechanisms for component customization, instantiation and interconnection is presented. xHDL improves VHDL code description and reuse. A tool has been developed for automatic code generation with parameter selection.

Finally, Chapter 14 addresses real-time (RT) SW modeling, a SystemC application with an increasing interest. The RT behavior of software is managed by a RT operating system (RTOS). One of the main services provided by the RTOS is the implementation of the required multi-task concurrency by applying a chosen scheduling policy. The current version of the SystemC kernel lacks of the required features for a direct modeling of many of the RTOS characteristics. In the paper, the modeling capabilities of SystemC 2.0 are extended to model SW decomposition, dynamic process creation and deletion, process control, preemption, static and dynamic process prioritization and scheduling and inter-process communication and synchronization.

Chapter 11

DESIGNING FOR DYNAMIC PARTIALLY RECONFIGURABLE FPGAS WITH SYSTEMC AND OSSS

Andreas Schallenberg,[1] Frank Oppenheimer[2] and Wolfgang Nebel[1]

[1] *Carl von Ossietzky University Oldenburg*
Andreas.Schallenberg@Uni-Oldenburg.de
Wolfgang.Nebel@Informatik.Uni-Oldenburg.de

[2] *OFFIS Institute, Oldenburg*
Frank.Oppenheimer@OFFIS.de

Abstract This paper presents a new approach to design embedded systems based on dynamic partial reconfigurable FPGAs. The approach is intended to allow designing of systems with runtime reconfiguration without explicit specification by the designer. The design entry point is the HDL OSSS, a SystemC extension allowing for synthesizable object orientation and polymorphism.

Keywords: Embedded Systems, HDL, Object Orientation, SystemC, OSSS

Introduction

Dynamic partial reconfigurable FPGAs (DFPGA from now on) are available in the market for quite some time. They combine characteristics of two worlds since they are hardware on one hand and can be changed after manufacturing like software on the other. Previously designing for a hardware execution model meant having the design fixed after delivery. However the requirement to modify systems after delivery forced designing for software execution models. So the choice was basically between ASIC-like circuits and processors/DSPs. FPGAs in general and dynamic partial reconfigurable FPGAs in particular have blurred this HW/SW borderline. Final design microstructure can now be modified after shipping and the circuit still has the basic advan-

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 183–198.

tages of a hardware execution model like massive parallelism. Examples for such devices are Xilinx Virtex II [Lim and Peattie, 2002] [Xilinx, 2003] and Atmel At40k [Atmel, 2002].

This way of designing reconfigured systems today has a major drawback: Some essential things regarding reconfigurations have to be done by hand, because there is no adequate design flow to support this. The possibility of changing hardware behavior requires the reconfiguration step itself to be designed manually. Current hardware description languages (HDL) do not support this process. The goal of automated synthesis is to avoid tedious and error-prone tasks to be done by the designer. The motivation for this is to save time and to reduce the risk of design flaws. Along with this goes the trend to raise the level of abstraction for design entry and the usage of new HDLs like behavioral level VHDL or SystemC. Consequently the reconfiguration step should be supported by an inherent concept of the language as well.

Recent work

In [Agosta, 2004] and [Bruschi, 2004] an approach to model reconfigurable systems based on the Java programming language is introduced. The dynamic class loading mechanism is used to modify the set of available classes during runtime. The set of classes may be extended or classes may be updated.

For our solution the OSSS [Grimpe and Oppenheimer, 2001] [Grimpe et al., 2002] [Radetzki, 2000] extension to the SystemC [SystemC, 2003] HDL is used as a basis to create language constructs which naturally map to dynamic partial reconfigurable FPGAs. This will be explained in more detail in Section 11.4.

This paper presents a new approach to design for DFPGAs and to make use of their partial dynamic reconfigurability. Characteristics of the invented methodology are:

- It allows for a timing accurate simulation to analyze the reconfiguration process. This is already implemented.
- All language concepts are designed with a hardware semantic in mind although the synthesis is not implemented yet.

Outline

The rest of this paper is organized as follows. The following section gives a motivation for building reconfigurable systems using a design example consisting of real world application benchmarks. Section 2 specifies desired properties for language constructs to support the design process. Afterwards the basic OSSS language constructs proposed in this paper are presented along with a set of requirements the system must meet to be conflict-free. This is followed by

an introduction of more sophisticated methodologies to achieve a more convenient way of describing reconfigurable systems. Section 6 concludes the paper and describes the future work that is forseen.

1. Motivation

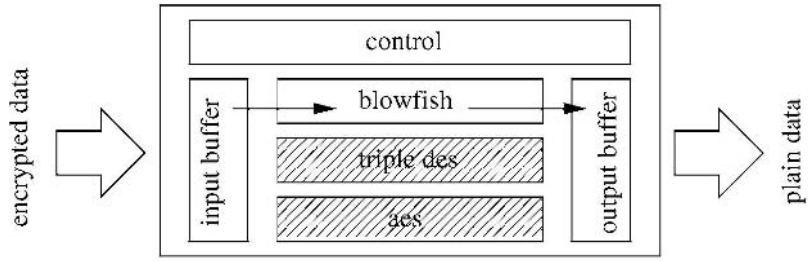


Figure 11.1. Cryptography module implementation using ASIC

On a standard ASIC every functionality which may be needed sometime during runtime is required to be constantly present on the circuit. For a stream based data cryptography module supporting three different algorithms and two different basic operations (encrypt, decrypt) this could lead to a design like in Figure 11.1. It shows a block diagram of a system consisting of input and output buffers, a control circuit and three cryptography submodules: Blowfish [Schneier, 1993], triple DES[DES, 1993] and AES (Rijndael)[Daemen and Rijmen, 1999]. The overall module reads data streams through its input buffer and decides which codec to use on the type of the input data. This input data additionally contains encryption keys which introduce a new data block to be processed and may also lead to a switch in the algorithm used. The shaded boxes indicate unused codecs at a specific point in time, while the blowfish encryption is performed currently. They waste chip area, making the design more expensive and usually more power consuming, too. Using a DFPGAs reconfiguration capability could change the situation as depicted in Figure 11.2. The submodules (Blowfish, DES3, AES) have been replaced by a reconfigurable area holding the single needed circuit. To store the inactive submodules an external memory device (e.g. PROM, Flash, ...) is required.

2. Language support

The lifetime of one specific algorithm in the example is bounded by the time of its creation and its replacement by its successor. It does not make sense to allow accesses to a specific algorithm outside its lifetime. Classes and objects are appropriate language elements to describe those submodules because

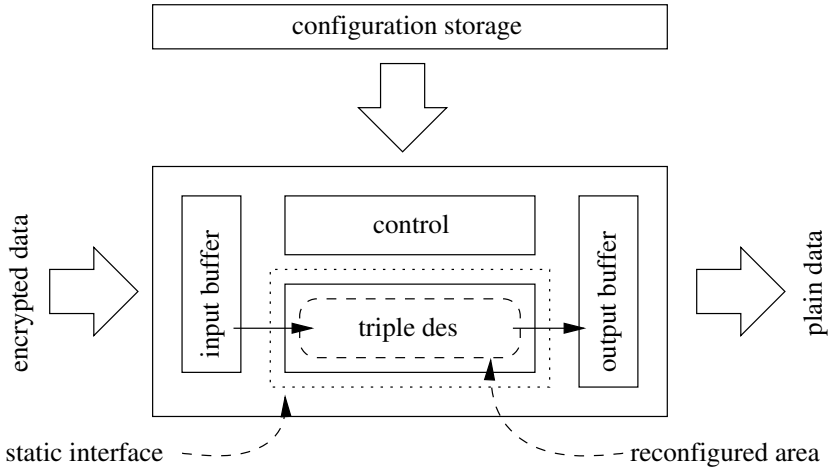


Figure 11.2. Cryptography module implementation using FPGA

the concept of lifetime of objects can be applied to describe the lifetime of an algorithm. Objects are combinations of variables and their associated methods. A second observation is that a certain area within the DFPGA housing the algorithm consists of both logic functionality and storage elements. This is reflected in the object oriented world by objects having both methods and attributes to provide functionality and store data.

Communication between the dynamically reconfigured objects on the DFPGA and other parts of the chip advises a static interface. The reason is twofold: First, from an implementation centric point of view the environment of a dynamic object is static and therefore cannot adopt a different interface to the changing area on the DFPGA. Second, from the modeling point of view polymorphism suggests that the environment may or even should not know which forming the dynamically reconfigured object currently has. Therefore it cannot rely on extensions to the interface introduced by derived classes.

In object oriented languages a common base class may be defining the interface and derived classes may provide the actual implementation. From an environmental point of view the object has the base classes' type but the current implementation originates from a derived class. The root class correlates with the static interface of the object so just knowing its root class only allows accesses though the root classes' methods.

The advantage of this restriction is a simplified synthesis step since the interface of each possible content in the reconfigured area is known to the synthesis tool at compile time. Moreover designing new objects after delivery of the system to the customer and exchanging pattern streams in the configuration

storage would be possible since the old and new objects described would be guaranteed to be interface compatible.

However, even with this restriction applied it is still possible to include method interfaces in the root class which are implemented by a subset of derived classes only. Given that situation knowing the current forming of the dynamic object would allow making use of those methods.

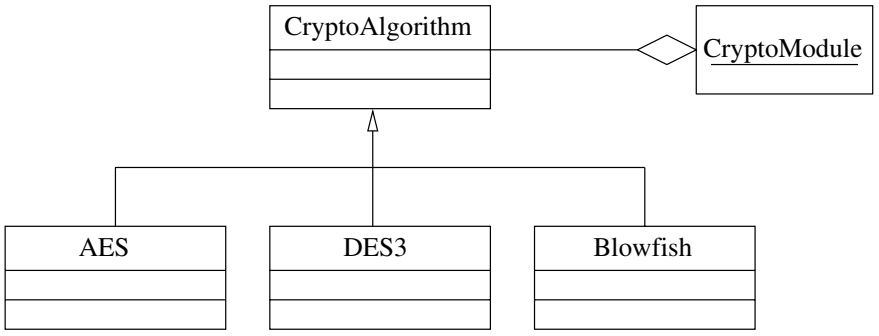


Figure 11.3. Cryptography classes

Figure 11.3 describes one possible class tree of the example system containing three different algorithms each described by derived classes of a common cryptography base class. During runtime objects have to be created and destroyed according to the type of the input stream. Therefore a supervising instance (it will be referred to as RECONFIGURATION CONTROLLER) has to detect which objects are required at a certain point in control flow and to initiate necessary reconfigurations. The goal is to let the designer describe this reconfiguration controller implicitly and not requiring him to implement its behavior.

One extract of this supervising instance's behavior is depicted in Figure 11.4. There are two changes in the incoming data stream detected, namely a switch to a DES stream and afterwards a switch to a Blowfish stream. These events cause reconfigurations of the DFPGA area holding the decoder object.

SystemC and OSSS

SystemC is the only prominent and widely available hardware description language that contains, borought from it's C++ roots, a class and an inheritance concept. It even supports polymorphism using a pointer concept. However, since pointers are problematic[Seméria and Micheli, 1998] [Seméria et al., 2000] to synthesize efficiently and there is no synthesis tool supporting it, the SystemC synthesis subset itself lacks polymorphism. The OSSS language ex-

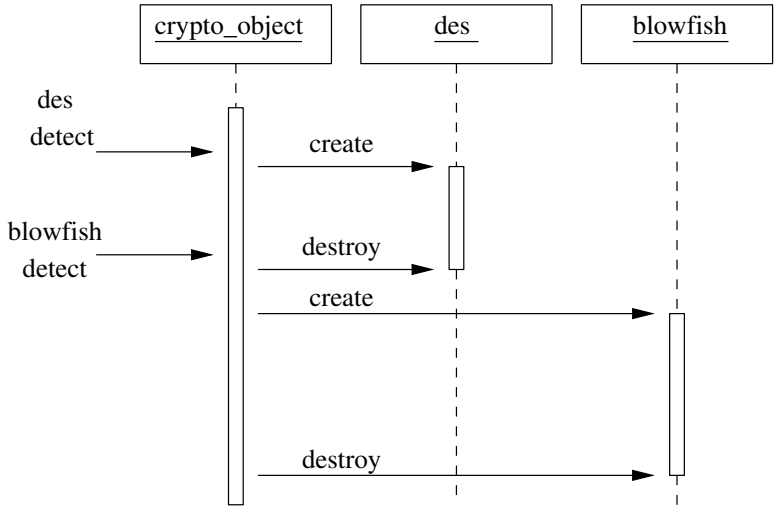


Figure 11.4. Scheduling

tension introduces synthesizable polymorphic objects avoiding a pointer based mechanism. There is a library provided to keep the ability of SystemC to execute specifications and a high level synthesis tool[1] to translate OSSS into register transfer / behavioral level SystemC being processed by commercial tools. Therefore OSSS is an object oriented HDL for which a synthesis tool flow for ASICs is available. The target technology for OSSS descriptions used to be ASICs. Therefore the concept of polymorphic objects needs to be extended to be applicable to DFPGAs and their dynamic reconfiguration ability. Another important aspect of the OSSS language is the concept of shared objects[2] which provide mutual exclusion and arbitration of accesses. They can also be used to provide communication interfaces for different modules within a hierarchy. Describing OSSS in more detail would exceed the available space for this paper.

2.1 Terminology

In this paper the terms *class*, *(member) attribute* and *object* are used like they are common to a C++ programmer. They describe items from a description language point of view. An object is a combination of class type, member attribute values and a unique identification (like a memory address in the software execution model).

From a hardware point of view a *content* of a reconfigurable area is referring to the logic implemented and the signal values within that area of the DFPGA.

The gate logic structure in a reconfigurable area is the pendant of a C++ class definition and the CLB^3 register values correspond to member attribute values.

We will introduce objects which have an ability of being reconfigured. They contain some content object being the current forming of that reconfigurable object. In the crypto example this forming could be an instance of a cryptographic algorithm. It is called *context* of the reconfigurable object. The hardware counterpart of a *context* is a pair consiting of a certain gate logic structure (programming of the device area) along with the register values.

3. Basic language concepts

On top of the OSSS environment additional language constructs are introduced to describe the reconfigurable objects.

ReconObject

First, a statement provides the ability to denote the candidate objects for being reconfigured on a DFPGA. The following code shows the declaration of a class which includes one reconfigurable object within a standard SystemC module:

```
class CryptoModule : public sc_module {
  ReconObject< CryptoAlgorithm > crypto_object;
  ... };
```

Since the basic concept depends on polymorphism it is required to have a common base class for the different algorithms. The base class defines the objects method interface to the environment. This is done with the `CryptoAlgorithm` class which serves as the root class for the `Blowfish` and `DES3` classes containing the desired implementations.

Assignment to ReconObject

The syntax of such an assignment is known from normal C/C++ assignments. The following code fragment shows three examples:

```
Blowfish my_algorithm;
ReconObject< CryptoAlgorithm > my_second_recon_object;
// Three ways for assignments
crypto_module.crypto_object = Blowfish();
crypto_module.crypto_object = my_algorithm;
crypto_module.crypto_object = my_second_recon_object;
```

The set of classes which may be assigned to a `ReconObject`[4] is determined by a common base class which is given as the template argument to the ReconObject. Note that it is possible to assign a reconfigurable object to another

one. Assigning any object of one class to a ReconObject which currently holds an object of a different class is equivalent with an explicit reconfiguration request. To be brief we say that the ReconObject is being reconfigured. This means that a specific area represented by that ReconObject is going to have a new logic structure. Assigning an object of the same class causes the ReconObjects attribute values to be replaced but does not lead to a reconfiguration of the ReconObject. This is also known as a separation into class switches and object switches in literature [Noguera and Badia, 2002].

The C++ polymorphism is implemented using pointers which do not need to have types that match the exact types of the objects they are pointing to. Assigning such variables to each other causes the addresses to be copied and not the objects contents. Assigning polymorphic objects in OSSS means copying its content, not a reference to it.

Access to ReconObject

Accesses to ReconObjects are done using one of those two statements:

```
RECON_OBJECT_PROCEDURE_CALL( crypto_object,
                             encrypt(dataword) );
RECON_OBJECT_FUNCTION_CALL( crypto_object,
                            encrypt(dataword),
                            return_value );
```

The first argument is the ReconObject itself, the second one the method call and the third one a return value, if the method provides one.

Consistency criteria

There are several possible situations in which a reconfigurable system may run into conflicts since there are limited resources (e.g. only one reconfiguration port per DFPGA) or even mutual exclusive lifetimes (e.g. two objects are to reside in the same area on the DFPGA). To avoid such conflicts some rules are required which have to be obeyed.

R1 There are no overlapping configuration processes of two objects allowed even if their locations on the DFPGA differ. This is required due to todays DFPGAs having just one configuration port allowing one configuration at a time.

R2 There is no access to an reconfigurable object until its configuration process is completed.

R3 All accesses to a specific ReconObjects content have to finish before the ReconObject is being reconfigured with different content.

R4 Assignments to ReconObjects are only allowed if the new object's type is derived from the base type the ReconObject symbol was declared with.

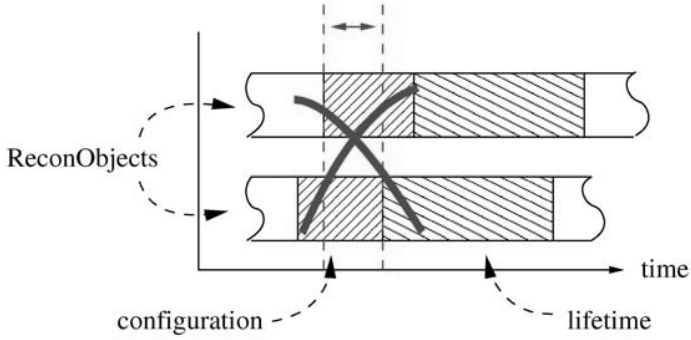


Figure 11.5. No overlapping configurations (R1)

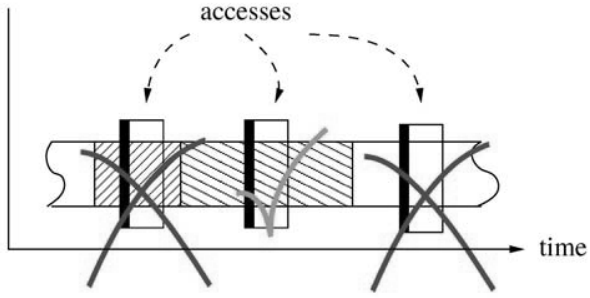


Figure 11.6. No accesses outside life (R2, R3)

There maybe attribute save- or restore phases around an objects presence on the DFPGA which are considered as being part of the reconfiguration for these rules. Those two phases are expained in Section 4 in detail. Rules R2 and R3 restrict accesses to an object to its lifetime.

ReconfigurationController

Rules R1 to R4 require the existence of a supervising instance which avoides violations at runtime. This is done using a reconfiguration controller. It has to keep track of which objects exist at a time and possibly delay accesses to them as long as the desired object is not functional. Furthermore its task is to decide which reconfiguration is required next and initiate those reconfiguration

processes. A reconfiguration controller is instantiated explicitly in the code, e.g. in the sc_main part:

```
ReconfigurationController< RoundRobin > controller_one;
```

The template parameter (in this case RoundRobin) defines the scheduling policy for concurrent reconfiguration requests. This mechanism is described in detail in [Grimpe and Oppenheimer, 2001] [Grimpe et al., 2002] [Radetzki, 2000]. Since there may be multiple reconfiguration controllers and reconfigurable objects in the design a binding mechanism has to be provided:

```
CONTROLLED_BY(crypto_module.crypto_object, controller_one);
```

This puts the reconfigurable object crypto_object which is a member of crypto_module under control of controller_one.

Timing

The duration of reconfiguration processes is mainly dependent on the size of the pattern stream which itself somehow correlates to the area being occupied by the object for which the stream is to be written. This area is unknown before synthesis but since it affects the systems timing we need to model reconfiguration times in the simulation:

```
DECLARE_TIME(controller_one, Blowfish,
             sc_time(26, SC_US), sc_time(200, SC_US))
```

This statement makes reconfiguration times for a Blowfish implementation known to the reconfiguration controller named controller_one. Each of the statements is describing reconfiguration times of one specific non-abstract class by two numbers. The first number is the attribute save-and-restore time and the second one the logic reconfiguration time. The latter one is the amount of time spent for reconfiguring the hardware resources in the specific area of the DFPGA.

The member attribute save-and-restore time is included here since there is a mechanism provided to prevent object attribute data from being lost during reconfiguration. This will be described in more detail in Section 4.

Up to now each reconfiguration controller represents one DFPGA. The controller has to be included in the DECLARE_TIME statement since the DFPGAs in the system may be of different types and therefore have different reconfiguration times for each class.

Recap

The language constructs introduced so far show how to to implicitly control the reconfiguration process. The content of the reconfigurable area is described

with an object and a reconfiguration controller takes care for all potential conflicts and schedules accesses and reconfigurations.

4. Advanced language constructs

Using the language constructs introduced so far only one context of a ReconObject can be alive at a certain point in program flow. The lifetimes of different contexts for a single ReconObject do not overlap. Accessing a ReconObject can be seen as accessing this context whatever object that may be at that time. The reconfiguration controller already disburdens the programmer of the need to explicitly care for the consistency criteria described in Section 3. However he still has to make sure the ReconObject always contains the right context for the functional correctness of the design. We now introduce a mechanism to automatically handle different contexts which may have overlapping lifetimes but still guarantee mutual exclusion regarding the occupation of the ReconObject.

Multiple identities

An identity object provides a special view to a ReconObject. There may be multiple identites for a single ReconObject and each of them represents a different context. Identities allow to separate the lifetime of a context from the time it is implemented in the assigned reconfigurable area on the DFPGA.

Accessing a ReconObject using an identity object means accessing one specific context. One may think of identites as named contexts which can be explicitly referenced. When initialized, the programmer may assume that the identity is accessible all the time. This is true from a functional point of view but wrong when referring to the timing. This aspect will be explained in more detail later.

Identites are used like a ReconObject which means that both method calls and assignments are written the same way as ReconObjects.

```
Identity<CryptoAlgorithm> blowfish_alg(crypto_object);
Identity<CryptoAlgorithm> des3_alg(crypto_object);
```

Using the keyword `Identity` two declarations are made which are similar to a ReconObject declaration except for the constructor argument which is the ReconObject for which this identity shall be used. The template parameter `CryptoAlgorithm` must be identical to that of the ReconObject which is provided as the constructors argument.

```
blowfish_alg = Blowfish();
des3_alg = DES3();
```

The first assignment to such an identity makes the identity valid and allows to access the newly created contents of the identity. It is not allowed to access an identity prior to this. In the example above both contents reside in the same ReconObject which was given at declaration time of the identity. Since the contents cannot be present at the DFPGA at the same time (and the overall goal is to describe reconfiguration abilities) they are swapped in and swapped out when necessary. For simulation the identity object provides an attribute backup storage which preserves the ReconObjects attribute values belonging to that identity while not being present in the simulated DFPGAs reconfigurable area. At synthesis step this could be implemented in swapping out this data to a SRAM or flash chip or a dedicated storage area on the DFPGA.

The two assignment operations shown above both create new contents in the ReconObject. When the assignment to `des3_alg` is made the previous member attributes of the ReconObject `crypto_object` are saved into the identity `blowfish_alg`.

Attribute handling

Each access to an identity causes the reconfiguration controller to make sure the ReconObject holds the desired contents, possibly saving objects' member attributes in a backup storage, reconfiguring the logic structure and restoring different member attributes which belong to the upcoming identity. This process of saving and restoring attributes will take some time in the final circuit and the simulation has to reflect this. In Section 11.6 the keyword `DECLARE_TIME` was introduced which provided one time value for logic reconfiguration and an additional one for attribute save-and-restore. This additional one is used once for saving the attributes and a second time for restoring them.

It may be desired that not all of an objects member attributes are being saved since excluding them from this save-and-restore process may reduce both area and runtime of the final implementation on the DFPGA. Therefore classes which are suitable to be placed in an ReconObject are required to provide one of the following specifiers as a member in the class definition:

```
DURABLE_RECONFIGURABLE(Blowfish);
TRANSIENT_RECONFIGURABLE(Blowfish, myClear);
```

The first one states that every attribute should be preserved during the save-and-restore processes while the second one allows excluding certain attributes. The second argument `myClear` contains a user-defined member method which resets attributes to constant values which they should contain after restore. There are no other statements allowed in this method than assignment of constants, in particular no function calls.

Recap

The main difference between accesses to a ReconObject in the direct way and accesses using an identity is that the former one works on any context there possibly is when the access is started while the latter one assures the presence of a certain context. A part of the system (e.g. a thread) may always use the same identity to access a certain context since that is swapped in on demand by the reconfiguration controller. From a functional perspective the thread does not see any interference caused by different identities. On the other hand a process which does not need or want to know which contexts are present in the ReconObject can still communicate or exchange data with processes using identities by utilizing the ReconObject directly. This leads to the following properties:

1 It is possible to access contexts belonging to an identity using the ReconObject directly.

2 It is not possible to access contexts belonging to an identity using a different identity.

5. Modeling and simulation experiment

The setup consists of one ReconObject with two possible contents: A Blowfish identity which has attributes to be saved and restored and a DES3 identity without such attributes. The Blowfish identity has to perform an `init` operation when being supplied with new keys to perform some preprocessing.

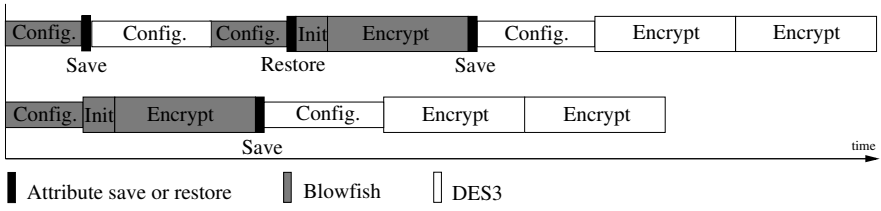


Figure 11.7. Two possible executions (different cases)

Two possible timelines are shown in Figure 11.7. The taller boxes represent times of real operation while the others (described with "configure") denote reconfiguration times. The black boxes show attribute save-and-restore times which are only required for the Blowfish identity. The reconfiguration and attribute handling times (provided by the `DECLARE_TIME` statement) are chosen manually. However, the length of the boxes in Figure 11.7 is derived from a simulation so their relative length to the normal operation (`Init` and `Encrypt`) is reflecting the simulation process.

The upper timeline in the diagram shows the crypto module initializing both identities with objects, starting with Blowfish. When Blowfish is operational its attributes have to be saved before the triple DES may be inserted. The first datastream to be processed is a Blowfish one so the third reconfiguration has to be performed which includes an attribute restore phase. To improve this startup overhead the lower timeline shows a simulation where the identites are constructed at their first use. Therefore two reconfiguration and one pair of attribute save/restore phases may be skipped.

The second datastream is a triple DES encrypted one. It requires a different identity to be accessible which supersedes the old Blowfish one. This causes a saving of member attributes and a different configuration pattern stream to be loaded onto the FPGA for this area (object and class switch). The third one is another triple DES stream which can be processed without reconfiguration.

The example was executed on a Linux PC using GNU gcc 3.3.2 along with the SystemC 2.0.1 simulation library. The simulation (2nd horizontal bar in Figure 11.7) lasted 75 seconds on an Athlon XP 1700+ and processed 1000 64-bit datawords on both Blowfish and DES3. The file sizes were:

Table 11.1. Source file sizes

Module	*Header size (bytes)*	*Implementation size (bytes)*
	Encryption algorithms	
cryptoalgorithm	487	786
blowfish	881	43672
des3	3991	64273
	Core module	
cryptomodule	645	7236
	Test environment, helper	
testbench	-	3950
examplenetworkpacket	2093	-
producerconsumermodule	688	2957
helper	-	2690

Where the CryptoModule was performing input, output, data packet type detection and processing steps using the cryptographic algorithms.

6. Conclusion

In this paper we presented a novel way of modeling and simulating reconfiguration processes in a high level hardware description language. It significantly reduces the efforts for the designer to model reconfigurable systems and disburdens him of several error-prone design tasks.

The approach provides the capability of easily modifying design decisions, e.g. binding from reconfigurable objects to DFPGAs supervised by a single controller or binding of an identity to a certain reconfigurable object. This allows fast design space exploration. Another advantage is the ability for the programmer to implement a reconfigurable system using well known concepts like polymorphism. All concepts presented here are designed to be completely synthesizable.

Future work

The only ressource limitating parallelism so far is the single configuration port of each DFPGA. But there may also be multiple DFPGAs competing for a single configuration pattern source or an attribute value storage (e.g. SRAM). Such further ressource handling will be investigated in.

Another topic is the introduction of a prefetch mechanism. The designer may then point at an identity which is required next. If possible, that identity is going to be preloaded using a background process.

Furthermore it is intended to extend the ODETTE synthesiser to cover this language extension in future, making it a synthesizable description.

Notes

1. The ODETTE synthesiser [Grimpe and Oppenheimer, 2001] [Grimpe et al., 2002] [Radetzki, 2000]
2. Formerly named global objects.
3. **C**omponent **L**ogic **B**lock [Lim and Peattie, 2002] [Xilinx, 2003]
4. **Recon**figurable **object**.

References

Agosta, Bruschi and Sciuto (2004). Synthesis of dynamic class loading specifications on reconfigurable hardware. *Proceedings of the Second IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)*.

Atmel Corporation (2002). *Datasheet AT40K*. `www.atmel.com`.

Bruschi, Francesco (2004). *Methodological Issues in the System Level Design of Embedded Systems*. PhD thesis, Politecnico di Milano, Politecnico die Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32, I 20133 Milano.

Daemen, Joan and Rijmen, Vincent (1999). AES proposal: Rijndael. Technical report. daemen.j@protonworld.com, vincent.rijmen@esat.kuleuven.ac.be.

DATA ENCRYPTION STANDARD (DES), (1993). Federal Information Processing Standards Publication, fips pub 46-2 edition. http://www.itl.nist.gov/fipspubs/index.htm.

Grimpe, Eike and Oppenheimer, Frank (2001). Aspects of Object-Oriented Hardware Modelling with SystemC-Plus. In *Forum on Design Languages FDL'01*.

Grimpe, Eike, Timmermann, Bernd, Fandrey, Tiemo, Biniasch, Ramon, and Oppenheimer, Frank (2002). SystemC Object-Oriented Extensions and Synthesis Features. In *Forum on Design Languages FDL'02*. Marseille.

Lim, David and Peattie, Mike (2002). *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. Xilinx, Inc. www.xilinx.com.

Noguera, Juanjo and Badia, Rosa M. (2002). HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 10, No. 4*, pages pp 399–415. IEEE.

Radetzki, Martin (2000). *Synthesis of Digital Circuits from Object-Oriented Specifications*. PhD thesis, Carl von Ossietzky Universität Oldenburg, http://odette.offis.de.

Schneier, Bruce (1993). Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Software Encryption, Cambridge Security Workshop Proceedings*. http://www.schneier.com/paper-blowfish-fse.html.

Seméria, Luc and Micheli, Giovanni De (1998). Encoding of Pointers for Hardware Synthesis. In *Proc. International Workshop on IP-based Synthesis and System Design IWLAS'98*, pages pp.57–63.

Seméria, Luc, Sato, Koichi, and Micheli, Giovanni De (2000). Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C. In *Proc. Design Automation and Test in Europe DATE'00*, pages pp.312–319.

SystemC 2.0.1 Language Reference Manual (2003). Open SystemC Initiative, 1177 Braham Lane 302, San Jose, CA 95118-3799, revision 1.0 edition. www.systemc.org.

Virtex-II Platform FPGAs: Complete Data Sheet (2003). Xilinx, Inc. www.xilinx.com.

Chapter 12

HETEROGENEOUS SYSTEM-LEVEL SPECIFICATION IN SYSTEMC

Fernando Herrera, Pablo Sánchez, Eugenio Villar
University of Cantabria
E.T.S.I. Industriales y de Telecomunicación
Avda. Los Castros s/n, 39005
Santander, Spain
fherrera,sanchez,villar@teisa.unican.es

Abstract A specification methodology for embedded system design should provide a capacity for heterogeneous specification. This would give the designer an effective tool to build a specification with different expressiveness needs, required by the multidisciplinary character of embedded systems, which, in turn, is due to their wide range of applications and an increasing integration capability. This specification methodology should be suitable for design tasks in order to improve design productivity. In this context, this paper deals with the general solution of the system-level heterogeneous specification in the framework of a specification methodology based on SystemC. This specification methodology is suitable for system-level modeling, but also for design procedures such as system-level profiling and single-source generation. Specifically, we study and propose a solution for a system-level SystemC specification which combines several untimed models of computations, (MoCs), namely CSP, PN and KPN. In order to situate clearly the heterogeneous specification methodology we will use a general study framework called Rugby metamodel.

1. Introduction

Embedded systems are becoming more common in our daily lives. This fact is reflected in the solid growth of the embedded market, (for instance, 49% for Embedded Micros, and 94% for Embedded Software Development Tools, RTOS and Services) [Forecast, 2004].

This trend is accompanied by a tendency towards an ever-increasing complexity in embedded components and systems. Components are more and more complex and powerful (for example, in 2001 already 67.5% of the processor

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 199–216.

architectures for embedded development were of 32 bits [Forecast, 2004]. In addition, one single chip is able to cope with the integration of more components, reaching the complete system (System-on-Chip, SoC) [Chang et al., 1999]. The increase of integration drives the evolution from the SoC to the NoC (Network-on-Chip) [Jantsch et al., 2003], where networks are the top connection between the main components (systems) inside the same chip. All these facts underline the importance of devising new design paradigms.

Another fact that is widely accepted is the variety of applications of embedded systems (telecom, home, automotive, medical, industrial, etc...), which are normally composed of parts that require design techniques from different fields. This fact, together with the increasing ability of integration leads to a global heterogeneity of embedded systems that will probably continue in the future.

Global heterogeneity means that heterogeneity is not restricted to the architecture of the system (composed of elements such as general purpose processors, DSPs, application specific HW modules,...), but also affects its functionality (reactive, data dominated, control dominated, etc...). For either architectural modeling or functional specification, depending on the characteristics of each component or part of the system, one or another Model of Computation (MoC) [Lee et al., 1998] may be more or less suitable. A MoC fixes the abstraction level and relationships between the different aspects of a system specification. As this complexity and heterogeneity grows, the choice of an adequate MoC becomes more important. An incorrect abstraction-level in one of the MoC features can lead to inefficiency, even unfeasibility, in modeling and design. Therefore, in order to allow the designer to use the most appropriate specification method for each domain, the design methodology should support heterogeneous specification.

Important advances have been made to cope with the growing complexity and heterogeneity. In order to handle complexity the abstraction level has been raised to what is called system-level. This rise can include one or more aspects (or, as we will see, domains of the Rugby metamodel) of the system description, which becomes a system-level specification, and lets us focus on the key aspects of a design or a model. Thus, system-level specification constitutes the initial and essential step of the design and it has a central role in the new integrated design framework [Allan et al., 2002]. New specification languages for system-level specification have been proposed. Among them, SystemC has been widely accepted by the user community as a system-level specification language [Grötker et al., 2002][Müller et al., 2003].

Heterogeneous specification is also an active research field. One of the most important contributions is represented by Ptolemy [Ptolemy], a Java-based modeling and simulation environment able to support heterogeneous specifications. Under the same Java framework, the specification can be composed

of different components, where each one can correspond to a different domain, each of them characterized by a different MoC. This work is based on the denotational framework established in [Lee et al., 1998]. This denotational framework is also called metamodel.

The Rugby metamodel [Jantsch, 2004] enables an objective and systematic analysis of different models of computation by fixing their coordinates in a rugby ball-shaped diagram (Figure 12.1 and 12.2). The Rugby metamodel considers two basic concepts in order to classify any language or specification method: domain and abstraction. Domain is a feature of the model that can be independently analyzed. The metamodel distinguishes four basic domains, namely Computation, Time, Communication and Data. The abstraction level determines the level of detail used in each domain. In the diagram, each line represents a domain. Each domain line goes from the highest abstraction (left) to the maximum specificity (right). A coordinate in each domain is fixed by the abstraction level used. A MoC coordinate is represented by the set of coordinates used in each domain. The Rugby metamodel does not include the hierarchy concept since this usually does not depend on the MoC considered.

In this paper, we address the heterogeneous specification in SystemC. The proposed technique is based on a specification methodology able to support different models of computation [Herrera, et al., 2003] and suitable for different automatic design steps, as SW implementation [Herrera, Posadas et al., 2003]. Firstly, Section 2 uses the Rugby Metamodel in order to systematically situate the design and the specification methodology that we are assuming and extending. The suitability of the whole methodology for a heterogeneous input in a design framework will be demonstrated. Section 3 reviews the two main existing approaches for MoC interfaces. Section 4 deals with the heterogeneous specification in our specification methodology. Firstly, in subsection 4.1, we explain the MoC interface framework for the specification methodology and introduce some new concepts for the analysis and design of those interfaces. Subsection 4.2 deals with the case of untimed MoC interfaces, that is, those involving the untimed MoCs studied in [Herrera, et al., 2003]. Section 5 summarizes the aims, main conclusions and contributions of our work.

2. Specification methodology in the Rugby metamodel

This section aims to clarify our methodology of system-level heterogeneous specification and its position in the design methodology from the point of view of the Rugby metamodel.

First of all, Figure 12.1 highlights the suitability of SystemC for heterogeneous specification, as well as for system-level specification. The shadowed region shows the flexibility of the current language release (2.x) in the different domains, since it allows different abstraction levels in the domains.

For example, by using Rugby terminology, SystemC 2.0 covers coordinates ((Alg-LB),(IPC-Top/PC),(Sym-LV/PDT),(Caus-CT/PCT)). This is really a coordinate range where slashes separate the SW/HW options at low abstraction level. For example, for the Computation domain, the range goes from the Algorithmic level (Alg), typical of C/C++ programming, to Logic Block in HW. The Rugby metamodel can also illustrate the current limitations. These are basically in not reaching the highest and lowest abstraction domain limits. For instance, SystemC does not enable the direct capture of a specification as abstract as only one set of relations and constraints in computation, interfaces, data types and time. It is expected that lower resolution levels will be available in Data and Time domains (if, for example, analog extensions are provided in future SystemC releases). Outside the scope of the Rugby metamodel, hierarchy is supported in SystemC through modules and ports. Forthcoming extensions (sc_export) will improve this. Summarizing, SystemC provides considerable flexibility for specification under different MoCs.

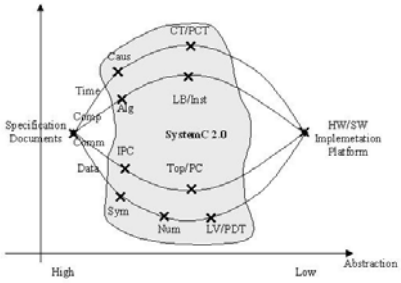


Figure 12.1. Flexibility of SystemC for heterogeneous specification.

The highest abstraction level in this specification methodology is placed under the (Symb, IMC, Alg, Caus) coordinates. It is represented as the shadowed region in Figure 12.2. It involves the highest abstraction level that can be reached in each domain with SystemC 2.x. This corresponds to untimed MoCs (i.e., PN (Process Networks), KPN (Kahn Process Networks) [Kahn, 1974], CSP (Communicating Sequential Processes) [Hoare, 1978] in [Herrera, et al., 2003]). In more detail, those coordinates correspond to Causality in the Time domain (time information can be abstracted to only partial order of events), Algorithmic in the Computation domain (process composed of sequential execution of statements), Inter-Process Communication in the Communication domain (through Interface Method calls, IMC) and Symbolic at Data domain (basically, supporting abstract data types).

The specification could incorporate parts with at least one domain at a lower abstraction level. For example, a part specified under a Synchronous MoC

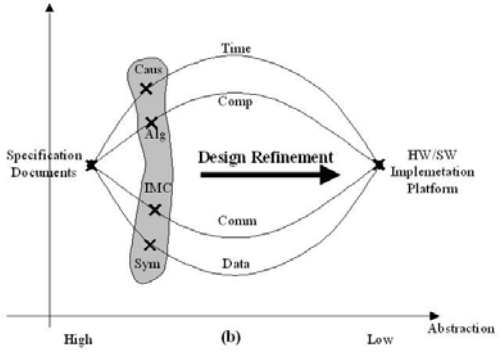


Figure 12.2. System-level specification in the design.

involves greater detail in the Time domain. This detail would be higher, not only in the Time domain, but also in the Data domain if certain parts contain HW descriptions.

Reaching the implementation involves one or more design steps (represented by the bold arrow), which through refinement in one or more domains (and more or less automatic), remove abstraction.

The MoC interface problem must be considered both at the specification (especially when we connect parts managing domains at different abstraction levels) and also at the design steps (which translate at least part of the system to lower abstraction levels).

3. General Resolution of MoC Interfaces

A fundamental problem in heterogeneous specification is the communication and synchronization of events among the system parts under different models of computation. This implies the development or use of a MoC interface. Taking into account the metamodel, this involves an adaptation (including or extracting information) in one or more domains. In some cases, this adaptation does not necessarily involve changing the abstraction level. Then, some other features (that differentiate one MoC from another) have to be considered. This will be seen in this paper when considering untimed PN, KPN and CSP MoCs. The adaptation has different complexities in each domain. The adaptation in the Time Domain can, in many cases, be considered more complex. However, in each domain, complexity depends on the decisions about which adaptation actions are finally performed (for example, at data domain, truncate or round, extend to 0 or interpolate) irrespective of whether the abstraction level is raised or lowered.

In [Jantsch, 2004], two main approaches for MoC interface solutions are shown:

- Interface definition for each MoC pair connection with separate frameworks (Figure 12.3a).
- Usage of a common framework for the communication of the parts specified under different MoCs (Figure 12.3b).

Before further discussion, we introduce two concepts: the specification and execution frameworks. The specification framework gives the specifier a set of elements with their own semantics and properties and rules for their interconnection. By using them, the specifier constructs a system specification, from which specific behavior and properties are expected to be obtained (system semantics). An execution framework sustains the specification framework and allows the structure and elements of the system specification to be executed without ambiguity respect to their semantics in order to perform a functional and/or performance verification of the specification without the need of the final platform. As can be seen, in this case we are basically referring to the simulation engines running over a general purpose OS (linux, unix, windows, etc,...), which runs in a host machine (PC, workstation). In this context, several types of simulation engines can be found, for execution (SystemC kernel), interpretation (i.e., a java runtime machine) or simulation (usual case of a VHDL simulator) of the specification. An execution framework for implementation provides the elements (technological libraries, embedded RTOS, etc,...) supporting the execution over the final platform. Notice that a physical distinction has been avoided. A simulation is actually running over a physical platform, the host machine. In a similar way, it is still usual to find physical platforms intended for development instead of final usage. They are also called target platform and are quite close to the final platform.

It is important for the correction of the design flow that the design steps maintain the semantics when passing from a simulation to an implementation execution framework. This is especially important when those steps become automatic (since the specifier loses control over the less abstract design representations).

With these concepts we will review these two MoC interface approaches. In the first MoC interface approach (Figure 12.3a), connection is more specific, explicit and localized. It usually deals with separate specification and execution frameworks. There are several examples. A typical one would be the traditional cosimulation of SW with HDL modules. Each part has quite different specification (i.e., HDL contrasting with a SW code style) and execution (discrete event HDL simulator versus instruction set simulator) frameworks. These differences are translated into different coordinates in Rugby domains

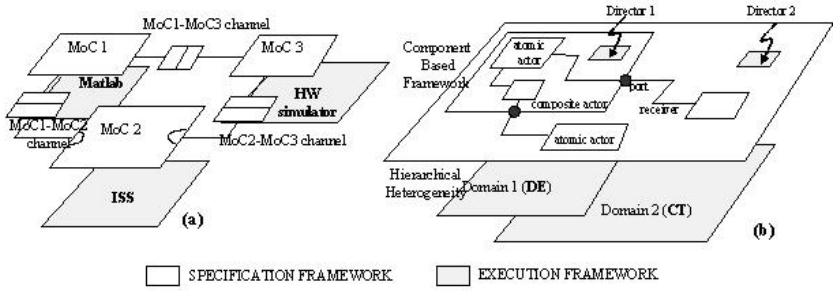


Figure 12.3. Two approaches for MoC link.

(i.e., bit types versus symbolic types in Data domain). The communication between them is usually performed by adaptation functions/processes that normally use a C-like interface provided by the HW simulator. These functions enclose and mask the necessary adaptations performed in the different Rugby domains.

The most representative example of the second approach is Ptolemy, which provides a common specification framework. It is a component-based specification framework where the specifier manages actors (components) with ports. Actors communicate among themselves through receivers (a common type of channel of the send-rev type). This has one important advantage: each time a new MoC is added, it only requires the implementation of the interface between the new MoC and the framework. Below and somehow hidden, several different (although managed with the same interface) execution frameworks are provided. In Ptolemy, those execution frameworks for simulation are called domains (note that here the meaning of 'domain' is different than in the Rugby metamodel) and their main element is the director class and its interaction with receivers. These domains are associated with a composite actor and control the execution of actor components (other atomic or composite actors). In this way, there can be a domain hierarchy matched with a component hierarchy. Because of this, Ptolemy is said to support hierarchical heterogeneity. The MoC interface task focuses on receivers and, as commented, the adaptation task avoids considering each MoC combination. A possible disadvantage for the management of this kind of "common channels" is that it could be too general for some MoC channels or too restrictive for others.

A difficult barrier that these approaches have to overcome (especially in the first case, where neither the specification nor the execution framework is common), is the separation of the different parts of the system in different frameworks, not only for writing MoC interfaces, but also for the rest of design tasks [Jantsch, 2004].

4. Heterogeneous Specification in SystemC

4.1 MoC Interfaces in the SystemC specification methodology

The main difference between previous methodologies (Figure 12.3) and the proposed methodology (Figure 12.4) is that the last one provides common specification and implementation frameworks.

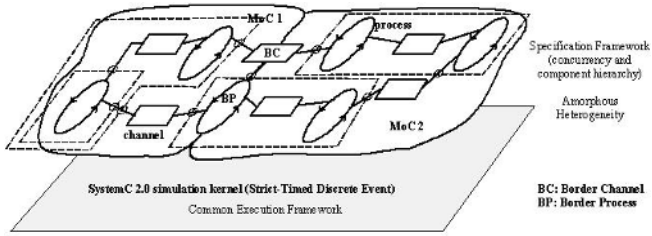


Figure 12.4. Specification and Execution frameworks in our SystemC methodology.

A system-level heterogeneous specification methodology based on SystemC provides the common specification framework. The homogeneity of the specification framework is due to the use of the same language for all the MoCs and the systematic use of processes and channels for specifying concurrent functionality. The heterogeneity is achieved through the definition of each MoC by the different set of channels and process styles. Therefore, the issue of MoC interfaces will basically deal with the nature of the processes and channels on the border of the parts specified under different MoCs. In this approach, suitable MoC interfaces will be provided for each MoC pair. Hierarchy at several levels is also supported. However, although possible, no match between components and MoC is necessary. We can find one module containing parts under two or more MoCs or a MoC distributed in several modules. Therefore, hierarchy will not be relevant (a flattening is considered) in terms of MoC location. This does not mean that hierarchy is completely discarded in the MoC study. The capacity for hierarchical composition from the MoC primitives will be considered when several MoCs are involved in the specification.

The common execution framework for simulation is based on a strict-timed discrete event simulation kernel. Due to the abstraction provided by the OO capabilities of C/C++, the underlying SystemC MoC can be hidden and other more abstract MoCs can be built on top [Grötker et al., 2002]. This enables the construction and execution of a SystemC specification combining several MoCs, and thus a heterogeneous specification. For instance, the same simulation can execute some processes where only causality relations are taken into account, while other processes are being triggered by a timed clock source.

The specifications constructed and executed in this way show amorphous heterogeneity (instead of hierarchical). That is, the SystemC simulation kernel, the topology of processes and channels and their blocking semantics control the execution, obliging a partial or a total order, or a time placement. Because of this, no match between component hierarchy and distribution of MoCs is needed.

As commented before, communication and synchronization of events is an important main problem to be solved in heterogeneous specification.

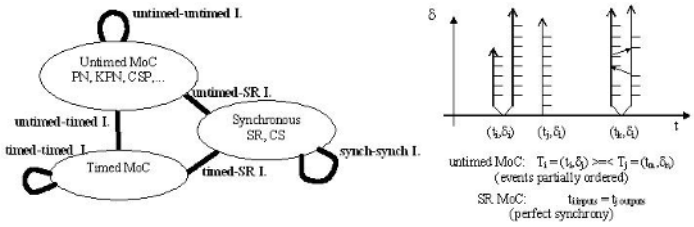


Figure 12.5. MoCs taking into account Time domain.

Figure 12.5 shows how the MoC interface problem is confronted in our methodology. It starts by considering a MoC classification that takes Time domain information into account [Jantsch, 2004]. Three basic MoC groups are distinguished, namely, untimed, synchronous and timed.

The classification can be interpreted in our SystemC methodology by means of the abstraction of the necessary time properties from the whole set of time information that the strict-timed discrete event simulation kernel maintains during the execution (basically coordinates (t_i, δ_i)). In [Herrera, et al., 2003], the PN, KPN and CSP untimed MoCs and the synchronous reactive (SR) MoC [Benveniste et al., 1991] (which assumes the perfect synchrony hypothesis) were treated.

The basic feature which defines untimed MoCs is that process events are related by a partial order (their time tags are partially ordered $Ti = (t_i, \delta_j) >=< Tj = (t_m, \delta_n)$). In our methodology, each δ_i represents an evaluation-update SystemC cycle. Therefore, an untimed specification could be executed at the same time (ti constant) over a δ-axis (a partial order imposed by a δ_i coordinate could be enough). Nevertheless, the order relationship between event time tags is the only information considered, regardless of whether the environment separates input stimuli in time (thus, having events in different ti coordinates) or not. In both cases, an untimed MoC is considered.

The SR MoC is based on the perfect synchrony hypothesis, which considers that outputs are synchronous with the inputs, that is, an instantaneous reaction. In our methodology, this is interpreted as input and output events having the

same t_i tag (t_i inputs $= t_j$ outputs). In order to achieve this, the SR MoC in SystemC assumes a certain separation between what is considered a generation process/event (usually, in the environment) or a reactive process/event (generally, part of the reactive system). It assumes that no generator process generates more than one generation event per channel at each t_i, therefore, the generation events are separated by an arbitrary $(t_j - t_i)$ time. Each of these events at t_i provokes the execution in a δ-axis with t_i constant of an SR evaluation and stabilization cycle of a reactive chain composed of reactive processes. A SystemC evaluation cycle is the first part of the evaluation-update cycle (called δ here) in the context of SystemC simulation. That is different from the SR evaluation cycle, which involves one or several δ. Each of these SR evaluation cycles is also called a slot. Thus, we have only one slot per t_i, arbitrarily separated in time, although the only important information is the total slot ordering (which can be represented by natural numbers, $\aleph$). The existence of concurrence in the generator processes provokes the existence of several reactive chains.

From this MoC classification we can derive a MoC interface taxonomy. This will enable a systematic study and generation of the interfaces. One group of interfaces do not involve time adaptation, that is, untimed-untimed, synchronous-synchronous and timed-timed MoCs Interfaces (in Figure 12.5, lines coming back to the same circle). For these, adaptations in other domains must be considered. Another group will consider interfaces that involve time adaptation. These are untimed-synchronous, untimed-timed and synchronous-timed interfaces (lines from one circle to another). In the next section, untimed MoC interfaces (highlighted line) will be presented. These represent the MoC connections with the highest abstraction level in our methodology. In this case, adaptation will not be necessary for Time domain, nor for the Computation and Data domains (Alg and Sym coordinates are maintained respectively). Therefore, the adaptation work will focus on communication. As shown in Section 2, the maximum abstraction level reached here is IPC (or what is called IMC, interface method call, in SystemC). In SystemC, processes/threads access channels, the elements that concentrate communication semantics, by means of access methods. Even a signal is conceptualized as a channel. Therefore, interface treatment for this case has to do with the different synchronization and data transfer semantics of the channels used in these MoCs, without affecting the process style. When interfaces with other less abstract models, such as SR, are handled, adaptations involve modifications even in the way the processes are written.

We shall now introduce two specific concepts useful for the treatment of MoC interfaces in this methodology. They originate from an effort to locate the MoCs and their interfaces in our SystemC heterogeneous specification framework. The MoC interfaces can appear in either of the two basic elements for concurrency and communication, process and channel. These concepts are:

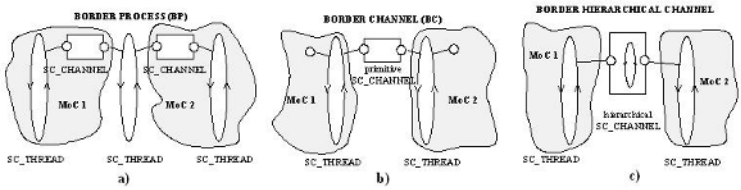


Figure 12.6. MoC links in SystemC by means of border channels and processes.

Border Process (BP) (Figure 12.6a) is a process accessing channels belonging to different MoCs. The border process is a highly flexible mechanism. By means of the same process, the specifier can mix two or more MoCs. The BP is just an extension in the use of the channels and specification style already presented in [Herrera, et al., 2003]. No more channels need to be considered and defined than those presented there. Because of this, the BP is suitable whenever only adaptations at the communication domain (channels) are needed (as has been seen, this was the case of untimed MoCs interfaces). It must be taken into account that if other domains should require adaptation, then this must be done explicitly in the border process. Then, for example, one part of the border process could have an algorithmic style where a rendezvous channel is accessed and another part a FSM style for implementing a protocol accessing signal channels. Taking into account that in our methodology, the specifier basically instantiates channels and writes processes, it could be interesting not to force the specifier to write this adaptation code, but to concentrate only on describing the concurrent functionality. In this case, the next concept may be more interesting.

Border Channel (BC)(Figure 12.6b) is a SystemC channel where the MoC interface is concentrated. The processes that this channel communicates belong to different MoCs (MoC 1 and MoC 2) with their own properties and characteristics. Adaptation mechanisms are hidden and performed inside channel implementation (and are thus implicit for the specifier in our methodology). The border channel presents two access interfaces Here we refer to SystemC interfaces, different from the general concept of MoC interface. We indifferently use the term interface and the meaning used should be clear from the context. One interface provides access methods to the MoC 1 and another to the MoC 2. From each MoC side, the channel is seen as if it were a channel proper of the MoC itself.

There are several reasons for its use. The first is that a BC saves the specifier the work of explicitly describing the adaptation by means of a BP. The BC has, in principle, its own semantics, different from other channels, which solves the MoC interface issue. Sometimes, they may require no more than a SystemC

interface addition. When the specifier includes this BC, he clearly separates MoCs, but doing this within the same language and framework (seamless). The BC also enables the direct connection of modules specified with different MoCs and different access interfaces (component-MoC match). This can be useful in the context of intellectual property (IP) exchange.

Up to now, for the sake of simplicity, it has been assumed that when referring to border channels we were talking about primitive channels. However, there is another derived possibility. This is the hierarchical border channel (Figure 12.6c). It is a hierarchical SystemC channel, which uses processes internally for the adaptation.

4.2 Untimed MoC Interfaces: CSP, PN and KPN MoC interfaces

Following the Rugby terminology, in this methodology, untimed MoC interfaces only require adaptation in the Communication domain. In the case of SystemC this is translated basically to the use of different channels accessed by a BP or a BC. In a SystemC channel, basically the management of synchronization events and transfer of data is considered. Since both sides handle the same kind of data, a Data domain adaptation is not necessary.

As seen in Figure 12.7, both kind of solutions, BP and BC are possible, without disturbing properties such as determinism in the connected MoCs. To this respect, the dynamic checking shown in [Herrera, et al., 2003] remains valid in the MoC sides, since it is performed at each channel instance, which only monitors which processes access the channel.

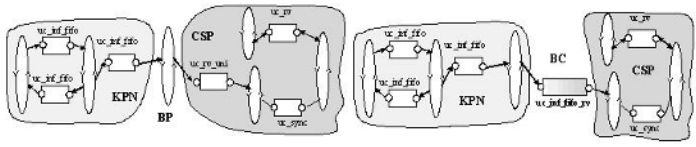


Figure 12.7. KPN-CSP equivalent specifications by means of border processes and channels.

As mentioned in the previous section, a solution of the BP type is very suitable for these kinds of MoCs, and rendezvous, and fifo channels can be easily mixed in the specification, without requiring the introduction of new elements.

The possibilities for BC have also been addressed. This gives a clear separation between the untimed MoCs. Each MoC part will have, apart from their own processes and channels, at least one process accessing the BC. The BC is asymmetrical, that is, it has a rendezvous access interface (for CSP side) and another for fifo access (for PN/KPN side). The semantic of each access

interface corresponds to the semantic content assumed for that type of access in the channel proper of that MoC. Therefore, the BC necessarily has an implementation intersecting the semantics of these different channels (fifo and rendezvous).

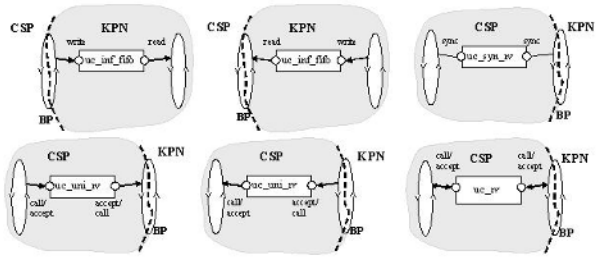


Figure 12.8. Possible accesses of the Border Process in the KPN-CSP MoC.

Before presenting the BC provided for those untimed MoC, some assumptions will be explained in order to restrict the problem. With respect to the Data domain in communication, generic data types will be transferred (so we really provide BC templates). As for synchronization semantics, we will deal only with the cases of channels with blocking accesses (except for the case of KPN), in general, guided by the maintenance of determinism properties. The basic questions to be solved in these channels have to do with blocking semantics (considering data availability and process arrivals), data transfer sense and the need for internal storing (infinite elements, limited, 1, 0,...).

Bearing this in mind, there are basically two options. The first is to write one general adapter channel (*uc_rv_fifo/uc_rv_inf_fifo* for the CSP/PN and CSP/KPN cases respectively) supporting the rendezvous and the fifo read/write interface. Its SystemC implementation requires either a dynamic checking or a static checking, by restricting the channel access through ports, in order to confirm the coherence of accesses in terms the sense of transference.

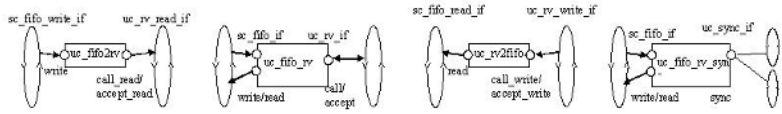


Figure 12.9. PN-CSP MoC interfaces with border channels.

In order to not to force always this check and let the specifier manage channels with easier semantics, another option has been considered. This consists in providing a set of four BC, one for each fifo/rendezvous interface combination

which keeps the coherence with the sense of data transfer through the channel. Therefore, there will be four versions for PN-CSP, *uc_fifo2rv*, *uc_rv2fifo*, *uc_rv_fifo* and *uc_rv_fifo_sync* and another four for KPN-CSP, *uc_inf_fifo2rv*, *uc_rv2inf_fifo*, *uc_rv_inf_fifo* and *uc_rv_inf_fifo_sync*. We will explain them simultaneously and considering the differences involved by the write unblocking access of infinite fifo.

uc_fifo2rv<T>, uc_inf_fifo2rv<T>: This is a unidirectional channel where the data go from KPN, through a fifo write interface, to the CSP part, which reads through a rendezvous read interface, *uc_rv_read_if*. This interface contains all the read methods of the unidirectional rendezvous provided in [Herrera, et al., 2003], *call_read* and *accept_read*. On the PN side (write access) the process will not block while it is possible to leave data tokens in the queue. Thus, blocking appears once the fifo is full in *uc_fifo2rv* and never in *uc_inf_fifo2rv*. In the read interface of the RV, the blocking condition requires an arrival in the PN/KPN to unblock, as would occur in the fifo read interface if we where talking about a *fifo/inf_fifo* channel. Therefore, *uc_fifo2rv<T>*/ *uc_inf_fifo2rv<T>* channel is basically a *fifo/inf_fifo* that offers a read interface of the rendezvous type. This means that its SystemC implementation basically consists in inheriting the fifo channel implementations and the read interface of rendezvous, making the read interface rendezvous methods call the fifo read interface methods. It must be noted that this implementation does not provide to the CSP side a strict-rendezvous behavior, since a writing on the PN/KPN side can leave access the channel without waiting for the arrival on the CSP side. The other choice would have been a pure rendezvous synchronization forcing in this case the KPN side to see a 1-size fifo and preventing the KPN. This occurs because a total compatibility in the intersection of the fifo and rendezvous synchronization semantics cannot be found. In cases like these, a decision must be taken and our criteria was to unblock as soon as possible while preserving determinism conditions.

Another question is to consider when the CSP access is *call_read* or *accept_read*. This affects the dynamic analysis that checks whether one or more processes access the channel. In the *accept_read* case, only one can access. In the *call_read* access, a concurrent access can be allowed (explicit acceptance of this possible indeterminism source). This also involves the combination of several types of checks (*SEVERAL_CALLERS* and *SEVERAL_WRITERS*) in the same channel.

uc_rv2fifo<T>, uc_rv2inf_fifo<T>: These are unidirectional channels where data go from the CSP side through a rendezvous write interface, *uc_rv_write_if*, which includes the methods *call_write* and *accept_write*, towards the KPN side, which reads through a fifo read interface. Now, the intersection of the blocking

semantics for reading the fifo and writing the rendezvous are considered. The KPN will block if there is not at least one arrival on the CSP side. The CSP requires the arrival of a reading access in the KPN side in order to unblock. From this, we deduce that the maximum storing size will be 1 and the behavior is that of a unidirectional rendezvous. In this case we do not find any incompatibility and the BC will basically be a unidirectional rendezvous plus a fifo read interface which calls those methods for reading data from rendezvous channel. At this time, *SEVERAL_CALLERS* and *SEVERAL_READERS* checks must be implemented.

uc_fifo_rv<T1,T2>, uc_inf_fifo_rv<T1,T2>: In this channel a bidirectional data transfer is given. It is a more general channel and has its own semantics. On the PN/KPN side it offers a write/read interface. On the CSP side, it is accessed by means of a read/write rendezvous interface, *uc_rv_read_write_if*. This contains the methods call and accept with two parameters, covering the two transfer senses. The semantics implemented is as follows. From the PN/KPN side it performs as two fifo accesses. The read access requires at least (and at most) one arrival at the CSP side, so that "internal read fifo" needs a storing capacity of 1 item. The write access performs as a finite/infinite fifo, thus requiring an N or an infinite storing capacity (PN/KPN cases). On the CSP side, the rendezvous is consumed (unblocked) when both a write and read is performed in the PN/KPN. Here we should consider new situations. Firstly, while only one process can access rendezvous for preserving determinism, on the KPN side, now, one or two processes (one accessing as writer and another as reader) can access without provoking indeterminism. This means only that *SEVERAL_WRITERS*, *SEVERAL_READERS* (for the PN/KPN part) and *SEVERAL_CALLERS* dynamic checking variables must be included in the BC to preserve determinism. Secondly, new deadlock conditions appear depending on the fifo size, on the number of accessing processes on the PN/KPN side (1 or 2), and on the sequence of the accesses. Basically, on the PN side, for an N-size (including the case of N=infinite) a burst of M writings before no readings (N $\geq$ M) is always possible. A later burst of M reads will enable M rendezvous potential consumptions on the CSP side.

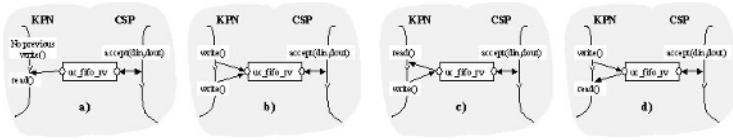


Figure 12.10. A KPN process access conditions in *uc_fifo_rv* border channel for deadlock study.

There is a new deadlock condition when considering one process accessing the (PN/KPN) part. For example, assuming N=1 in Figure 12.10, cases a) and b) would have deadlock, but not c) and d). If, for example, N=infinite, then case b) will not present deadlock. A new check (*CHECK_SINGLE_KPN_PROCESS_DEADLOCK*) is necessary for detecting this case.

uc_fifo_rv_sync<N>: This channel has a fifo read/write interface on the PN/KPN side and a synchronization interface (sync access method) on the CPS side. It ignores data arriving at the KPN interface, only the data arrival events being relevant. There is synchronization with both read and write methods on the KPN side and it allows any number of synchronizing processes, freely distributed on any of the sides, PN/KPN and CSP, although N will be the limit which guarantees determinism. Note that there is no infinite version since no storing capacity is required.

Regarding the connection of KPN-PN, it is clear that, functionally, it is possible to combine infinite and finite fifos in the specification. It is not necessary to consider a *uc_fifo_inf_fifo* BC, since this channel would not require interface adaptation.

As for the capacity for hierarchical composition, there is no more limitation than that of the correct use of interfaces of the module ports. Assuming modules M1 and M2, a match M1-KPN and M2-CSP is possible and modules can be connected by means of some of the BC shown. If no BC is used, a module M3 could include M2 and a BP. Then M3 would act as a wrapper and M1-M3 is seen as a KPN, where a fifo channel joins them. The future *sc_export* primitive will allow the M3 wrapper to include only a BC (rather than a BP) and the CSP M2 module.

5. Conclusions

Embedded system design is undergoing a methodological change. In this context, an important issue is the support of higher abstraction levels and heterogeneity in the specification methodology.

In this paper, the basis for establishing a methodology for system-level heterogeneous specification in SystemC has been shown. The Rugby metamodel enabled us to situate the specification methodology in the context of an integrated design methodology and generate some useful concepts, mainly for the solution of a basic problem in the heterogeneous specification; the MoC interface generation. This problem is attenuated in our methodology since it is based on a common specification and execution framework. A taxonomy of MoC interfaces, based on the required adaptation in the Rugby Time domain lets us identify the main groups of MoC interfaces to be solved. These are those no-requiring time adaptation (untimed-untimed, synchronous-synchronous, timed-timed) and those requiring it (untimed-timed, untimed-

synchronous, synchronous-timed). In the context of SystemC, the concepts of border process (BP) and border channel (BC) have been introduced.

Finally, applying all these concepts, the analysis and generation of untimed-untimed MoC interfaces for the untimed MoCs PN, KPN and CSP have been presented. These represent the highest abstraction level in our methodology. This level basically requires adaptation only at the Rugby Communication domain. The analysis of the possibilities with BP and the development of a set of BC demonstrates this relative simplicity.

This work opens some future lines. In the short term, these concepts can be applied to solve untimed-synchronous SR interfaces. In the longer term, interfaces with timed MoCs can be studied. In this way, the potential of the hierarchical border channel for communicating MoC with quite different time properties can be analyzed, especially if different simulation engines (for example, for analog extensions) are added to the simulation kernel of SystemC.

References

A. Allan, D. Edenfeld, W. Joyner, A. Kahng, M. Rodgers, Y. Zorian. "2001 Technology Roadmap for Semiconductors". IEEE Computer. January 2002.

A. Benveniste, and G. Berry. "The Synchronous Approach to Reactive and Real-Time Systems". Proceedings of the IEEE, V.79, N.9, September 1991.

H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd."Surviving the SoC Revolution: A guide to platform-based design". Kluwer; 1999.

T. Grötker, S. Liao, G. Martin, and S. Swan. "System Design with SystemC". Kluwer.2002.

F. Herrera, H. Posadas, P. Sánchez, E.Villar. "Systematic Embedded Software Generation from SystemC". Proc. of DATE, IEEE, 2003.

F. Herrera, P. Sánchez, E. Villar. "Modeling of CSP, KPN and SR Systems with SystemC". Proc. of FDL, ECSI, 2003.

C.A.R. Hoare. "Communicating Sequential Processes". Communications of the ACM, V.21, N.8, August 1978.

A. Jantsch. "Modeling Embedded Systems and SoC's". Morgan Kaufmann, 2004. ISBN:1-55860-925-3.

A. Jantsch, H.Tenhunen. "Networks on Chip". Kluwer Academic Publishers. 2003. ISPN:1-4020-7392-5.

G. Kahn. "The Semantics of a simple Language for Parallel Programming". Proc. of the IFIP Congress 74, North-Holland, 1974.

E. Lee, and A. Sangiovanni-Vincentelli. "A Framework for comparing Models of Computation". IEEE Trans. on CAD of ICs and Systems, V.17, N.12, December 1998.

W. Müller, W. Rosenstiel, and J. Ruf. "SystemC: Methodologies and Applications". Kluwer. 2003.

Ptolemy. http://ptolemy.eecs.berkeley.edu

Worldwide Shipments growth forecasts (2000-2004). Electronic Market Forecasters. Gartner Dataquest 2001 from www.embedded.com/advert/update.htm

Chapter 13

xHDL: EXTENDING VHDL TO IMPROVE CORE PARAMETERIZATION AND REUSE

Miguel A. Sánchez Marcos, Ángel Fernández Herrero, Marisa López-Vallejo
Dpto. Ingeniería Electrónica, E.T.S.I. Telecomunicación
Ciudad Universitaria s/n, 28040 Madrid, Spain
{masanchez,angelfh,marisa}@die.upm.es

Abstract Traditional hardware description languages are currently limited in their use to build complex systems through parameterization and reuse. In this chapter, we present xHDL, a meta-language designed to improve VHDL that provides flexible mechanisms for component customization, instantiation and interconnection. It has been conceived to ease the specification of highly parameterized cores and the reuse of already designed ones, keeping the currently available methodologies and synthesis tools. At the same time, it can help on parameter and component selection through the evaluation of functions that report on estimated characteristics of the design before the long synthesis phase. Finally, an FFT core illustrates the use of the meta-language for the specification of a complex design.

Keywords: HDL, IP, reuse, parameterization, component-based design

Introduction

Current VLSI designs are characterized by their increasing complexity and performance, while the design environment must ensure as short as possible development time. In this context, only the reuse of previous designs allows meeting such strong design constraints [Gajski, 1999]. Reused components, known as cores, may come from former designs or may be obtained from third parties. In the last case, they are qualified as Intellectual Property (IP), based on licensing reasons.

The design of systems can be significantly simplified by the proper assembly of already available components. However, integrating cores into a system is mostly manual, and consequently error prone. Designers must rightly know all

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 217–235.

the characteristics of complex cores, as they are functionality, interfaces, and basic parameters.

Hence, there is a clear need of tools that help during component assembly and core generation tasks, while providing quality assessment measures (area, speed, power, accuracy) to ease the selection of cores and their key parameters. These goals can only be accomplished if the cores are specified with a language that allows subcomponent customization, instantiation and interconnection, with the corresponding simplification of the design of hierarchical collections of modules.

In this sense, conventional HDLs, as VHDL or Verilog, do not satisfy all the requirements that IP reuse may have. For instance, these languages exhibit serious limitations to parameterize a design or do not allow the specification of additional attributes to drive a design space exploration process.

To overcome these limitations, in this work we present xHDL (*Extended HDL*), a meta-language that helps the designer in the difficult task of IP core description and reuse. Our goal when creating xHDL was not to develop a new HDL, but to improve some aspects of existing ones (currently VHDL) to easy both the parameterization of designs and the reuse of previous works. In this way, xHDL provides a simple and clear way to describe complex systems, while being easy to learn and use. Since a traditional language is used as base, not only previous designs can be reused, but also tools and methodologies.

The main advantages of the proposed meta-language are the following:

- It is conceived to emphasize reuse.
- It allows a high degree of parameterization of the cores.
- It simplifies specification due to the definition of new constructs.
- No conditions are imposed to the HDL used to describe the designs.
- It is easy to learn, since it stays close to conventional HDLs.
- Design space exploration is allowed based on feedback information.

The meta-language is accompanied by a compiler that generates VHDL source code. In this sense, the designer can write customizable code in whatever style is necessary, to simulate or synthesize, both architectures or testbenches, as the meta-language is effectively independent of the target technology considered for a final implementation.

The conception of xHDL allows its use in many applications, as the implementation of core-generation tools, automated design space exploration or generation of testbenches.

The chapter is organized as follows. Next, other related works will be reviewed. Section 2 provides the description of the basic constructs of xHDL,

while section 3 illustrates the use of the meta-language through an example. Finally, some applications are described and some conclusions are drawn.

1. Related work

Previous work on languages for IP reuse has been addressed from very different points of view, tackling many different aspects. In this section, we will contrast the key points of these approaches with the present work.

First, some languages conceived for other purposes have been previously used in IP environments. For instance, SystemC [Panda, 2001] can support modelling at the register-transfer, behavioral and system levels. However, these languages do not provide the parameterization facilities that xHDL exhibits. XML has also been used for IP-based design [Zhang et al., 2001], but this language was not devised to be directly used by a designer. It requires a long learning time when used to describe hardware, and is not the best way to express some particularities required for IP specification. Finally, other languages have also been used targeting reuse, as is the case with SpecC [Dömer and Gajski, 2000], which deals specially with the protection of IPs.

Important efforts have been carried out in the field of interface description and synthesis. Rowson et al. [Rowson and Sangiovanni-Vincentelli, 1997] introduced the concept of "interface-based design". The authors state that IPs should be designed in two parts, behavior and communication. Regular expressions are used to describe interface circuits, and an algorithm for their synthesis is presented in [Passerone et al., 1998]. Another interesting work on interface specification and verification is presented in [Suzuki et al., 1999]. Here, O_wL is an interface language defined for IP reuse, providing multiple applications. It is is devised to work at a very low level, what makes difficult its application for the specification of complex systems.

Confluence [Confluence, 2004] is a declarative, functional language that eases RTL code generation. This language combines the dataflow and component-based methodologies of HDLs with the expressiveness of modern functional programming. Some key ideas behind the language are shared by our proposal, but Confluence does not allow the definition of functions to evaluate internal parameters of the IP core and provide feedback about its suitability.

Finally, the component composition framework BALBOA [Doucet et al., 2003] is a bottom-up approach for SOC construction using reusable IPs. The framework includes a component integration language (*CIL*) which can be translated into a C++ implementation. In this case, as happened with Confluence, a new language is fully defined, making necessary to learn the new syntaxis and features and provide a complete and new set of tools to deal with.

Nowadays, there are workgroups upgrading traditional HDLs as Verilog or VHDL [EDA, 2004]. However, the time required to change a standard is too

long, and designers need tools to overcome language limitations even before the changes required by the language have been addressed.

In this chapter, we present a meta-language defined on top of VHDL to specify and generate IP cores. Our approach supports hierarchical implementation of complex designs, with emphasis on providing parameterization, module interconnection and reuse capabilities.

2. Fundamentals of xHDL

Source files for xHDL are known as *templates*. They contain the code for the available cores as embedded VHDL. The meta-language uses a reduced set of well defined types to symplify the learning process. This set is based in VHDL types, extended with new ones to implement specific functionalities. Arithmetic is performed by function calling, and a mechanism is provided to let the designer implement new functions and access them through the templates.

To allow reuse, xHDL implements a flexible way of communication between subcomponents, a calling mechanism. This helps in parameter fixing (e.g. a module can report to the core on the number of iterations needed for some computation), and also in getting feedback information from a subcomponent (e.g. multiplier latency that determines some buffer latency).

In the meta-language there are two different concepts for functions: as elements for expressions (introduced above) or as feedback information from a template. The last will be described later, and both can be identified in each context.

Similarly to VHDL, templates are divided into three sections: `generics`, `ports` and `architecture`.

2.1 Generics

This section allows the designer defining the basic data items for the core:

1 **Parameters**. They hold input values to customize the core. They can be provided directly, but also by an upper level calling core (if the current one is needed as a subcomponent), what allows passing parameters along the hierarchy. An example declaration with default value is:

```
parameter width = 8;
```

Differently from VHDL, where the generics are interpreted by the synthesizer, in xHDL the parameters will disappear after code generation, being substituted with their values. This greatly helps in a synthesis stage.

2 **Functions**. These are a set of output values to report on characteristics of the core, previously to the generation process. They can provide

bounds or recommended values for parameters, depending on the values chosen for some others, or they can also be simple estimates on core characteristics that help in the selection of parameter values:

```
function max_nstages = Add(width, 2);
```

As it happens to properties, their calculation is determined by the designer by using (expression) functions with parameters as arguments.

3 **Properties**. They are similar to functions, but calculated during the generation process, and reported afterwards. They can be used for estimators that strongly depend on how the core is created, or which subcomponents are finally selected for it, as throughput, timing, area, power or accuracy (expected quality magnitudes).

In the template, this type behaves as a global variable, so that, when declared, properties must be initialized with proper values:

```
property latency = 0;
```

In this example, the latency takes a null initial value and will be subsequently updated in the template, probably depending on specific generation options. The resulted value is stored to be recovered when needed.

2.2 Ports

The ports section is devoted to declare the interface for the core. In this sense, the statement:

```
XI: IN word;
```

is equivalent to the VHDL one:

```
XI: IN STD_LOGIC_VECTOR (word-1 DOWNTO 0);
```

where the type STD_LOGIC is considered as a base to interface definitions, being unnecessary its insertion in the declaration statements.

The meta-language introduces several facilities over VHDL, as is the possibility to conditionally declare ports:

```
( PIPELEVEL != 0 ) CLK: IN 1;
```

or the use of arrays, which are declared and accessed by using brackets:

```
DIR [Log2(word)]: IN 1;
```

2.3 Architecture

The meta-language aims to simplify the tasks that are difficult or limited in VHDL. For instance, even with the use of generate sentences, VHDL parameterization and generalization possibilities are awful and complex. In this sense, xHDL can be used to automate VHDL code insertions and subcomponent interconnection during core generation.

In xHDL, signals and components are declared only when needed, while variables allow carrying references to out-of-scope objects between different control structures, and can be used in embedded VHDL source code.

In this section, the architecture description takes place, and thus, it is where actual VHDL source code will be inserted. The constructions that can be used are within several types: *declarations*, *control structures* and *references*.

Declarations. In a template, it is possible to use data items declared in the `generics` or `ports` sections, but it is also possible to declare new ones for internal use. Differently from VHDL, declarations can be placed anywhere, but always before the object is used. *Variables*, *signals*, *entities* and *components* can be declared.

When a core reuses another one as subcomponent, it can use information provided by feedback functions and facilities for conditional declaration to declare only the types really needed at each moment.

Variables The variables allow storing values or references in a section:

```
variable var_word = word;
```

Variables are internally stored as strings, but their actual types depend on the use (integer, floating, etc.). They are similar to properties, as their values can be updated along the template:

```
latency = Add(latency, 1);
```

However, they are conceived as intermediate holders during component specification and are declared as needed in the `architecture` part, while properties are introduced in the `generics` section, as they keep important characteristics of the core that will be exposed to the user.

Variables in xHDL, which have no correspondence in VHDL, allow both store intermediate values in a template and keep references to ports and signals, which helps in the generalization of a core specification.

As mentioned before, it is possible to use functions, both tool predefined (`Add`, `Sub`, `Max`, `Log2`, `Dec2Bin`, etc.) and user defined. For calculations, constant values are also allowed.

Signals This type, together with entities, is closely related to the corresponding in VHDL. The keyword `signal` will insert a signal declaration into the declarative part of the VHDL component generated, keeping at the same time a reference to that signal into the meta-language:

```
signal mem2add[radix] var_word;
```

Variable and signal declarations have scope, the control structure where they are defined. Hence, out of there, references are lost. This simplifies name tracking during specification, differently from VHDL, where the signals must be declared at the beginning of the architecture. If needed, variables can be used to keep references to out-of-scope signals.

At the same time, signals are only generated if their control structure is accessed during component generation. Moreover, the same xHDL signal can produce several VHDL ones, as is the case with iterative sentences, simplifying declarations.

Entities and components Entity declarations are key in the reuse context. They allow refering to already described subcomponents.

To reuse a component, it is first necessary to provide its name and the library where it is stored. Then, we can initialize some of its parameters, differently from their default values, and get the value of feedback functions for the combination of parameters that results:

```
entity cordic lib.arith.Cordic;
cordic.generics (NVBITS = 16);
variable N = cordic->function ("NROTS_MAX");
```

The last two statements can be performed as many times as necessary, allowing an iterative process to search for the right values in each application (a process of inter-component communication for negotiation of parameter values). The component will not be generated. The strength of xHDL to perform design space exploration is partly due to this ability.

Then, it is possible to set again the same parameter or others, and finally declare the component, triggering its source code generation. This also inserts the declarative part of the subcomponent into the current one:

```
cordic.generics (NROTS = N);
component i_cordic = cordic->generate;
```

Now, we can access component properties defined by the designer:

```
variable L = i_cordic->property ("NREGS");
```

In summary, several entities can be evaluated within the meta-language to finally choose only the most convenient of them, or even several ones to be placed in different locations into the architecture.

Since our final goal is core reuse and automatic generation, in xHDL there is a way to nest templates for architectures, similar to function calling for expressions. It allows automatically generating code for the port-to-signal mapping for a subcomponent, ensuring at the same time the correct use of conditional ports:

```
i_cordic.ports ( CLK = CLK, ..., Y0 = rot_y0 );
var_temp_y = rot_y0;
```

The use of variables in the meta-language is very advantageous, giving a great flexibility to the generation process, as they are not declared in the finally generated component, what is different from the case of signals and subcomponents. Variables can transport signal references between control structures at the meta-language level.

Control Structures. In VHDL, the `if` construction does not allow `else`, being therefore necessary to reevaluate conditions. Moreover, the VHDL `for` has a closed range, which must be fixed from the beginning.

In this sense, conditional structures in xHDL are `if-else` based:

```
if ( ... ) { ... }
if ( ... ) { ... } else { ... }
if ( ... ) { ... } else if ( ... ) { ... }
```

On the other hand, iterative structures are based in `while` constructs, with condition evaluation previous to loop execution:

```
while ( ... ) { ... }
```

Conditions have been already introduced in port declarations, and can be simple (relational operators: `==`, `!=` , `>`, `>=`, `<`, `<=`) or composed (logical operators: `&&`, `||`). Finally, it is possible to use meta-language functions to evaluate complex conditions.

Code insertions. As was previously stated, xHDL is not a new language, but it is conceived to ease the creation of parameterizable designs based on other description languages. In this sense, control structures and data types available in xHDL have already been described, but to properly describe a component, it is also necessary to perform source code insertions in the middle of the meta-language. Those insertions can be customized by using data types, through the substitution of their actual values during generation.

Source code insertions are performed by using reserved words in the meta-language, one for each possible location. In this way, to insert code into the VHDL library declarations section, we will use:

```
lib { LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL; }
```

To insert code into the declarative part of VHDL architectures:

```
decl { CONSTANT MEM_[index] : STD_LOGIC_VECTOR ([var_msb] DOWNTO 0) :=
   "[Dec2Bin(lib.arith.cordic.ATRCoeff(index, word))]"; }
```

Finally, to insert code into the implementation part of VHDL architectures:

```
code { MEM_[index] WHEN DIR = "[Dec2Bin(index, word_index)]" ELSE ... }
```

In code insertions, VHDL is interpreted as plain text, but accepts substitutions anywhere, triggered with brackets around meta-language expressions, which use formerly defined types (e.g. parameters, functions, properties, variables). This allows getting extensive code customization.

In this sense, the `decl` example above illustrates the use of variable substitutions and function calling, both tool predefined and user defined.

3. Design Example

To check the usefulness of xHDL, showing its customization and interconnection capabilities, a set of cores has been implemented and integrated into a meta-lenguage library. This contains from simple cores, as adders, registers, etc., to more sophisticated ones, as general KCM's or CORDIC rotators.

The library can be used as the base for new cores, as it is with the selected example for this chapter, the FFT. This core has been developed both reusing the library and building new components.

3.1 FFT specification

The chosen implementation follows Cooley-Tukey's algorithm [Rémondeau, 1999], with an online pipelined and parallel architecture. This architecture results from a regular repetition of several non-identical stages, so it is good to illustrate hierarchical design and parameterizable reuse of the components needed in the stages.

The VHDL code embedded in the templates for this example has been optimized to target the Xilinx Virtex II architecture, as is the case with the feedback information provided in properties.

The main parameters of the FFT core are the number of stages and the radix, which determine the final length. Other basic parameters are wordlength for input samples and scaling, providing the last one the growing policy after butterfly calculations:

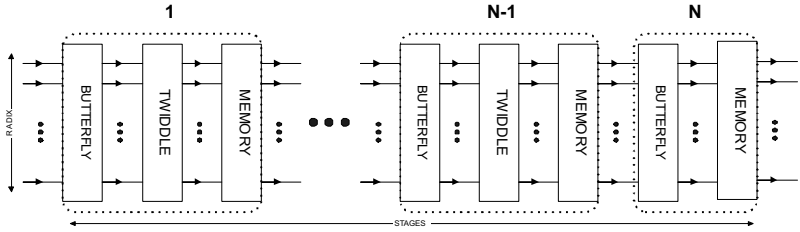


Figure 13.1. Structure of the FFT

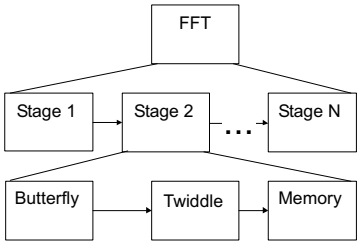


Figure 13.2. Hierarchy of the FFT

```
generics {
     parameter STAGES = 4;
     parameter RADIX = 4;
     ...
     }
```

These parameters are introduced in the top template of the design, which is based on the structural description of the selected algorithm (see figure 13.1) and built around three sub-cores: memory, butterfly and twiddle multipliers. These new cores are also implemented as xHDL templates with their own set of parameters, though related to those of the top template.

The top template merely specifies where and when the sub-cores are inserted, and how to customize (with xHDL functions and properties) and interconnect them (with xHDL variables and signals). Figure 13.2 shows the way these sub-cores are hierarchically instantiated into the specification.

The sub-cores are instantiated within a `while` control structure, which is used to unroll the algorithm into parameterized stages. In each one, first a butterfly is customized and inserted, then a twiddle multiplier and the memory core are generated, if needed:

```
variable fft_index = STAGES;
while ( fft_index > 0 ) {
    entity butterfly = fft.online.butterfly.dif;
    ...
    if ( fft_index > 1 ) {
       entity twiddle = fft.online.ffwd.twdarray;
```

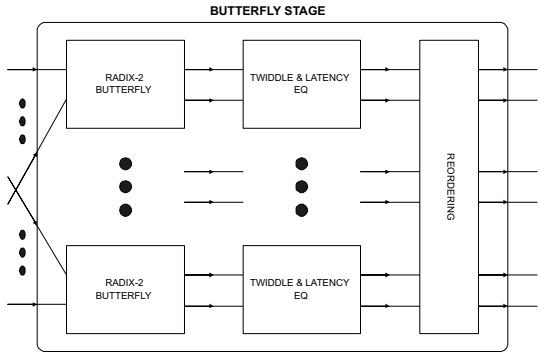


Figure 13.3. Components in a general butterfly element

```
   entity memory = fft.online.ffwd.datamem;
   ...
   }
...
fft_index = Dec(fft_index);
}
```

3.2 Butterfly

In this subcomponent, a key parameter is `RADIX`, which can only take power-of-two values. This parameter not only defines the inner structure but also provides the number of ports needed:

```
ports {
      ...
      IN_REAL  [RADIX] : in  WIDTH;
      IN_IMAG  [RADIX] : in  WIDTH;
      OUT_REAL [RADIX] : out OUT_WIDTH;
      OUT_IMAG [RADIX] : out OUT_WIDTH;
      }
```

The template is internally implemented as `Log2(RADIX)` stages, and hence, two variable arrays are declared to store the references to intermediate signals:

```
variable data_real [RADIX];
variable data_imag [RADIX];
```

The mapping of input ports into these variables is made as in:

```
variable index = 0;
while ( index < RADIX ) {
    data_real [index] = IN_REAL [index];
    data_imag [index] = IN_IMAG [index];
    index = Inc(index);
}
```

Now, they are used inside a main while loop, which creates the structure of the butterfly (shown in figure 13.3):

```
while ( index < RADIX ) {
    pipereg.ports (
          CLOCK = CLOCK,
          RESET = RESET,
          ENABLE = ENABLE,
          DATA_IN = data_real [index],
          DATA_OUT = real_pipereg [index] );
    data_real [index] = real_pipereg [index];
    index = Inc(index);
}
```

Finally, variables are again used to connect the output of the last component to the output ports:

```
while ( index < RADIX ) {
     code {
          [OUT_REAL [index]] <= [data_real[index]];
          [OUT_IMAG [index]] <= [data_imag[index]];
     }
     index = Inc(index);
}
```

In each stage of the main loop, radix-2 butterflies are firstly performed between pairs of ordered input samples. In these butterflies, the arrays are used as inputs, whereas local signals are declared for outputs. When the components are inserted, the variables are updated with these output signals. In this process, a bit is added if the scaling option is set, otherwise the result is truncated:

```
signal real_r2a [RADIX] : WIDTH;
   ...
// radix-2 butterflies
   ...
if ( SCALE != 0 ) {
   data_real [i] = real_r2a [i];
   ...
} else {
   data_real [i] = vhdl.Range(real_r2a[i], width, Dec(width), 1);
   ...
}
```

Next, a twiddle for each sample is calculated using xHDL functions, and the most suitable rotator for each one is chosen, together with the pipeline level for every radix-2 butterfly stage.

```
variable twiddle_term = fft.Twiddle(base_radix, index_radix);
if (( twiddle_term == "360.0" ) || ( twiddle_term == "0.0" )) {
   ...
} else if ( twiddle_term == "315.0" ) {
   ...
} else { // general twiddle rotator
   ...
}
```

Finally, the samples, which are stored in the array, are reordered by changing their positions:

```
while ( index < Div(RADIX, 2) ) {
   variable i1 = index;
   variable o1 = Mult(index, 2);
   ...
   tmp_real [o1] = data_real [i1];
   tmp_imag [o1] = data_imag [i1];
   index = Inc(index);
   }
```

During component generation, the template updates their own properties, as latency (which is used in the top FFT to synchronize all the sub-cores), with those of the subcomponents that built it (adders, subtracters, etc.).

3.3 Twiddle multipliers

This template acts a wrapper for the rotators which will be inserted in each FFT stage during source code generation. The choice of the rotator is based on the value of the angle associated with the twiddle position into the FFT. When the rotators are inserted, the template also has to equalize every output so that all of them have the same final wordlength, gain and latency.

For trivial rotations, a set of simplified sub-cores is available that implement them with great savings of resources with respect to a general rotator.

On the other hand, the general rotator chosen is based on CORDIC [Andraka, 1998]. This component is already available in the xHDL library, so that it only has to be instantiated with proper values for its parameters, some of them with recommended values obtained by using feedback information functions:

```
cordic.generics (KWIDTH = WIDTH, ...);
variable cordic_nrots = cordic->function ("NROTS_MAX");
cordic.generics (NROTS = cordic_nrots);
```

The CORDIC lets implement a twiddle rotator in two ways. First, as a completely general rotator with a register and an adder to internally generate the twiddle angle. Second, for fixed angles, the rotation sequences can be externally supplied from a ROM, which is a new sub-core whose coefficients are calculated by meta-language functions at generation time (figure 13.4).

Selection among the different rotator alternatives is determined by an input parameter to the core. This parameter can be fixed to one of the possible alternatives, or it can take the value `automatic`, in which case the template tries to minimize resources:

```
// TYPE 0 Automatic
// TYPE 1 Fixed-angle rotations
if (( TYPE == 1 ) || (( NUM_DATA < 512 ) && ( TYPE == 0 ))) {
   entity cordic = arith.cordic.online.fixrots;
   ...
} else {  // General rotator
   entity cordic = arith.cordic.online.basic;
   ...
}
```

```
decl {
   CONSTANT rom_[i] :
   STD_LOGIC_VECTOR ([Dec(WIDTH)] DOWNTO 0) :=
   [fft.CordicRots(i, POINTS, STAGE, WORDLENGTH)];
   }
```

a) source xHDL code for a ROM

```
CONSTANT rom_0 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "011110100"; -- "0.0"
CONSTANT rom_1 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100010001"; -- "-5.625"
CONSTANT rom_2 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100010111"; -- "-11.25"
CONSTANT rom_3 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100011101"; -- "-16.875"
CONSTANT rom_4 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100100100"; -- "-22.5"
CONSTANT rom_5 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100101011"; -- "-28.125"
CONSTANT rom_6 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100110001"; -- "-33.75"
CONSTANT rom_7 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100110111"; -- "-39.375"
CONSTANT rom_8 : STD_LOGIC_VECTOR (8 DOWNTO 0) := "100111110"; -- "-45.0"
...
```

b) generated VHDL code for a ROM

Figure 13.4. Example of code generation from xHDL

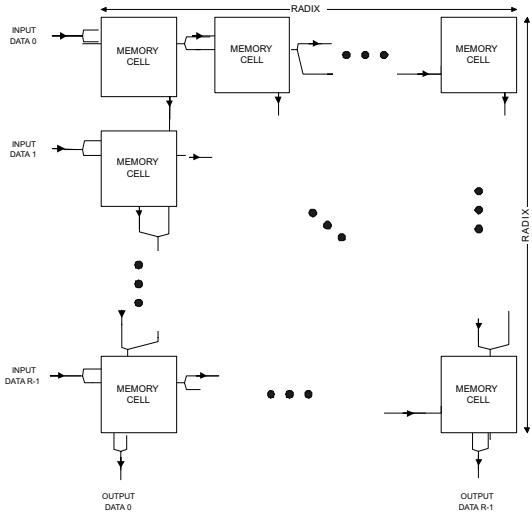


Figure 13.5. Structure of the memory template.

3.4 Memory

The function of this element is to keep and reorder intermediate samples between stages. Consequently, it needs a set of parameters that determine the amount of space which is necessary, and are passed from the upper template depending on their own parameters and variables.

The memory sub-core collects input samples from an FFT stage and sends ordered ones to another stage. To implement this function, the memory is arranged as $R \times R$ matrices, and the template describes this structure by using a double loop and a sub-template (memory cell):

```
variable index_in_radix = 0;
while ( index_in_radix < RADIX ) {
      ...
      variable index_out_radix = 0;
      while ( index_out_radix < RADIX ) {
              memory_cell.ports(...);
              ...
      }
      ...
}
```

The template for the memory cell provides the type of memory (block or distributed RAM, registers) through an internal parameter. The template for the top memory performs both cell interconnection and generation of the read/ write and addressing signals from the available set of input control signals.

4. Applications

The formerly described meta-language is mainly oriented to the specification and reuse of IP-cores, but it has many more applications in the IP domain:

- Design space exploration,
- Description of general IP-cores,
- Generation of testbenches,
- RTL code generation.

For instance, a *system designer* can make design space exploration from the very early stages of the development cycle, especially if it is based on the reuse of components from a library. If these components have well defined feedback functions and properties, they can be instantiated into a structural design and characteristics like estimated area, speed, etc. can be obtained even without a synthesizable design.

At the same time, design space exploration lets a *core designer* to select between different alternatives for subcomponents. Once the final components and their parameters have been chosen, the design can be fully specified with the meta-language to obtain a new parametric IP-core. Now, it can be added to the library to be reused in further designs.

On the other hand, during hardware design, it is also mandatory to take into consideration a set of testbenches to verify correct operation. Using xHDL, several parametric testbenches can be automatically added to the core and generated within the synthesizable source code.

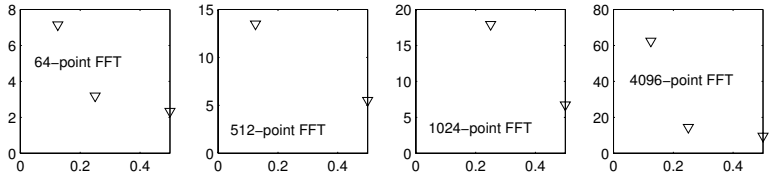


Figure 13.6. Design points of different FFT implementations (area vs. performance).

We address some considerations about the design space exploration and a core generation tool in the following sections.

4.1 Design Space Exploration

A design process is usually iterative, where different options need to be checked to finally get the best option. In this sense, the meta-language allows getting several core configurations by parameterization, while providing design feedback by functions and properties.

Based on these concepts, the designer can choose parameters within the bounds fixed by some of the functions, while checking the values from others to decide if the core meets restrictions. Finally, he can perform several tentative generations to check properties, which can give a more accurate or elaborated information, and then choose the best option. This is far cheaper than several complete synthesis stages.

Figure 13.6 illustrates the process of design space exploration for different FFT implementations. Four figures are shown for different N-point FFTs. The horizontal axis shows the inverse of the radix, which is proportional to the throughput for the same clock frequency, while the vertical one depicts area related values (thousands of LUTs for our FPGA implementation) for the resulted architecture.

The typical area-time tradeoff can be observed, and these results can be employed by the designer while choosing a core architecture.

4.2 Core Generation Tool

We have implemented an interactive tool to provide easy access to every possible configuration of a target core described with xHDL, and also to manage the available feedback functions [Fernández et al., 2004]. The tool collects the necessary information for the generation of a subcomponent from two main sources:

- Meta-language templates and some configuration files.
- Parameter values obtained from the user through the interface.

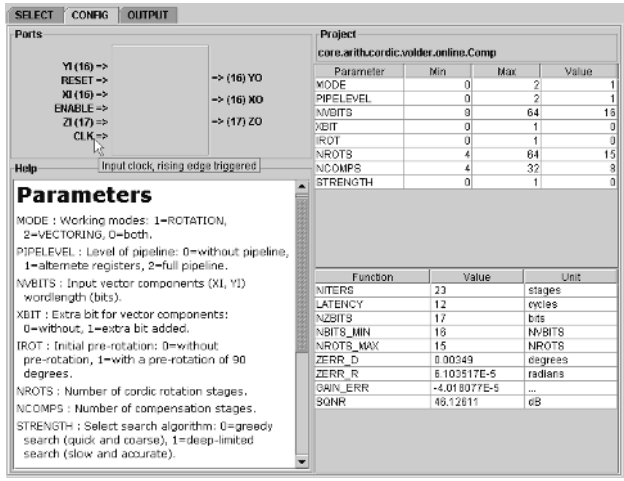


Figure 13.7. Second panel of the interface window for a CORDIC core.

The GUI is composed of a window with three panels, selectable by tabs placed in the upper part (figure 13.7). The first panel (*Select*) gives a list of the available cores by using a tree graph, while also shows some description information for the selected one.

The second panel (*Config*), shown in figure 13.7, consists of two parts. On the right, it displays the available component parameters, with bounds and a field to provide the desired value, and the feedback functions, with name, current value and unit. On the left, a block diagram for the core instance, together with HTML formatted help information. Both are automatically generated using the template contents.

The last panel (*Output*) includes fields to provide the final instance name and destination email address for the generated files, and buttons to begin core generation.

5. Conclusions

This chapter has dealt with the definition of xHDL, a meta-language for IP core description and reuse. This meta-language allows extensive VHDL source code parameterization and simplifies the specification phase, automating many awful tasks, as subcomponent instantiation and interconnection. It also provides several complex code manipulations, as conditionals and loops, many of them dependent on input parameters.

The underlying template concept is general enough to cope with guided source code generation of any hardware component whose implementation, parameters and feedback functions are available. In fact, the modular descrip-

tion allows the reuse of whatever component previously implemented as templates, and this is widely used to instantiate simpler components, as registers, multiplexers, arithmetic, or more complex sub-cores.

There are many interesting applications for xHDL. In this sense, a tool for core generation with parameter selection has been built as demonstrator. The meta-language can also be used to perform design space exploration, which is guided by the evaluation of feedback functions and properties that report on selected characteristics of the resulted components.

Acknowledgements

This work has been supported by the Spanish Government under Research Projects TIC2003-07036 and TIC2003-09061-C03-02.

References

Andraka, R. (1998). A survey of CORDIC algorithms for FPGA based computers. In *Proc. ACM/SIGMA 6th Int. Symposium on FPGAs*.

Confluence (2004). Confluence language. `http://www.confluent.org`.

Dömer, R. and Gajski, D. (2000). Reuse and Protection of Intellectual Property in the SpecC System. In *Proc. ASP-DAC*.

Doucet, F., Shukla, S., Otsuka, M., and Gupta, R. (2003). BALBOA: a Component-based Design Environment for System Models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*.

EDA (2004). Electronic Design Automation Industry Working Groups. `http://www.eda.org`.

Fernández, A., Sánchez, M. A., and López-Vallejo, M. (2004). A Web-based Environment for the Evaluation and Generation of Complex IP Cores. In *IP-SOC*.

Gajski, D. (1999). IP-based Design Methodology. In *Design Automation Conference (DAC)*.

Panda, P. R. (2001). SystemC – A Modeling Platform Supporting Multiple Design Abstractions. In *14th Intl. Symposium on System Synthesis*.

Passerone, R., Rowson, J. A., and Sangiovanni-Vincentelli, A. (1998). Automatic Synthesis of Interfaces between Incompatible Protocols. In *Design Automation Conference (DAC)*.

Rowson, J. A. and Sangiovanni-Vincentelli, A. (1997). Interface-based Design. In *Design Automation Conference (DAC)*.

Rémondeau, J.-M. (1999). Scalable parallel architecture for ultra fast FFT in an FPGA. In *Proc. ICSPAT*.

Suzuki, K., Ara, K., and Yano, K. (1999). O_wl: An interface description language for IP reuse. In *IEEE Conf. on Custom Integrated Circuits*.

Zhang, T., Benini, L., and Micheli, G. De (2001). Component selection and matching for IP-based design. In *Design Automation and Test in Europe (DATE)*.

Chapter 14

SYSTEMC MODELS FOR REALISTIC SIMULATIONS INVOLVING REAL-TIME OPERATING SYSTEM SERVICES

Prih Hastono, Stephan Klaus, and Sorin A. Huss
Integrated Circuits and Systems Laboratory
Technische Universität Darmstadt
Hochschulstr. 10 , 64289 Darmstadt
{hastono|klaus|huss}@iss.tu-darmstadt.de

Abstract The paradigm shift on embedded systems synthesis currently brings the design exploration towards higher levels of abstraction. Consequently, a need arises for an early and realistic assessment of system-level design decisions as well as its support from the design language used. Moreover, while execution properties of embedded software processes, which more and more dominate the functionality of embedded systems, can considerably vary, the chosen scheduling policy influences distinctly the execution properties. Unfortunately, the current version of SystemC is still lacks of that software modeling support. Therefore, the modeling capability of SystemC is being extended in this paper by generic real-time operating system services thus providing more realistic software modeling features. System and software design alternatives can thus be early explored and different scheduling policies can be easily validated.

Keywords: SystemC, Embedded Systems, Simulation, Real-Time Operating Systems

1. Introduction

The application of embedded systems in human life are rapidly spreading in many fields - from small and simple devices used in kitchens up to complex systems, which are part of highly dependable and safety-critical systems such as nuclear power plants and aircraft flight control systems. Due to the growing complexity of such systems, the design process considering software and hardware in concert (co-design) is constantly moved towards higher levels of abstraction. The proposed co-design flow starts from an abstract specification and

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 237–253.

results in a final implementation as illustrated in Figure 14.1. The system-level synthesis step constitutes first design decisions for the fundamental synthesis problems. Implementation alternatives should be validated as early as possible in the design flow. Therefore, a realistic system-level simulation support is mandatory for a successful design methodology. The tight time-to-market window imposes an automatic generation of these simulation models based on the specification and on first design decisions.

Especially safe-critical systems, but many other embedded systems too, belong to the class of real- time (RT) systems. As defined in literature, e.g., [Kopetz, 1997], an RT system is characterized by the fact that its overall correctness not only depends on the correctness of its functionality, but also on the timing of the response of the corresponding functionality. The RT behavior of software parts is managed by a real-time operating system (RTOS). The RTOS provides particular services required for composing the RT system that need to be considered during software design and analysis. The services provided by the operating system are mainly intended to support the implementation of the required multi-tasking or job concurrency by applying a chosen scheduling policy. Different policies can be explored for RT execution of embedded software including static event-driven, priority-based, and time-triggered scheduling. These features are of utmost interest, because the execution times of software tasks can vary considerably due to modern processor architectures, which heavily exploit pipelines and caches. Thus, it is obvious that some constructs or services are necessary, which are to be used for process creation or destruction and for communication and synchronization purposes. The model of RTOS simulation presented in this paper provides such generic services and allows to easily explore and validate both embedded software design alternatives and different task scheduling policies. SystemC 2.0 [SystemC, 2001], a system-level description language based on C++, was selected as the underlying modeling language and as implementation means for the executable computational models. The fundamental C++ class library of SystemC provides a cycle-based simulation kernel as well as all the necessary constructs required to create a cycle-accurate system model. Just one SystemC model is necessary to specify both the hardware and software parts of an embedded system. When a gradual refinement of the specification is performed, then an executable model will be available at the any point in time. However, SystemC must also be able to provide realistic assessment for develop-validate-and-test cycles of embedded software before any decisions with respect to processor and operating system are finalized, as well as before an executable platform model and prototype board becomes available. The SystemC community still intends to extend SystemC to software modeling features. Unfortunately, the current version of SystemC is still lacks support of those facilities. Therefore, the modeling capabilities of SystemC 2.0 is being extended in this paper by generic services of

RTOS providing constructs to model software decomposition, dynamic process creation and deletion, process control, preemption, static/dynamic process prioritization, static/dynamic scheduling, and inter-process/tasks communication and synchronization. Embedded software (architecture) design alternatives can thus be easily explored and validated at any point in time.

So far, the proposed approach considers both a stochastic timing model and the effects of different scheduling policies, which can easily be observed and validated from the execution of the associated SystemC models. The figures of merit of the proposed approach are demonstrated by means of the embedded information processing of a mobile robot.

2. Related Work

The basis of any appropriate design process is the specification model. Task graphs are a general and powerful specification concept for data-flow dominated embedded systems at system-level. This holds especially for concurrent software functions. Extensions to control-flow modeling of task graphs are introduced in [Eles et al., 1998, Klaus et al., 2003]. SystemC provides validation or simulation by the concept of executable specifications. A first approach for automatic generation of SystemC simulation models based on an abstract specification is presented in [Klaus et al., 2003].

Modeling of embedded systems at transaction level and a definition of this abstraction level can be found in [Yu et al., 2003]. Other concepts for modeling RTOS may be found in [Gerstlauer et al., 2003]. Proposals in [Guthier et al., 2001] are targeting application-specific operating systems. The main difference to the presented work is that we are working on system-level exploiting SystemC. Moreover, current version of SystemC is still lack of support of such a generic RTOS model and other facilities required to model software especially for real-time embedded systems. An idea to improve this situation was presented in [Moigne et al., 2004]. However, we are more focused on providing an environment for a highly flexible schedulability analysis by providing not only basic RTOS services, but also by implementing the model of scheduler for static and dynamic scheduling as well as processes control, preemption and dynamic priority assignment. Another important aspect for realistic simulations is the problem of run-time estimation of functional blocks. These difficulties arise from modern processors featuring pipelining and cache hierarchies. Behavioral intervals for the non-functional parameters are discussed in [Wolf, 2002]. In addition to pure intervals, [Manolache, 2002] considers a stochastic distribution of the execution times for schedulability analysis. In [Petters, 2002] the Gumbel distribution is introduced as a realistic metric to assess software execution times. In addition, different scheduling policies are explored for RT execution of the embedded software. Finally, Rate monotonic schedul-

ing (RMS) [Liu and Layland, 1973] algorithm, dynamic scheduling, such as earlier deadline first scheduling (EDF) [Liu and Layland, 1973], and static approaches are being covered by the proposed RTOS services.

3. Design Flow and System Modeling

The general design process starts from an abstract system specification. For this purpose a task graph based specification model is advocated. Task graphs capture the intended behavior by means of functional blocks and data dependencies between them. At system-level a task consists of complex functions, e.g., described by algorithms. Thus tasks represent the computational units that must be operated either sequentially or concurrently thus defining a number of computational jobs to be performed within an embedded system. This basic approach of using task graphs can be easily adopted to more complex specification models such as eTG [Klaus et al., 2003] or hCDM [Klaus and Huss, 2003], which capture to some extent control-flow information, too. After specifying the system with such an abstract model the initial assignment and optimization steps concerning the fundamental synthesis problems allocation, scheduling, and binding are to be performed next. Based on the results of these initial design decisions an executable model is highly desirable, because it allows an early assessment of the timed system behavior by simulation before implementing the core functionality. The system level synthesis pro-

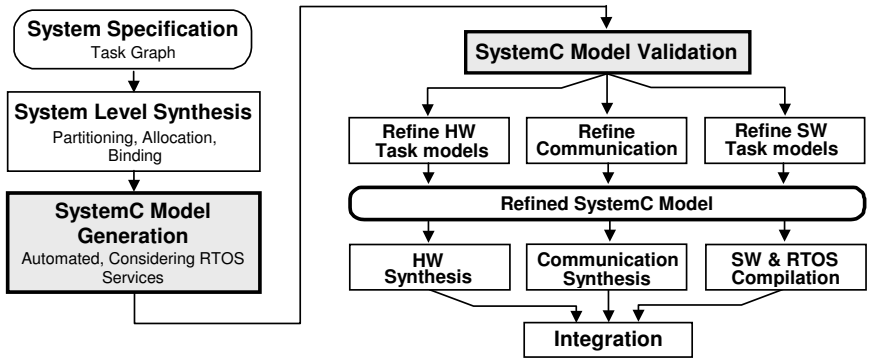


Figure 14.1. Proposed co-design flow for embedded systems

cess results in partitioning of system. Certain tasks may be implemented as hardware modules, whereas other tasks are best candidates for software implementations. Some already available hardware IP and third-party software code for re-use may inspirit this decision. The next step in the design flow is aimed to a translation of the result from system level synthesis processes into a corresponding SystemC model. In this model each task is assigned either to

hardware or software domain. Each processing element running software code is implemented as a distinct SystemC module. Such models can be automatically generated and afterwards refined by the support of the proposed RTOS services into fully compilable and synthesisable code.

The software tasks, which are allocated into some processing elements of the embedded system (ES) architecture, are modeled according to the functionality that is provided by the single SystemC module. The module (Main PE module in Figure 14.3) is an abstraction of a single processor, so all software tasks inside this module will be executed in a pseudo-parallel manner. As illustrated in Figure 14.2 due to system timing requirements and to specified quality of services, the tasks must be in general ordered in the time domain according to a certain scheduling algorithm. This means that an RTOS is required to provide such services for the software execution. Figure 14.3 illustrates, where the model of a generic RTOS is located within the SystemC model of a processing element. The example visualized in Figure 14.3 contains three software tasks P3, P4 and P5 surrounded by four hardware tasks. Figure 14.3 also illustrates that the RT operating system model along with its services, which support the inter-task communication refinement. The RT operating system is responsible to provide a viable means such that tasks can be synchronized in order to be able to communicate with each other. The synchronization between software tasks needs some specific constructs as detailed in Section 4. In case of communication requirements between a hardware task and a software task, the RTOS provides the driver library of the hardware accessed by software tasks. The operating system has in addition to offer some kind of bus driver services such that software tasks can communicate with other tasks that are running on another processing element. This service is mandatory for the support of distributed embedded systems, which involve several processing elements. In order to get more realistic models, variable task execution times exploiting a stochastic timing model are introduced. The Gumbel probability density function [Petters, 2002] is applied for this purpose in order to denote the distribution of the software execution times. The distribution trend of the timing behavior of a task within some timing interval is illustrated in Figure 14.4. Such an execution time distribution is represented by its mean value μ and statistical deviation σ. A problem arises in general since it is impossible to provide an absolute value for the worst-case execution time (WCET) for a task considering such a distribution. Even if the probability if very low, the WCET can be infinite in this model. However, the WCET can be approximated by a 10σ model at a failure rate of just 10^5, which may be viewed as sufficient for most practical applications. Based on this model of task execution timing the effects of different scheduling policies may be assessed by means of the introduced RTOS model. The implementation of scheduling policies in our generic RTOS model covers both static and dynamic scheduling policies: Static event-driven

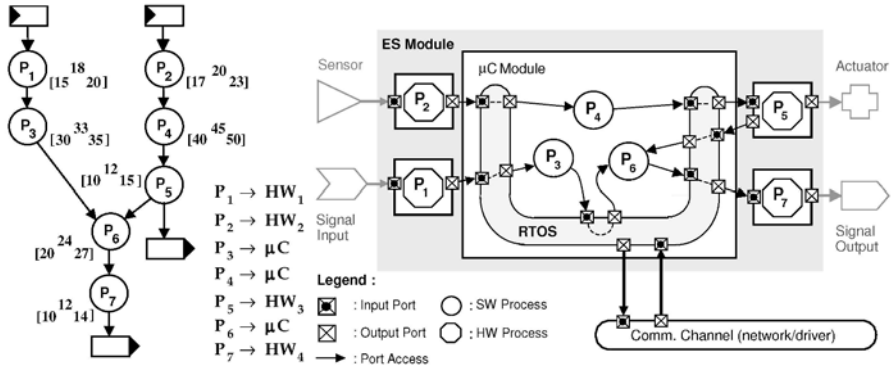


Figure 14.2. System specification as attributed task graph

Figure 14.3. Refinement of SystemC model and inter-task communication

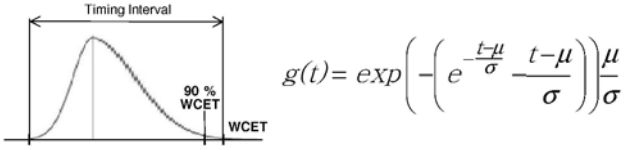


Figure 14.4. Task execution times model by Gumbel density function $g(t)$

scheduling, Static priority based scheduling, Static time-triggered scheduling, and Preemptive scheduling.

4. Extensions to SystemC

While Verilog and VHDL are being extended in order to improve their system-level capability, the community of SystemC intends to expand the usage of SystemC into software modeling. The usage models of SystemC in this sense outlined in [Grötker, 2002]:

(a) Design/Analyze system architecture

   i. HW/SW interface (partitioning, memory, bus access, etc.)
   ii. SW architecture (scheduling, preemption, priorities, communication, synchronization, etc.)

(b) Develop and validate/test SW well before

   i. Decision w.r.t. processor and RTOS have been finalized
   ii. Executable platform models (ISS, peripherals) are available

iii Prototype boards are available

However, [Grötker, 2002] also addresses a number of necessary SystemC software modeling capabilities that are not present in the current available version [SystemC, 2001] of SystemC such as: dynamic process creation, process control (suspend, resume, kill, etc.), scheduler modeling, and preemption.

The generic RTOS services presented in this paper provide these missing capabilities to overcome the weakness of SystemC for software modeling in the system design and analysis domains. Furthermore, the proposed RTOS model also provides the capabilities for scheduling assessment for both static (event-driven, time-triggered, priority based ordering) and dynamic scheduling (RMS, EDF). In order to support software modeling for the case of software coding towards a specific RTOS API, the advocated generic RTOS model also features a POSIX-like interface along with the process ID (PID) model.

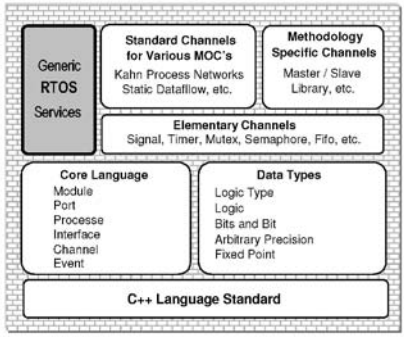


Figure 14.5. Enhancement of SystemC by generic RTOS

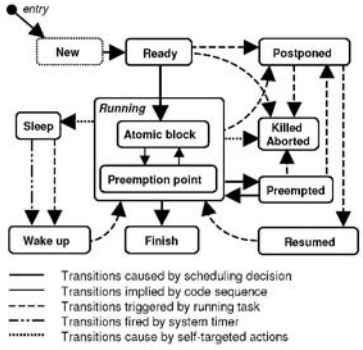


Figure 14.6. Model of task state transitions diagram managed by the RTOS model

4.1 Architecture of Generic Services

Many embedded applications for safety critical systems require the use of RT operating systems as an inherent part of their software architecture. For a realistic system-level simulation it is thus mandatory to capture the fundamental features of such an operating system. Therefore, SystemC is extended by the new library detailed in the sequel that provides basic scheduling and interprocess communication services. The resulting generic RT operating system is placed on top of SystemC 2.0 as illustrated in Figure 14.5. It provides most of the fundamental services commonly available in usual RT operating systems, especially those services, which are required for task creation, for task management and for inter-task synchronization. By means of these considerable

extensions to SystemC, the overall modeling capabilities become more suited to design space exploration as well as for early performance analysis of design decisions. The proposed model of the RTOS is implemented in the core SystemC as a module that encapsulates all threads instances and the scheduler, as well as both the simulation time and the scheduling policy. The operating system is thus responsible to provide associated services as illustrated in Figure 14.3. Because the software tasks have to be scheduled according to the selected schedule (time table, task order) or to a certain scheduling algorithm, additional services for this purpose are required. Models of task priority, preemption, and inter-task synchronization in the RTOS support efficiently the implementation of task scheduling. The scheduling service provides a way to validate task-scheduling decisions taken for created tasks. The algorithms of scheduling - both static and dynamic - implemented in this RTOS scheduler are based on a priority approach as detailed in Section 4.4.

4.2 Modeling Concurrency and Process Control

The notion of a task or process (used interchangeable in this paper) is the underlying concept to model concurrency in the abstract RTOS. It is implemented by means of the *sc_thread* macro in SystemC and encapsulates the thread as a task object. Thus, the RTOS model keeps the C++ object-oriented concept that is inherent in the basic language of SystemC. Task identification is represented by the task name and is mapped to a module name (*sc_module*) as an implication of SystemC constructs, where modules implementing the containers are the building blocks of the SystemC model architecture. The task name as well as the main code that embodies its functionality and the entry point of the task have to be passed to the operating system method when a task is being created. At the same time, some parameters such as task priority, ready time, arrival periodicity status, arrival period, and simulation time unit have to be passed as well to characterize the instance of task.

In order to support process control during the simulation run, some additional services are required in the RTOS model. These services embody the postpone/resume and the sleep/wake-up mechanisms. The allowed actions regarding synchronization are shown in table of Figure 14.6 with respect to the state of suspended tasks when the active task tries to perform a synchronization action by calling some RTOS services. Figure 14.6 also illustrates feasible task state transitions within the RT operating system model. All possible state transitions of the tasks are modeled, which exist in the runtime environment of the operating system. Only a currently running task is allowed to perform a sleep action. This action is invoked by calling the *sleep()* method of the RTOS object. This action will then shift the task to sleep state either until a specified time interval value passed as a method parameter is elapsed or until the task is

awaked by the currently running task. If the current task has a lower priority, then the task will be preempted by the awaked task unless some other tasks with higher priority values get ready to be scheduled.

```
ReAssPriRM
 for each active tasks τ_i do
    assign null priority into task τ_i
 end for
 T ← 0
 P ← smallest priority
 Repeat
    for each active tasks τ_i do
      if task τ_i has 0 priority τ_i do
        if T less than period of τ_i do
          T ← period of τ_i
        end if
      end if
    end for
    for each active tasks τ_i do
      if period of task τ_i equal to T do
          assigned priority P into tasks τ_i
      end if
    end for
    increment P // higher value for higher priority
 until no tasks has null priority value
```

Figure 14.7. Fixed priority re-assignment based on RM scheduling

```
ReAssDynPriEDF
 for each active tasks τ_i do
    assign null priority into task τ_i
 end for
 S ← 0 // remaining time
 P ← smallest priority
 for each active tasks τ_i do
    d_i ← D_i + (CPN_i − 1)× period of τ_i
 end for
 Repeat
    for each active tasks τ_i do
      if task τ_i has 0 priority τ_i do
        if S less than (d_i − t) do
          s ← (d_i − t)
        end if
      end if
    end for
    for each active tasks τ_i do
      if (d_i − t) eual to S do
        assigned priority P into tasks τ_i
      end if
    end for
    increment P // higher value for higher priority
 until no tasks has null priority value
```

Figure 14.8. Dynamic priority re-assignment based on EDF scheduling

Other synchronization actions such as *postpone()* and *kill()* can be called by the currently running task to postpone or to kill, respectively, either some other tasks or the currently active task itself. The *resume()* and *wakeup()* methods can be invoked by currently active tasks to resume and to wake-up postponed or sleeping tasks, respectively.

4.3 Model of Preemption and Granularity

The preemption model is constructed in conjunction with the timing model, i.e., by exploiting mainly the *await()* service provided by the RTOS model. We assume that it is possible to model atomic actions such that any functionality of a task can be constructed from these actions only. In other words, any function representing the functionality of a task is decomposable into a number of actions whereas their executions are atomic in the sense of run-

to-completion. Each atomic action code is assumed to have a fixed current execution time, which in turn is chosen from a given timing interval for this action. The timing behavior is implemented by passing the execution time value as a parameter when calling the *await()* method of the RTOS module. As a consequence of this conceptional model, preemption can only happen at the end of any *await()* statement, which is denoted as the preemption point that determines the granularity of preemption. The execution of each atomic action code provides the run-to-completion property and the associated piece of code is denoted as atomic block.

4.4 Scheduler and Task Timing Model

The proposed RTOS model implements the scheduler such that the scheduling algorithm can be simulated according to the selected scheduling policy. The scheduler is implemented as an internal thread that executes scheduler functionality embodied in its main function. The priority based scheduling for both static and dynamic operations are implemented by algorithms as shown in Figure 14.7 and 14.8, respectively. The timing model of task scheduling is illustrated in Figure 14.8, where the deadline of task i (d_i) is expressed by an absolute value (relative to the beginning of simulation). D_i denotes the absolute deadline at first period (first instance) at which task i is invocated for the first time. At this point in time the current period number (CPN) is equal to 1. The simulation time (current time stamp) is denoted by t.

4.5 Additional Services

Supporting software architecture modeling by means of SystemC toward a specific RTOS API, the proposed RTOS model also provides a POSIX like interface too such as handling a task pointer by means of the task identity as typically referred by process ID or PID in POSIX [IEEE, 1990]. The associated services have been implemented in the advocated OS model, they are highlighted Figure 14.10.

For this purpose the RTOS module contains some methods to access the pointers to task objects being managed by the operating system model. The method *get_pid()* requires either a task pointer or a task name, respectively, to be passed as input parameters, whereas the task PID will then be returned. The method *get_task()* acts as an inverse operation, i.e., it requires the task PID to be passed as input parameter and a pointer to the corresponding task will then be returned. By means this mechanism, an easy handling of all tasks based on their PIDs as commonly used in POSIX becomes possible. Thus, the design flow of embedded systems is fully supported even for such applications, where the exploitation of a POSIX-compatible RTOS is a mandatory part of the systems specification.

```
Scheduler_main()
 Repeat
    Call wait(0,SC_NS)
 until simulation is started
 if EMS is selected do
    Call ReAssPriRM
 end if
 Repeat
    evaluate tasks current time stamp
    evaluate tasks woke up at current time stamp
    evaluate tasks resumed at current time stamp
    if EDF is selected do
      Call ReAssDynPriEDF
    end if
    schedule active tasks according their pri.
    evaluate tasks killed at current time stamp
    evaluate tasks that turns to sleep
    evaluate tasks that finish
    evaluate tasks postponed
    call wait(sc_get_time_resolution())
 until simulation time is elapsed
```

Figure 14.9. Algorithm embodying the RTOS scheduler

```
class sc_rtos_posix_if:virtual public sc_interface{
  virtual rt_pid get_pid(sc_module_name )=0;
  virtual rt_pid get_pid(rt_task *)=0;
  virtual rt_task * get_task(rt_pid)=0;
  virtual sc_module_name get_name(rt_pid)=0;
  /* Task creation & termination services */
  virtual rt_pid create_task(
     sc_module_name name,
     void (__closure *mainfunc)(),
     rt_priority priority,
     rt_time first_ready_time,
     rt_periodicity periodicity,
     rt_time period,
     sc_time_unit time_unit)=0;
  virtual void kill(rt_pid)=0;
  virtual void abort(rt_pid)=0;
  /* Task synchronization services*/
  virtual void resume(rt_pid)=0;
  virtual void postpone(rt_pid)=0;
  virtual void wakeup(rt_pid)=0;
  }
```

Figure 14.10. POSIX-like services of the generic RTOS model

5. System-Level Simulation

The proposed concept of a generic RTOS model built on top of SystemC is mainly aimed to an assessment of the overall effects of various scheduling policies by means of executing the associated models of the embedded system. According to our experience, the results of this model execution, i.e., simulation of the generated SystemC models, differ extremely, especially when comparing system-level synthesis results to simulation results exploiting an RTOS. This is because real task execution times vary considerably due to specific properties of the architecture of the chosen processor elements, such as cache structures and data path pipelining.

The assessment of the overall system behavior resulting from different scheduling policies and from stochastic task execution timing is demonstrated for the task graph specification and HW/SW task allocation of Figure 14.2 by means of Gantt charts. The variety of RT responses is illustrated in Figure 14.11, where different scheduling policies are applied. An average case task execution time (ACET) is derived from high-level architectural decisions. Then, it is assumed that the completion of any previous task generated an event which, in turn, triggers the invocation of successive tasks. Static event-driven scheduling takes these events to compose the static schedule. In other words,

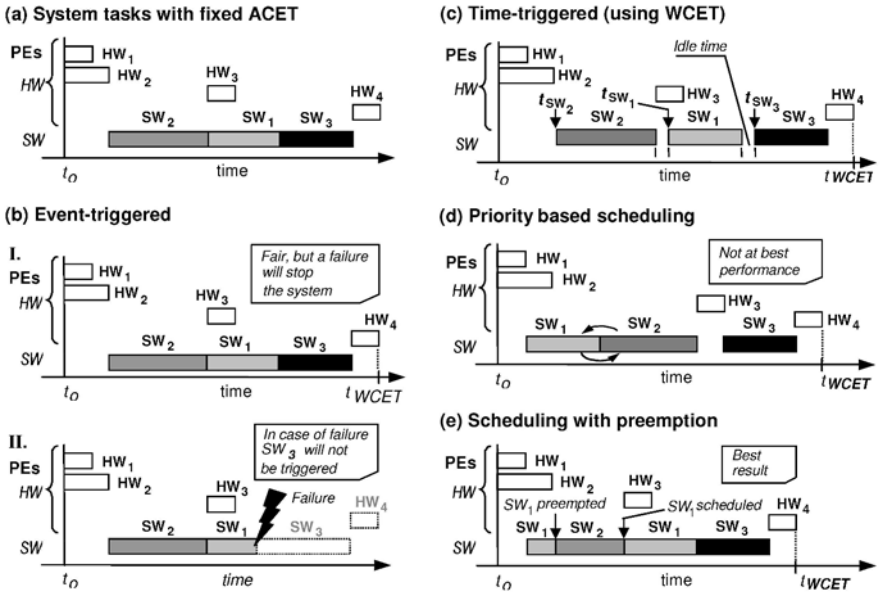


Figure 14.11. Assessment of different scheduling policies

there are no idle times between any two tasks in sequence. This means that a shorter total execution time may be yielded for the overall system behavior. However, as can be seen from Figure 14.11(b), the event-driven scheduling policy does not guarantee reliability, determinism, or a faster total execution time, respectively. The systems reliability is not guaranteed, because of the possibility of failures, i.e., in case that a task fails to complete on time, then this will skip the invocation of the successive task (Figure 14.11(b)II). Furthermore, the predictability of total execution time is questionable due to the unpredictability (statistical distribution) of the actual execution time of each task. Thus, the total execution time is not guaranteed to be shorter compared to other scheduling policies.

Time triggered scheduling as illustrated in Figure 14.11(c) shows a better performance in terms of predictability, i.e., guaranteed total system execution time, but the assumed WCET causes rather long idle times of the processor. This means that this policy is in general less efficient in terms of resource utilization. The potential disadvantages imposed by these static-scheduling policies can be overcome by dynamic scheduling. Priority-based scheduling as illustrated in Figure 14.11(d) is a typical policy in order to react dynamically to changed execution conditions, which may be caused by a slower or a faster execution of tasks as can be found in EDF scheduling. Another scheduling

policy, i.e., scheduling with preemption as illustrated in Figure 14.11(e), can considerably reduce idle processor time and speed up the overall execution because the idle time intervals can be utilized to partially execute tasks being in the ready state without destroying the optimal schedule order on the processing elements. The main advantage of scheduling with preemption is thus the possibility to apply scheduling algorithms as found in many dynamic scheduling techniques. RM scheduling, e.g., exploits both priority-based and scheduling with preemption policies. It is obvious even from the simple example given in Figure 14.2 that the results gained from high-level system synthesis given in Figure 14.11(a) differ considerably from more detailed simulation models featuring various scheduling policies. This is because these models take task execution timing intervals into account. This is the reason why simulation and a thorough validation of its results are mandatory at each refinement stage of the design process. Especially an introduction of preemption allows a realistic assessment of the overall system behavior and provides superior implementations in terms of both reliability and performance.

6. Application Example

Next, a considerably more complex application example - an autonomous robot equipped with ultrasound distance sensors, a camera, and with a wireless communication subsystem - is introduced for the purpose of a demonstration of the feasibility of the proposed methods to embedded systems design. The entire specification as depicted in Fig. 14.12 is composed from 25 tasks. The envisaged generic hardware architecture for information processing of this robot, i.e., the embedded system to be developed, consist of a micro controller located in the robot and the controlling host PC, which may be equipped by a co-processor on top of a PCI FPGA board. The communication between the mobile and the fixed parts of the embedded system is established by a wireless RS232 interface between micro controller and PC. Thus, the behavioural specification denoted as an attributed task graph in the left part of Figure 14.12 has to be mapped accordingly to these 3 hardware resources as indicated in the right part of the figure. The execution times for the different resources are assigned to each communication and computation task. Based on this specification the system-level synthesis determines implementation alternatives. These alternatives can then be validated by the automatic generated SystemC models of the distributed information processing of the robot, according to the proposed design approach. This forms the basis for both the evaluation of system- level design decisions and for an exploration of the effects of different scheduling policies.

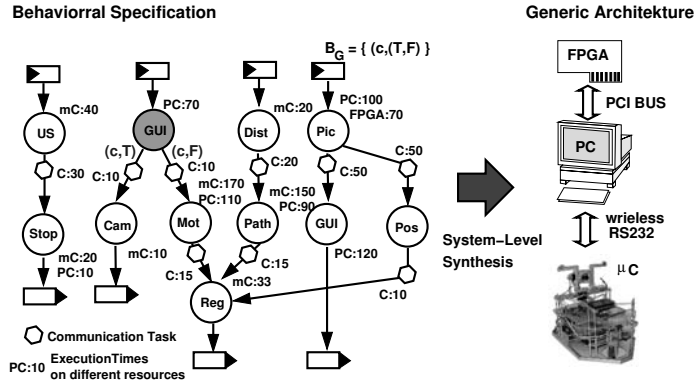


Figure 14.12. Behavioral specification and generic architecture of a mobile robot

7. Results

The evaluation results regarding task execution time for different scheduling policies compared to the system-level synthesis result are presented in Table 14.1. Based on the previously determined system level specification and design decisions the simulation models are automatically generated and then refined by exploring different scheduling policies. Due to stochastic execution times thousand different runs, i.e. experiments, were executed and evaluated. The best, average, and worst execution times are determined from this data. Some of resulted execution time distribution is presented in Figure 14.13.

Table 14.1. Scheduling results

	BCET	ACET	WCET
System-Level Synthesis	-	250ms	-
Time Triggert	540 ms	540 ms	540 ms
Event Driven	331 ms	357 ms	431 ms
Priority based ordering	335 ms	361 ms	435 ms
Tasks with preemtion	333 ms	370 ms	463 ms

The time-triggered execution policy leads to deterministic execution times without variation, but on the other hand to the worst overall system performance as visible in Table 14.1. When using the 90% timing limit (see Figure 14.4 instead of WCET, a speed-up of 10% can be reached, but due to the time-triggered scheduling still 1% system failures occur. Event-driven static scheduling is faster in terms of overall execution speed compared to the time-

triggered method, but the system behavior is less reliable and, in addition, the probability for a task to miss its deadline can be higher. However, in average, the system performance is still better. So far, it can be seen that the implementation variants differ from the synthesis result and from different scheduling policies.

Priority based scheduling, as illustrated in Figure 14.11(d) and discussed before, can provide a way for system execution time optimization. A higher priority can be given to tasks on which the total execution time is sensitive. Unfortunately, selecting priority based scheduling for the mobile robot system will not give better performance in our case and longer execution time may yield due to overhead of the scheduling algorithm.

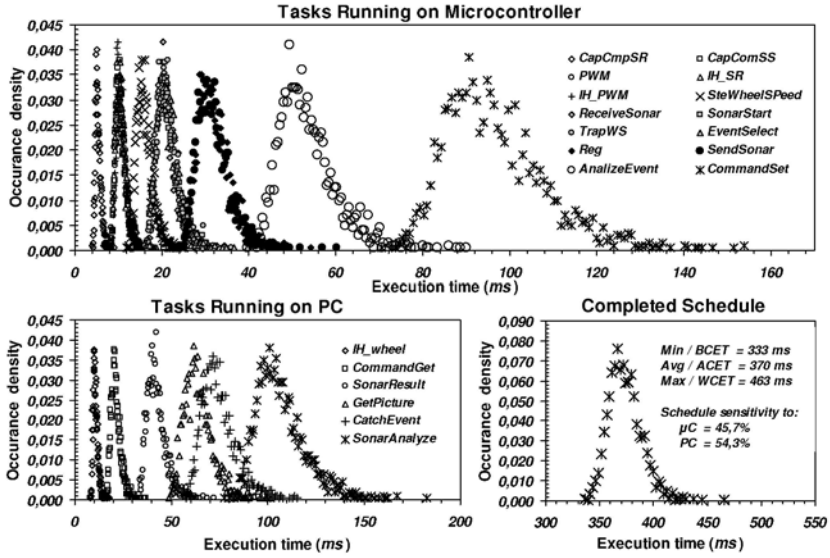


Figure 14.13. Performance results stemming from scheduling decisions

The preemptive scheduling method can yield better timing performance because idle times of some of the tasks can be used in a highly flexible manner. But for this example the dynamic scheduling doesnot lead to significantly better results. So, since the robot is not a highly safe-critical application, event driven scheduling is considered as the most feasible schedule strategy and resulting the best solution for implementation. In the mobile robot example, the figure of merit given by preemptive scheduling can not be exploited due to inter-task dependence affecting the feasibility of task scheduling. In this case, no tasks can be given higher right to run by preempting current running task

to proceed because they have lower precedence according to the dependence implied by task graph as system requirement.

8. Conclusion

A realistic simulation of system-level design decisions is mandatory for a successful design process, since the assumptions for the system-level synthesis execution semantics are very general and do not consider any scheduling policies. Therefore, the proposed approach extends SystemC by the capability of general RTOS modeling and allows an early and fast validation of different implementation alternatives due to an automatic generation of system-level simulation models considering the RTOS. The use of the stochastic timing model leads to realistic system simulations of overall execution properties. Thereby different scheduling policies can be tested and validated based on the general SystemC RTOS library.

Future work will adress the granularity of the basic blocks. Fine-grained blocks lead to a higher accuracy concerning the preemption point, but fine-grained blocks also increase simulation time required. Application of the RTOS models for distributed system is another subject for future work to extend SystemC RTOS servicees by providing support for both an assessment of inter-nodes task synchronization in a network and for global scheduling analysis.

References

Eles, P., Kuchcinski, K., Peng, Z., Doboli, A., and Pop, P. (1998). Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 132–139. IEEE Computer Society.

Gerstlauer, A., Yu, H., and Gajski, D. (2003). Rtos modeling for system level design. In *DATE-Conference*, pages 10130–10135.

Grötker, T. (2002). Modeling software with systemc 3.0. OSCI Language Working Group, Synopsys Inc, *In 6th European SystemC Users Group Presentations*.

Guthier, L., Yoo, S., and Jerraya, A. (2001). Automatic generation and targeting of application specific operating systems and embedded systems software. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 679–685. IEEE Press.

IEEE (1990). IEEE, New York.

Klaus, S. and Huss, S. A. (2003). A Novel Specification Model for IP-based Design. In *Proc. of EUROMICRO Symposium on Digital System Design*, pages 190–196, Belek, Turkey. IEEE Computer Society.

Klaus, S., Huss, S. A., and Trautman, T. (2003). Automatic Generation of Scheduled SystemC Models of Embedded Systems From Extended Task Graphs. In Villar, E. and Mermet, J. P., editors, *System Specification & Design Languages - Best of FDL'02*, pages 207–217. Kluwer Academic Publishers.

Kopetz, Hermann (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

Liu, C.L. and Layland, J.W. (1973). Scheduling algorithms for multiprograming in a hard rt environment. *Journal of the Association for Computing Machinery (ACM)*.

Manolache, S. (2002). *Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times*. PhD dissertation, Linköping University, Department of Computer and Information Science.

Moigne, R. Le, Pasquier, O., and Calvez, J-P. (2004). A generic rtos model for real-time systems simulation with systemc. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 30082. IEEE Computer Society.

Petters, S. M. (2002). How much worst case is needed in wcet estimation? In *2nd International Workshop on Worst Case Execution Time Analysis 2002*, Vienna, Austria.

SystemC (2001). Functional Specification For SystemC 2.0. `http://www.systemc.org`.

Wolf, F. (2002). *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers.

Yu, Haobo, Gerstlauer, Andreas, and Gajski, Daniel (2003). Rtos scheduling in transaction level models. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 31–36. ACM Press.

Chapter 15

SYSTEMC AND OCAPI-XL BASED SYSTEM-LEVEL DESIGN FOR RECONFIGURABLE SYSTEMS-ON-CHIP

Kari Tiensyrjä[1], Miroslav Cupak[2], Kostas Masselos[3], Marko Pettissalo[4], Konstantinos Potamianos[3], Yang Qu[1], Luc Rynders[2], Geert Vanmeerbeeck[2], Nikos Voros[3] and Yan Zhang[1]

[1]*VTT Electronics, P.O.Box 1100, FIN-90571 Oulu, Finland;* [2]*IMEC, Kapeldreef 75, B 3001 Leuven, Belgium;* [3]*INTRACOM SA, P.O. Box 68, GR-19002 Peania, Attika, Greece;* [4]*Nokia Technology Platforms, P.O.Box 50, FIN-90571 Oulu, Finland*

Abstract Reconfigurability is becoming an important part of System-on-Chip (SoC) design to cope with the increasing demands for simultaneous flexibility and computational power. Current hardware/software co-design methodologies provide little support for dealing with the additional design dimension introduced. Further support at the system-level is needed for the identification and modeling of dynamically re-configurable function blocks, for efficient design space exploration, partitioning and mapping, and for performance evaluation. The overhead effects, e.g. context switching and configuration data, should be included in the modeling already at the system-level in order to produce credible information for decision-making. This chapter focuses on hardware/software co-design applied for reconfigurable SoCs. We discuss exploration of additional requirements due to reconfigurability, report extensions to two C++ based languages/methodologies, SystemC and OCAPI-xl, to support those requirements, and present results of three case studies in the wireless and multimedia communication domain that were used for the validation of the approaches.

Keywords: co-design; communication; configuration overhead; context switching; design space exploration; dynamic reconfiguration; mapping; multimedia; OCAPI-xl; partitioning; reconfigurable; reconfigurability; SystemC; system-on-chip; wireless.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 255–269.

1. Introduction

Reconfigurable systems have raised a lot of research interest in recent years, and various reconfigurable architectures and technologies have been proposed [Compton et al., 2002]. Reconfigurable hardware combines the capability for post fabrication silicon reuse by different application tasks with computational efficiency due to the hardware-like spatial computation style. This fact allows the efficient adaptation to different operating conditions and/or different standards. The presence of reconfigurable resources on-chip also allows post shipment functionality modifications/upgrades and bug fixing capability similar to that of software.

Reconfigurability does not however come at no cost. The reconfiguration introduces extra delays as well as area and energy overheads. A further concern is technology portability, since embedded reconfigurable blocks are available only for a limited number of silicon processes. Additionally, current design methodologies and tools do not provide efficient support for dealing with this new design dimension at the system level.

A number of reconfigurable technologies are commercially available. Off-the-shelf Field Programmable Gate Arrays (FPGAs) available by companies such as Xilinx and Altera offer system-level densities of logic, memories and hardwired resources including processor cores [Virtex II Pro]. Due to their high unit costs FPGAs are not suitable for large volume consumer applications. Embedded FPGAs that can be integrated in customized SoCs are also commercially available [Varicore]. Coarse grain reconfigurable architectures include functional units of word level granularity such as PACT XPP technology [XPP]. However their commercial presence is limited compared to FPGAs mainly due to the difficulties in developing efficient mapping tools for such architectures.

The cost of deep submicron semiconductor technologies and the increasing design costs in state of the art semiconductor designs push for a move from conventional SoCs towards heterogeneous partly reconfigurable SoCs. Especially for signal processing type applications, coarse grain reconfigurable architectures are likely to dominate [Srikanteswara et al., 2003] since bit level granularity architectures (such as different FPGA flavors) offer high flexibility at the too high expense of power and area.

The rest of the chapter is organized as follows: The next section describes related research. Section 3 discusses specific requirements imposed by reconfigurability on the SoC design flow. Support extensions based on SystemC and OCAPI-xl languages and tools are described in Section 4. Section 5 summarizes three case studies in the domain of wireless communication that were used for experimenting and validating the approaches. Conclusions are drawn in Section 6.

2. Related Research

There have been several approaches researched towards developing reconfigurable architectures and associated software tools. However, they concentrated mainly to develop novel architectures, and their tools do not directly address the problem of modeling the reconfigurability at system-level. Such projects include e.g. Garp [Callahan et al., 2000], Xputer [Hartenstein, 2001], RAW [Taylor et al., 2002] and PipeRench [Goldstein et al., 2000].

Most existing high level design approaches for reconfigurable systems target compilation of C or C-like descriptions of targeted applications on reconfigurable architectures. In [Venkataramani et al., 2003] the compilation of a single assignment C-like language on the Morphosys coarse grained reconfigurable architecture is described. In [Cardoso et al., 2003] the compilation of C programs on the XPP reconfigurable platform is described. The approach also considers the temporal partitioning of the targeted behavioral description. Recently also a few co-compilation and co-synthesis type [Becker et al., 2003] design approaches for reconfigurability have been published.

The reconfigurable hardware brings a new dimension to system partitioning. The functional blocks of executable specification are partitioned into parts that will be implemented with software, hardware or dynamically reconfigurable blocks. The dynamic reconfiguration requires partitioning to address both temporal and spatial dimensions. Such an automatic partitioning is in a general case still an unsolved problem, but in specific cases solutions for temporal partitioning [Bobda, 2003], for task scheduling [Noguera et al., 2003] and for context management [Maestre et al., 2001] have been proposed.

3. Reconfigurability Requirements for Design Methodology

Reconfigurability manifests itself throughout the design flow. The System-Level Design (SLD) phase is the main focus of this paper and a more detailed diagram of it is shown in Fig. 15.1. The latter phases of the design flow are already more or less technology/vendor-specific and design flows and tools provided by vendors need to be used.

The SLD phase identifies reconfigurability needs, scenarios, constraints, and functionality in C or C++. It analyses and estimates the functional blocks with respect to reconfigurable implementation, decides on system partitioning and performs a system-level simulation to estimate the performance and resource impacts.

At the SLD phase the handling of reconfigurability requires an approach that addresses the three possible resource classes, i.e. software, fixed hardware and reconfigurable hardware. The SLD phase should support the following tasks:

- Unified description of system functionality in e.g. C

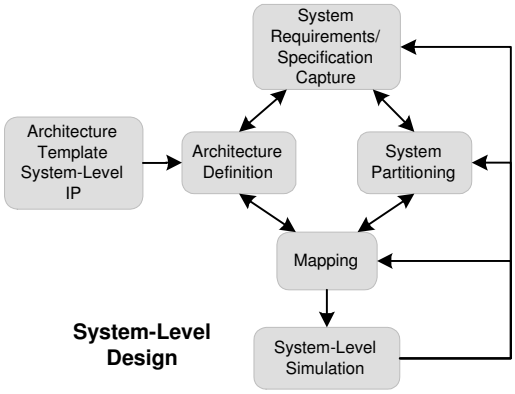


Figure 15.1. System-level design.

- Analysis and estimation to support justified partitioning decisions
- Reuse of architectures and Intellectual Property (IP) blocks
- Fast and efficient design space exploration
- Modeling of the effects of reconfiguration.

There are three major issues related to the reconfigurable technology that need to be modeled at the system level in order to get reliable information about the trade-offs between area, speed and total cost: the computational capacity requirements of functional blocks, the required resources needed for the largest reconfigurable context, and the delays and memory consumption caused by the reconfiguration.

The approach of this work is to take a holistic view to the overall SoC design flow in order to identify parts of the co-design methodology, where the inclusion of reconfigurability has the largest effects. It should be noted that reconfigurability does not appear as an isolated phenomenon, but as a tightly connected part of the overall SoC design flow. We do not constrain the system-level design to a specific architecture instance, but the designer defines the granularity by decomposing the functionality and explores reconfigurability through estimation, modeling, transformation and performance simulation. Reuse is supported through templates, and design space exploration allows alternatives to be studied in order to fine-tune the architecture, partitioning and mapping. The main properties of the explored reconfigurable technology alternatives are annotated to the system-level design. All the main decisions are already made at the exit of the system-level design phase.

4. Extending Systemc and OCAPI-xl for Support of Reconfigurability

SystemC is a standard modeling language based on C++, class libraries and a simulation kernel that provides the basic mechanisms for the system level modeling.

We have adopted SystemC language and tools [Grötker et al., 2002] as a base environment, on top of which we build our extensions for support of modeling, design space exploration and performance evaluation of reconfigurable parts at the system level.

4.1 SystemC Based Support

For designing of reconfigurable parts at system level, we developed: 1) an estimation method and tool for estimating execution time and resource consumption of function blocks on dynamically reconfigurable logic to support system partitioning, 2) a SystemC based modeling method and tool for reconfigurable parts to allow fast design space exploration through 3) system-level simulation using transaction-level models of the system.

Estimation Approach. The estimation approach applies basic principles of high-level synthesis, and is used for selecting candidate components that could benefit from implementation on a reconfigurable resource [Qu et al., 2003].

The starting point is the functional description given as a C-language algorithm. The designer decides the granularity of partitioning by decomposing the algorithm down to function blocks. A single function block may then be assigned to either software, reconfigurable logic or a fixed functional unit. Each of the function blocks will be individually studied and the set of estimation information will be fed into the system-level partitioning decision phase.

Together with the profiling information that is collected by running the executable algorithms with certain application data, the C codes of function blocks are transformed into a control data flow graph (CDFG) using the SUIF compiler of the Stanford University [SUIF]. For a selected instruction-set processor, the software (SW) estimator produces the estimated execution time. The hardware (HW) estimator produces the estimated HW execution time and the estimated HW resource utilization for each individual function block. The estimates can be used to narrow the design space in order to obtain a satisfactory solution with a few iterations.

Reconfigurability Modeling. The modeling method and associated tool transforms candidate components presented as SystemC modules to use a special SystemC template called Dynamically Reconfigurable Fabric (DRCF) as depicted in Fig. 15.2 [Pelkonen et al., 2003].

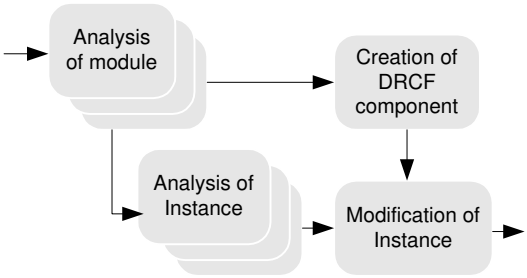


Figure 15.2. SystemC modeling method.

The template of the DRCF contains a configuration scheduler and an input splitter that routes data transfers to the correct instances. An example of a DRCF component is shown as part of a SoC model on the right hand side of Fig. 15.3. The configuration scheduler checks the target of each interface method call, forwards it if the target is active, or activates a context switch if the target is not active. This process automatically models context switching and the memory bus traffic. The automatic model transformer keeps the functionality of candidate components unchanged. The approach provides the means to test the effects of implementing some components in dynamically reconfigurable hardware.

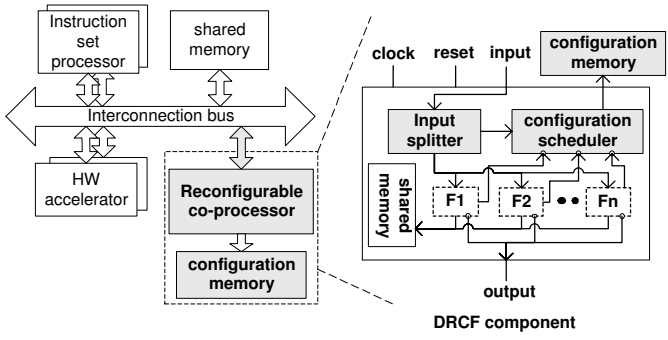


Figure 15.3. SoC model and DRCF template.

System-Level Simulation. As reconfigurability adds a new dimension to the design space, a method of analyzing performance of the resulting system in the early phase of design is needed. This can be achieved using SystemC transaction-level workload operation models. The timing information of com-

putation can be obtained from an estimation tool or from designers' experience. The factors related to the communication are architecture-dependent and can be set as parameters.

4.2 OCAPI-xl Based Support

To allow modeling of reconfigurability features at system level, we developed: 1) new software process type in OCAPI-xl [Ocapi-XL], 2) coupling of OCAPI-xl to SystemC for co-simulation, and 3) context switching from one resource towards another (software, reconfigurable hardware).

Software Processes Scheduling Extension. In the high-level software model of computation, concurrency is considered at the processor level. This means that for every process there is a separate processor assumed. Naturally, in real life this will typically not be the case. In realistic software implementation an operating system allows all the processes to be assigned to the same software processing resource. So from the performance point of view, the processes are not running concurrently, but they are sequentialized by the operating system scheduler onto the processing unit. To model such a behavior in the OCAPI-xl performance model, a separate process type, procManagedSW, has been introduced as depicted in Fig. 15.4.

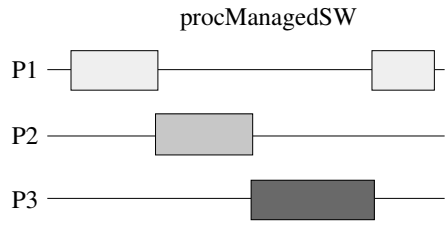


Figure 15.4. Sequentializing computation over time.

To be able to create a process of the type procManagedSW, the designer must first create a scheduling object. This scheduler will perform the actual sequentialization of all the processes that are attached to this object. The way this is done is defined in one of the member methods of this scheduling object. The user can define its own scheduling objects to model the behavior of the scheduler present in the target operating system.

It is important to realize that switching between the different SW tasks is not penalty-free. In order to come to the most accurate performance results, context-switching overhead is also considered in the performance model. The user can define extra context switching time for every process created, which is then applied to that process during the OCAPI-xl simulation.

SystemC Implementation of OCAPI-xl Threaded Process Extension. SystemC provides an implementation of a thread library bundled together with an event-driven simulation engine with notion of virtual time. Thus SystemC is used for implementation of the threaded-process extension with the additional bonus of automatically having an OCAPI-xl/SystemC co-simulation environment. Such an environment brings together the advantages of OCAPI-xl and SystemC. The essential idea of the common OCAPI-xl/SystemC environment is to let the whole OCAPI-xl part run in a single SystemC thread and to use SystemC synchronization mechanisms inside a modified OCAPI-xl simulation kernel to synchronize with the rest of the (SystemC) system. Since the OCAPI-xl kernel is single-threaded there are no thread compatibility problems created by this setup. The basic structure of the OCAPI-xl/SystemC co-simulation environment is shown in Fig. 15.5.

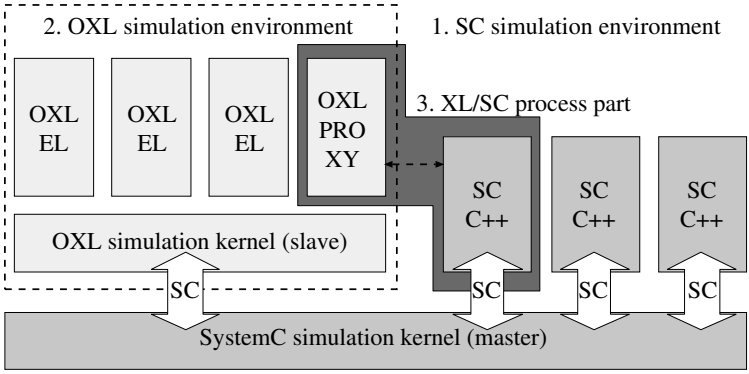


Figure 15.5. OCAPI-xl/SystemC co-simulation environment structure.

It consists of three domains within the SystemC environment:

1 The native SystemC processes controlled only by the SystemC simulation kernel

2 The OCAPI-xl domain running in a single SystemC thread and controlled by the OCAPI-xl kernel acting as a slave to the SystemC simulation kernel

3 OXL/SystemC processes running C++/SystemC code with access to OCAPI-xl's communication and synchronization primitives. These processes contain internally two parts: the SystemC part running the C++ or SystemC code, and a small OCAPI-xl proxy process, which is active when the process is executing an OCAPI-xl synchronization / communication primitive.

High-Level Modeling of Context Switching. The ability to reschedule a task either in hardware or software is an important asset in a reconfigurable system-on-chip. To support this feature, high-level implementation and management of hardware/software relocatable tasks in OCAPI-xl have been modeled. The aim is to model a pre-emptive relocation of tasks from the reconfigurable logic to the SW and vice versa. The model supports spatial temporal scheduling in hardware and software.

The OCAPI-xl code below illustrates the example of coding context switching for a task P1, switching between the different contexts (High-Level HW and ManagedSW), and simulating its behavior.

```
procDRCF P1("P1");
  //-- initial context: High-Level HW (default period of 10)
  P1.context(HLHW);
  //-- second context: SW under Round-Robin scheduler(RR)
  P1.context(ManagedSW, \&RR);
  //-- next context: High-Level HW with period of 2
  P1.context(HLHW, 2);
{
    //-- here goes "normal" OCAPI-xl task code

    //-- upon this operator the task will switch itself to the next context
    switchpoint();

    //-- here goes some more task code
}
  //-- and run the simulation for 2000 cycles
run(2000);
```

5. Design Cases

The SystemC and OCAPI-xl based approaches and extensions have been applied in the WCDMA, WLAN and MPEG-4 design cases in order to validate them at the system level, and to get experiences on the detailed and implementation design of reconfigurability on selected demonstrator platforms. The three cases represent different reconfiguration scenarios:

- The WCDMA detector case represents a study of applying partial dynamic reconfiguration in a mobile terminal
- The WLAN case presents a static reconfiguration scenario to allow generation of a family of wireless networking systems
- The MPEG-4 case presents a scenario where tasks are relocated between software and reconfigurable hardware.

5.1 WCDMA Detector

The application is an adaptive linear minimum mean-square error (LMMSE) detector [Heikkila, 2001] that is used in the downlink part of the WCDMA system. When compared to traditional RAKE detectors, it achieves 1- 4 dB

better performance in challenging channel conditions, uses a channel equalizer for performing multi-path correction instead of multiple RAKE fingers, and can be scaled to higher data rates by increasing clocking rates.

In the downlink data channel, each frame has 15 slots (2560 chips/slot) in a time period of 25 ms. The detector contains an adaptive filter, a channel estimator, a multi-path combiner and a correlator bank. In addition to the detector part, the searcher (code, frame and slot synchronization), de-interleaver and channel decoder of the WCDMA receiver are shown in Fig. 15.6. Starting with the C-representation of the WCDMA detector, the SystemC based approach was applied.

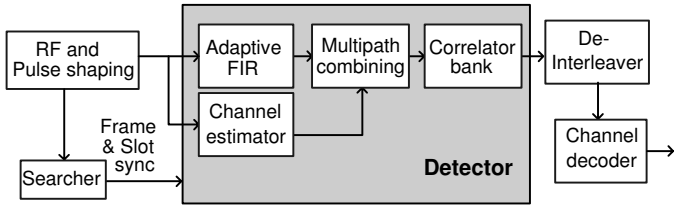


Figure 15.6. WCDMA base-band receiver.

The high-level estimation of the briefly optimized C code show that 1078, 1387, 463 and 287 FPGA look-up tables (LUTs) are required for the adaptive filter, channel estimator, combiner and correlator respectively. Based on the estimated resources, three different SystemC models were created in the system partitioning, mapping and performance simulation phase. For the fixed system, the processing time achieved was 1.12 ms per slot in an FPGA running at 100 MHz and the resource consumption was as mentioned above. In the case of dynamic partial reconfiguration, the processing was partitioned in two contexts, one containing the channel estimator and the other the rest three blocks. The benefit was that almost 50% of resource reduction could be achieved when compared to the fixed system, but at the cost of 8 times longer processing time. The pure software system resulted in 30 times longer processing time than the dynamic partial reconfiguration. In the partial reconfiguration case, device data such as configuration speed were taken from Xilinx Virtex-II Pro datasheet, which showed overwhelming reconfiguration latency that is almost 8 times of the processing time.

The WCDMA detector was implemented through the detailed and implementation design phases on the Memec Design's Virtex-II Pro FF1152 P20 Development Board using the Xilinx Embedded Development Kit (EDK) and ISE tools. Controlling of the WCDMA detector was handled by a SW process running on a PowerPC hardcore embedded in the FPGA platform. The run-time reconfiguration is realized using the Xilinx SystemACE solution. In

the implementation, 920 LUTs and 4 Block RAMs are required for the context containing the channel estimator, and 1254 LUTs, 6 Block RAMs and 12 Block Multipliers for the other context.

5.2 WLAN

The WLAN case presents a scenario, where reconfigurability is exploited to allow generation of a family of wireless networking systems. Specifically a dual mode WLAN - outdoor fixed wireless access system-on-chip is targeted. The WLAN was developed first based on the Hiperlan/2 standard [ETSI]. The system-on-chip realizes both MAC and Physical layer functionalities of the standard. The Hiperlan/2 physical layer is based on Orthogonal Frequency Division Multiplexing (OFDM). The MAC layer of Hiperlan/2 is based on a TDD/TDMA approach. Functionality upgrades/modifications will be developed in a second step to allow operation of the targeted system-on-chip in outdoor environments under a fixed wireless access scenario (in a non-overlapping fashion with Hiperlan/2). The simpler choice for the integration of the extra functionality after fabrication would be in the form of software upgrades. However due to the complexity of certain parts of the extra functionality (mainly corresponding to physical layer tasks) acceleration may be required. This can be achieved by including reconfigurable resources in the targeted system-on-chip.

The high level exploration for the development of the targeted dual standard system-on-chip was based on ANSI-C and OCAPI-xl. The Hiperlan/2 system was explored at the first step. Following an algorithmic exploration for the physical layer using MATLAB, a unified ANSI-C model of the targeted functionality (physical and MAC layers of Hiperlan/2) was developed. Physical layer blocks were modeled as parameterized procedures that produce the minimum amount of data required by the next procedure in the flow (pipelining). The procedures are parameterized with respect to different factors such as input data bit widths. Shared data for the inter module communication are presented as global variables. Communication between MAC layer modules is activated when all data produced by each module are ready (no pipelining). The physical layer part of the ANSI C model has a size of 9000 code lines while the MAC part includes 10000 code lines. Both parts use a common library of 1000 code lines.

Using the ANSI C model as input an OCAPI-xl model of the complete targeted functionality was developed. High level exploration was performed using high level OCAPI-xl processes (HLHW, HLSW) to evaluate different partitioning solutions. The complex tasks that are expected to be upgraded/duplicated in the future, i.e. need to be realized in reconfigurable hardware, were modeled as hardware (HLHW) processes. The physical layer part of the OCAPI-xl

model has a size of 13000 code lines while the MAC part includes 8000 code lines. Both parts use a common library of 500 code lines.

A prototype of the targeted system-on-chip was developed on the ARM Integrator platform. The platform hosts the AMBA AHB bus with all the required support peripherals for its operation (arbiter, interrupt controller etc.). Two ARM7 TDMI microprocessors on two separate boards - "core modules" are included in the platform and realize the software parts of the targeted system-on-chip. One processor acts as protocol processor while the other runs lower MAC functionality and controls the modem. The size of the code running on the protocol processor is 1.4 Mbytes while the size of the code running on the modem control processor is 50 Kbytes. The tasks that will be realized on ASIC or reconfigurable blocks in the targeted system-on-chip are mapped on two FPGAs - "logic modules". Each "logic module" hosts a Xilinx Virtex E 2000 FPGA. The total utilization of the two FPGAs resources is: 405 I/Os, 31450 function generators, 23416 CLBs and 14912 D flip flops.

5.3 MPEG-4 Decoder

The initial C/C++ code was obtained from FDIS (Final Draft International Standard) sources. First, the ATOMIUM toolset [ATOMIUM] for pruning of code, data transfer and storage exploration and advanced source-to-source transformations were applied to increase the performance and reduce the power of the system. Applying automatic pruning with functionality test bench reduced code to 40 % of its original size and further manual reorganization and rewriting reduced the code size by factor 5.4. The memory optimizations were driven in two phases, frame-based to macro-block-based data flow transformation and introduction of block-based data flow, with the aim of reducing the number of accesses and improvement of the locality of data. In the next steps of the design, platform dependent optimizations were applied. The targeted platform for implementing the MPEG-4 decoder was Xilinx's Multimedia Development Board containing Virtex-II FPGA with a single embedded MicroBlaze soft processor, surrounded with external ZBT memory banks.

After the optimization phase, the design proceeded with modeling of performance estimation in OCAPI-xl environment. A number of small tests were done on the board to find the operator execution times and memory access times and to select proper memory architecture. Taking into account the ATOMIUM analysis results from optimization phase, the most CPU time and memory access demanding blocks of the decoder were selected to become the candidates for HW acceleration. From the OCAPI-xl point of view, the C/C++ code representing their behavior was rewritten to OCAPI-xl managed SW processes and later refined to high-level hardware processes for clock-true sim-

ulation and OCAPI specific processes for HDL code generation. Exploiting OCAPI-xl Operation Set Simulator, the design was modeled in two flavors:

- Configured as pure SW version of MPEG-4 decoder running on soft processor core
- Configured as HW accelerated version, where most cycle demanding blocks have been implemented in reconfigurable HW.

OCAPI-xl high-level modeling of context switching extension allowed the performance estimation of two reconfigurable scenarios, with the ability to model the reconfiguration time. The design was mapped on xc2v2000 Virtex-II FPGA with 46% utilization (5000 slices) for HW accelerator and 71% utilization (7703 slices) for the whole decoding system. It uses 33% of available 18x18 multipliers, 76% of block RAMs and 52% of LUTs.

6. Conclusions

Reconfigurability is a promising technique in a SoC to obtain software-type flexibility, while maintaining hardware-type computational capacity. The related implementation technologies and architectures are still under development. It looks obvious that different reconfigurable technologies will become accepted in different application and business domains. Static reconfiguration does not change the design methodology radically, but dynamic reconfiguration adds a new dimension to SoC design flow.

We presented support extensions to SystemC and OCAPI-xl that address the system-level design of reconfigurable parts in the context of the SoC design flow. The effects of reconfigurability are studied before committing to a specific reconfigurable technology or architecture instance. Experiences of extending SystemC and OCAPI-xl show that this is a viable approach and credible data for system-level decision-making can be produced with reasonable effort. All the material and tools of SystemC are still valid.

The case studies represented three different reconfigurability scenarios: in the WCDMA detector case dynamic partial reconfiguration was explored, in the WLAN case static reconfiguration was applied for obtaining a family of networking systems, and in the MPEG-4 decoder case task relocation between software and reconfigurable hardware was studied.

Acknowledgments

This work is partially supported by the European Commission under the contract IST-2000-30049 ADRIATIC, and partially by the participating organizations: IMEC, INTRACOM, NOKIA, STMB and VTT.

References

ATOMIUM: `http://www.imec.be/design/atomium`.

Becker J., Hartenstein R. (2003). *Configware and Morphware Going Mainstream*. Journal of System Architecture, Vol. 49. pp. 127-142.

Bobda C. (2003). *Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement*. Dissertation. University of Paderborn. 90 p.

Callahan T., Hauser J., Wawrzynek J. (2000). *The Garp Architecture and C Compiler*. IEEE Computer., pp 62 - 69.

Cardoso J.M.P., Weinhardt M. (2003). *From C Programs to the Configure-Execute Model*. Proc. of 2003 Design Automation and Test in Europe Conference and Exhibition. Munich, Germany, March 3 - 7, 2003, pp. 576 - 581.

Compton K., Hauck S. (2002). *Reconfigurable Computing: A Survey of Systems and Software*. ACM Computing Surveys, Vol. 34, No. 2. 171-210.

ETSI: *Broadband Radio Access Networks (BRAN)*. HIPERLAN type 2; Physical (PHY) layer, V 1.2.1 (2000-11).

Goldstein S.C., Schmit H., Mihai B., Cadambi S., Matt M., Taylor R.R. (2000). *PipeRench: A Reconfigurable Architecture and Compiler*. IEEE Computer. April 2000, pp. 70 - 77.

Grötker T., Liao S., Martin G., Swan S. (2002) *System Design with SystemC*. Kluwer Academic Publishers, Boston, 240 p.

Hartenstein R. (2001). *Reconfigurable Computing - Architectures and Methodologies for System-on-Chip*. SoC technology seminar "Enabling Technologies for System-on-Chip Development". Tampere, Finland, November 19-20.

Heikkila M.J. (2001) *A Novel Blind Adaptive Algorithm for Channel Equalization in WCDMA Downlink*. 12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, Volume:1, pp . A-41 - A-45.

Maestre R., Kurdahi F.J., Fernandez M., Hermida R., Bagherzadeh N., Singh H. (2001). *A Framework for Reconfigurable Computing: Task Scheduling and Context Management*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, Issue 6, pp. 858 - 873.

Noguera J., Badia R.M. (2003). *System-Level Power-Performance Trade-offs in Task Scheduling for Dynamically Reconfigurable Architectures*. Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. pp. 73 - 83.

Ocapi-XL: `http://www.imec.be/design/ocapi`.

Pelkonen A., Masselos K., Cupak M. (2003) *System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC*. 17th International Par-

allel and Distributed Processing Symposium (IPDPS 2003). Nice, France, 22 - 26 April 2003. IEEE Computer society, pp. 174 - 181.

Qu Y., Soininen J.-P. (2003) *Estimating the Utilization of Embedded FPGA Co-Processor*. 2003 Euromicro Symposium on Digital Systems Design (DSD 2003): Architectures, Methods and Tools. Antalaya, Turkey, 3 - 5 Sept. 2003. IEEE Computer Society. Los Alamitos, pp. 214 - 221.

Srikanteswara S., Palat R.C., Reed J.H., Athanas P. (2003). *An Overview of Configurable Computing Machines for Software Radio Handsets*. IEEE Communications Magazine, pp. 134 - 141.

SUIF: http://suif.stanford.edu/.

Taylor M.B., Kim J., Miller J., Wentzlaff D., Ghodrat F., Greenwald B., Hoffman H., Johnson P., Lee J.-W., Lee W., Ma A., Saraf A., Seneski M., Shnidman N., Strumpen V., Frank M., Amarasinghe S., Agarwal A. (2002). *The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs*. IEEE Micro, Volume: 22, Issue: 2, pp. 25 - 35.

Varicore: `http://www.actel.com`.

Venkataramani G., Najjar W., Kurdahi F., Bagherzadeh N., Bohm W., Hammes J. (2003). *Automatic Compilation to a Coarse Grained Reconfigurable System-on-Chip*. ACM Transactions on Embedded Computing Systems, Vol. 2, No. 4, pp. 560-589.

Xilinx Virtex II Pro: `http://www.xilinx.com`.

XPP: `http://www.pactcorp.com`.

IV

INVITED CONTRIBUTIONS

Introduction

Wolfgang Müller
Paderborn University, Germany
wolfgang@c-lab.de

Christoph Grimm
University of Frankfurt, Germany
christoph@grimm-www.de

The following two contributions come from invited presentations of the Automotive Invited Session and LFSV (Languages for Formal Specification and Verification) workshop. They both give most recent insights to emerging technologies and to new trends in the respective domains.

In system verification we can currently find several upcoming approaches for new directions towards the convergence of the classical simulation and formal verification like model checking, equivalence checking, and formal refinement. The next chapter compares two complementary approaches for the checking of properties, which are given by temporal PSL specifications. The first is a SystemC simulation based approach, whereas the second one introduces a unique checking algorithm for formal verification.

In automotive system design, the AUTOSAR initiative aims at introducing a standardized middleware for automotive applications. The second chapter gives a comprehensive introduction to the AUTOSAR initiative.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 273.

Chapter 16

SYMBOLIC MODEL CHECKING AND SIMULATION WITH TEMPORAL ASSERTIONS*

Roland J. Weiss, Jürgen Ruf, Thomas Kropf and Wolfgang Rosenstiel
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{weissr,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de

Abstract

Assuring correctness of digital designs is one of the major tasks in the system design flow. In the last decade, traditional functional verification techniques like simulation with test benches and monitors have been augmented with formal techniques. Formal techniques can be divided into equivalence and property checking. Equivalence checking tools at the gate level are now part of most design flows. However, property checking is still subject to intensive research efforts due to the omnipresent state explosion problem.

Property checking is performed in two steps. First, a set of property specifications has to be written in an appropriate formalism. The system model is then checked against these properties. A property checking tool then either reports the absence of defects on the explored paths or generates a counter example trace.

In this work we show that formal property specifications can be reused in all phases of the verification process, including both functional and formal approaches. The properties provide the link between these usually rivaling techniques. First, we discuss current formalisms for specifying temporal properties. Then we present two automata based techniques for checking temporal properties given in the standardized Property Specification Language (PSL). The first approach checks the properties during SystemC simulation, whereas the second approach performs fully formal property checking of the temporal properties against a transition system employing a unique checking algorithm.

*The results described in this article have been achieved in the course of the DFG project GRASP within the DFG Priority Programme 1064.

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 275–291.

1. Introduction

Real-time systems pervade almost every aspect of our daily life. Accelerating a contemporary vehicle initiates a plethora of processes involving micro controllers: fuel injection should be optimized for economical fuel usage and for smooth engine operation, wheelspinning should be avoided based on data provided by special sensors, and so on. The same applies to aerospace industry, medical systems, home entertainment, telecommunication, large-scale industrial manufacturing, and of course the classical computer industry. Even in outdoor activities we rely on integrated circuits in equipment like wrist watch heart rate monitors and GPS receivers.

A study by the Center of Automotive Research (CAR) for ADAC, a major german automobile association, revealed that 59.2% of car breakdowns are caused by faults in the automobiles' electronics and electricity systems. This number increased from 50.5% five years ago, whereas the overall figures remained constant. Furthermore, all car brands are affected by this problem. This observations show that the whole car industry is facing a fundamental challenge in building reliable products with a substantial amount of integrated digital hardware and software components. Other industry areas also suffer from these phenomenons.

Therefore, establishing the correctness of hardware/software systems poses an increasingly difficult and time-consuming challenge in the design process. Two aspects primarily motivate the significance of the verification process:

Safety-critical systems. These systems mandate error-free operation because malfunctioning could endanger human life or the environment. Obviously, an erroneous circuit in a car controller unit constitutes a major threat for passengers and road users. Medical systems and power plants are other important constituents in this category.

Economic risks. It is of primary importance to detect design errors in early stages of the development process. Once a hardware chip is shipped, it becomes extremely expensive to fix an overlooked fault. The Pentium floating point bug has cost Intel millions of dollars because of an insufficient verification process. A simple overflow handling error in one of the control systems of the Ariane 5 rocket lead to the loss of the 7 million dollar rocket.

The ever increasing system complexity and shorter development cycles make verification the bottleneck in the design process. Nowadays, verification consumes up to 80% of the development time.

Simulation of the design under verification is the predominating validation technique. A testbench provides an executable model with stimuli and monitors the resulting outputs. However, for large systems simulation cannot pro-

vide a complete coverage of the system. Simulation time is becoming a prohibiting factor.

This has ignited interest in formal methods that can provide better coverage and run more efficiently in certain scenarios. Equivalence checking has become state of the art in verifying evolutionary changes in designs even for very large hardware circuits.

Unfortunately, if no *golden design* exists, this technique is not applicable. Formal property checking is bridging this gap. Property checking tools try to automatically prove properties extracted from the design description against a system model, or to generate a counter example trace if the property is violated. However, linear temporal properties checked by formal tools can also be monitored during traditional simulation runs. Thus, property specifications provide an important link between functional and formal verification techniques.

The rest of this paper is structured as follows. Next we introduce formalisms for specifying temporal properties, including CCTL, FLTL and the specification language PSL. Thereafter, we explain techniques for checking real-time properties in more detail. Then we present some experimental results and conclude.

2. Property Specification

A property is a description of design intent [Cohelo and Foster, 2004]. This central statement reveals the importance of property specifications. A property specification is the formalization of design intent in a human and machine readable format with a clearly defined semantics. In order to discuss properties in more detail, it is beneficial to take a layered view on them. Properties are composed of three layers:

1 The *Boolean layer* consists of propositions and Boolean connectives.

2 The *temporal layer* adds operators for temporal reasoning to the Boolean layer.

3 The *verification layer* provides indicators for verification tools how to apply the property.

The third layer is used to control the high-level bahavior of the verification tools, e.g. if a property violation should stop the verification process or simple emit a logging message. The first two layers make up the actual property that relates parts of the system under verification, thus describing desired or error states.

2.1 Temporal Logics

We are mainly dealing with reactive systems, and temporal logic is a formalism for describing transition sequences in such systems. A temporal logic provides *path quantifiers* A (all) and E (exists), and *temporal operators* X (next), F (eventually/in the future), G (globally), and U (until) additional to the typical boolean connectives.

The most widely used temporal logics in model checking and related formal verification techniques are *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). Both describe overlapping subsets of CTL*, i.e. CTL* is expressive enough to state all formulas from LTL and CTL, but there exist formulas in LTL that are not expressible in CTL and vice versa. For a more detailed discussion on these fundamental temporal logics refer to chapter 3 in [Clarke et al., 1999].

A major characteristic of temporal languages is the underlying model of time. In branching temporal logics, a moment in time can branch into various futures. Thus, infinite computation trees describe the systems that are subject to property checking. CTL is one such temporal logic, and it is well suited for algorithmic verification. Checking a transition system against a property given in CTL takes time linear in the length of the property specification. However, in linear temporal logics like LTL each moment in time has only one possible future. Therefore, formulas in linear temporal logics are interpreted over linear sequences that describe one computation of a system. Model checking takes time exponential in the length of a LTL specification. However, LTL is commonly regarded as more intuitive. Furthermore, dynamic validation is inherently linear as computation sequences are generated. This allows linear temporal logic specifications to be used in contexts ranging from dynamic validation to full formal verification. A thorough discussion on the trade-offs between branching and linear temporal logics is presented in [Vardi, 2001].

All these logics have in common that they express time only implicitly, e.g. a property specifies that a state is eventually or never reached. However, real-time systems such as production automation systems often require conformance to strict time bounds. Time constraints are very important to maximize throughput times and to minimize wait times. Furthermore, timing constraints also have a safety aspect, since actions in production automation systems consume time, and they have to be scheduled such that no accident occurs.

In order to make time constraints explicit in property specifications we have introduced a variant of CTL called *Clocked CTL* (CCTL), and a variant of LTL called *Finite Linear Time Temporal Logic* (FLTL). We will briefly describe these two temporal logics now.

2.2 CCTL

CCTL [Ruf and Kropf, 2003] is a temporal logic extending CTL with quantitative bounded temporal operators. In contrast to CTL its semantics is defined over interval structures and it contains two new operators which make the specification of timed properties easier. It is a variant of RTCTL [Emerson et al., 1991]. The syntax of CCTL is the following:

DEFINITION 1 *Let P be a set of atomic propositions, $m \in \mathbb{N}$, and $n \in \mathbb{N} \cup \{\infty\}$. The set of all syntactically correct CCTL formulas is the smallest set satisfying the following properties:*

- $P \subseteq \mathcal{F}_{CCTL}$
- *if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi \in \mathcal{F}_{CCTL}$*
- *if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also $\mathsf{AX}_{[m]}\phi$, $\mathsf{AG}_{[m,n]}\phi$, $\mathsf{AF}_{[m,n]}\phi$, $\mathsf{A}(\phi\mathsf{U}_{[m,n]}\psi)$, $\mathsf{A}(\phi\mathsf{C}_{[m]}\psi)$, $\mathsf{A}(\phi\mathsf{S}_{[m]}\psi) \in \mathcal{F}_{CCTL}$*
- *if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also $\mathsf{EX}_{[m]}\phi$, $\mathsf{EG}_{[m,n]}\phi$, $\mathsf{EF}_{[m,n]}\phi$, $\mathsf{E}(\phi\mathsf{U}_{[m,n]}\psi)$, $\mathsf{E}(\phi\mathsf{C}_{[m]}\psi)$, $\mathsf{E}(\phi\mathsf{S}_{[m]}\psi) \in \mathcal{F}_{CCTL}$*

We also support the temporal operators C (conditional) and S (successor). Operator C requires formula ψ to hold if ϕ was true in the previous $m-1$ steps, and operator S is a special case of operator U with $m = n$.

All interval operators can also be used with a single time-bound. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the EX-operator has no time bound, it is implicitly set to one. A definition of the formal semantics of CCTL is given in [Ruf and Kropf, 1999].

Example. Signals a and b will become true simultaneously within the next 30 time steps: $\mathsf{EF}_{[30]}a \wedge b$.

2.3 FLTL

FLTL extends LTL with bounded temporal operators. The main difference however lies in the definition of the formal semantics. LTL is defined over infinite sequences, whereas FLTL is defined over finite sequences. The reason for defining FLTL over finite state sequences comes from its application in simulation for validating formal properties. A simulation run always generates only a finite trace of the system's behavior. If the simulation terminates one does still like to argue about the specification's state, i.e. if the formula holds or not. Because this predication is not always decidable with finite sequences, the definition of the formal semantics of FLTL applies a third state: *pending*.

For a detailed discussion and definition of the semantics of FLTL refer to [Ruf et al., 2001].

DEFINITION 2 *Let P be a set of atomic propositions, $m \in \mathbb{N}$, and $n \in \mathbb{N} \cup \{\infty\}$. The set of all syntactically correct FLTL formulas is the smallest set satisfying the following properties:*

- $P \subseteq \mathcal{F}_{FLTL}$
- *if $\phi, \psi \in \mathcal{F}_{FLTL}$, then also $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi \in \mathcal{F}_{FLTL}$*
- *if $\phi, \psi \in \mathcal{F}_{FLTL}$, then also $\mathsf{X}_{[m]}\phi$, $\mathsf{G}_{[m,n]}\phi$, $\mathsf{F}_{[m,n]}\phi$, $\phi\mathsf{U}_{[m,n]}\psi \in \mathcal{F}_{FLTL}$.*

Example. Signal a will become active for the first time at time step 300: $\neg a \mathsf{U}_{[300,300]} a$.

2.4 PSL

The need for a common specification language accross tool and vendor boundaries culminated in PSL, the Property Specification Language by Accellera [Accellera, 2004]. PSL is divided into four layers. In addition to the layers already mentioned above it introduces a *modeling layer*. This layer contains means to model behavior of design inputs, e.g. the Verilog flavor supports integer ranges, structures, non-determinism and built-in functions.

PSL provides temporal operators both from branching (CTL) and linear (LTL) time temporal logics. Of course, it depends on the verification tool and technique which operators actually apply. However, PSL provides the syntax and semantics for these operators, and tools are free to ignore properties that make no sense in their verification process. Boolean operators, sequence expressions and linear time temporal operators comprise the PSL Foundation Language (FL), branching time temporal operators are subsumed as Optional Branching Operators (OBEs).

Boolean expressions. are composed of signals and variables available in the HDL description. Therefore, PSL supports various flavors of Boolean expressions, depending on the targeted HDL. As of version 1.1, SystemVerilog, Verilog, VHDL and GDL flavors exist.

Temporal operators. are available both in typical one-letter notation like X for the next operator and in an extended, readable version, in this case `next`. Furthermore, PSL differentiates between weak and strong operator forms: "The strong form requires that the terminating condition eventually occur, while the weak form makes no requirements about the terminating condition" [Clarke et al., 1999].

Sequential Extended Regular Expressions. (SEREs) describe sequences of Boolean conditions that are recursively built from simple Boolean expressions. SEREs can be constructed by concatenation '`;`', fusion[1] '`:`', parallel matching (SERE and), and providing alternatives (SERE or). Also, repetition operators exist that allow various consecutive or nonconsecutive repetitive concatenations of SEREs. Finally, SEREs can be grouped using braces and PSL provides special sequence implication operators: '`|->`' and '`|=>`'.

2.5 Higher level property specification

Formulating property specifications in temporal logics has turned out to be difficult even for engineers with a mathematical background. Therefore, efforts were taken to provide means of specifying properties at a higher abstraction level such that they are easier to grasp for the human reader.

1 Graphical notations of properties with Live Sequence Charts (LSC).

2 Natural language property specification based on specification patterns.

The brief discussion on higher level property specification in this section will focus on these two techniques.

Live Sequence Charts. Live Sequence Charts [Damm and Harel, 2001] were introduced to overcome the major shortcomings of Message Sequence Charts (MSC) and Sequence Diagrams (SD) from UML [Object Management Group (OMG), 2003]. The criticism of MSCs and SDs concentrates on these points:

- Only an existential view of the system is supported.
- The point of activation of the chart is unclear.
- No means to specify the necessity to reach certain points in the chart.
- There is no formal semantics for SDs.

In [Klose et al., 2002], an algorithm is detailed that allows extraction of an automaton from a LSC. This automaton is then checked against a system model. Thus, LSCs are used as graphical notation for property specifications.

Natural language property specification. Another idea to facilitate property specifications is to use natural language expressions and convert them into temporal logic formulas. The idea originated in the context of specification patterns [Dwyer et al., 1999]. These patterns classify common property specifications into categories for later reuse. In [Flake et al., 2002], a predefined grammar consisting of structured English sentences is introduced with which

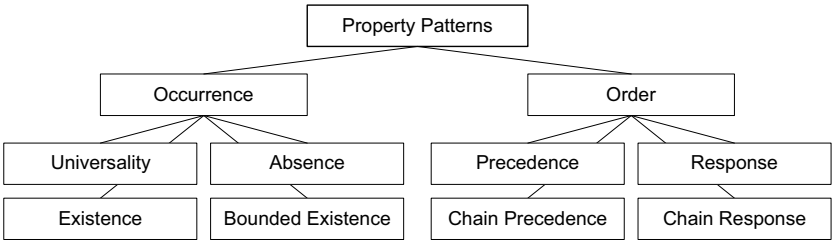


Figure 16.1. The pattern system hierarchy.

the user can specify properties. These will be translated into CCTL formulas thereby fixing the semantics of the structured sentences.

RT-OCL. Industrial modeling mostly relies on UML. In accompanying work [Flake et al., 2004], a state-oriented real-time OCL extension (RT-OCL) was developed that allows modelers to specify state-oriented real-time constraints over UML models. The semantics of RT-OCL is described by mapping temporal OCL expressions to CCTL formulas. Thus, we have a transition path from UML to our lower level formalisms.

2.6 Property specification patterns

Property specification patterns represent a slightly different approach to further practical usage of formal property specifications [Dwyer et al., 1999]. These patterns try to capture knowledge of specification experts and present frequent properties in a reusable form. A specification pattern consists of the following five items:

1 The pattern's name.

2 A statement of the pattern's intent.

3 Mappings into specification formalisms. Currently, the following ones are supported: CTL, LTL, QRE (Quantified Regular Expressions), GIL (Graphical Interval Logic) and INCA.

4 Examples of known uses.

5 Relationships to other patterns.

The devised pattern system consists of eight basic patterns that are grouped into two main categories: *order* and *occurrence*. Figure Fig. 16.1 shows this pattern system.

A survey of freely available property specifications was performed to assess the validity of the proposed pattern system. The survey discovered that from

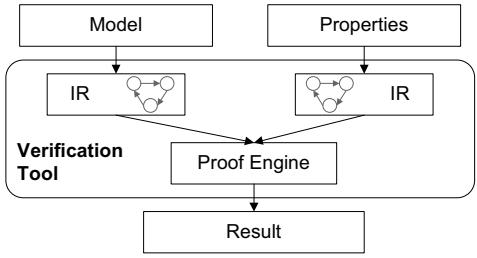


Figure 16.2. Structure of a property checking tool.

the 555 examined specifications 92% of the specifications matched properties from the 8 base patterns. Furthermore, 80% of specifications were covered by the three patterns response, universality and absence.

Summarizing, temporal logics are the semantic foundation for formal property specification. Several approaches exist to ease creation of new specifications either by providing higher level abstractions or making available for reuse often encountered property specifications.

In general, formal property specification improves the understanding of a system and its requirements, as well as the communication of its design intent among involved parties.

3. Property Checking

Properties are checked against a given model. In formal verification tools, this model is typically formulated as a variant of Kripke structures [Clarke et al., 1999], whereas functional verification works directly on the production implementation. The property checking tools proceed by optionally converting specifications and model into an internal representation (IR) such that the proof engine can apply the verification algorithm efficiently (see Fig. 16.2).

Properties given in specification languages with a linear time model like LTL or PSL FL can be shared by functional and formal verification tools. Therefore, we describe two tools and their proof engines that work on such properties. Both accept specifications in PSL FL, augmented with time bounds (see section 2.1).

3.1 Time Bounded Property Checking with SymC

In [Ruf et al., 2003] we proposed a formal verification technique for time bounded property checking[2]. The technique performs forward image computation for state traversal, a characteristic shared by forward model checking [Iwashita and Nakata, 1997][3]. Properties are specified with FLTL formulas, therefore a tight integration with other property checking tools is provided.

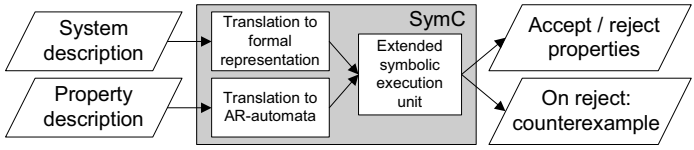


Figure 16.3. Overview of SymC operation.

The temporal logic formulas are converted to special finite state machines, so-called AR-automata [Ruf et al., 2001], which can then be used in the symbolic execution phase.

For system description, SymC uses its own input language that captures finite state machines. Such SymC models can either be written by hand, or they can be generated from Verilog netlists or RAVEN [Ruf and Kropf, 2003] models. The automatic transformation from RAVEN to SymC input files allows us to check one model description with both verification tools. Fig. 16.3 shows the general operation of SymC, the tool based on this approach.

The current implementation of SymC uses a BDD-based approach, where one symbolic execution step corresponds to one forward image computation of the given state set. In contrast to standard state space traversal techniques, in this method we forget already visited states. The symbolic execution is stopped if a given time bound k is reached, or the property can either be proven correct or incorrect in the current state. The time bound k is either predefined by the user or determined by the formula if no infinite operators are used.

Both, the system description and the AR-automata, are translated to BDDs. In order to avoid the construction of the complete transition relation we use a set of transition relation partitions together forming the whole relation T.

The main iteration of our checking algorithm works in two steps. In the first step we compute the successor states of the AR-automata and we check whether a formula is accepted or rejected. In the second step of each iteration we perform one symbolic execution step on the system under inspection. During image computation we build the conjunction of all partitions on-the-fly to obtain the successor state set. We do not build the complete state space, a feature shared with bounded model checking [Biere et al., 2003]. Rather, from a given start set we visit states reachable within a given time bound. The choice of the start set allows tuning a SymC execution either towards complete coverage or towards smaller memory footprint and faster runtime.

We apply several standard optimizations like cone of influence reduction and early quantification [Clarke et al., 1999] to speed up the state space traversal. Also, two optimizations unique to our approach have proven very effective in speeding up SymC. First, *pruning* removes already accepted states. This can be done safely because successors of accepted states stay accepted. Sec-

ond, *splitting* tracks the size of the current state set. Once it reaches a certain threshold value, the state set is split and the partitions are handled separately. Splitting improves the runtime efficiency of SymC in most cases because of early result detection and because the memory consumption is reduced. However, for pathological the splitting and state set stacking can even increase the verification time.

Experimental results [Ruf et al., 2003] show that SymC outperforms other property checking methods for certain classes of systems and properties. This technique is well suited for properties with large time bounds.

3.2 Simulation with SystemC

Simulation and modeling with SystemC. SystemC [Grötker et al., 2002] is a C++ library developed to support modeling at the system level, but also at other levels of abstraction, such as register transfer level (RTL). The modeled systems may be composed both of hardware and software components. The whole library is written in ISO/ANSI compliant C++ [ISO/IEC, 2003] and therefore runs on all standard compliant C++ compilers. It constitutes a domain specific language embodied in the library's data types and methods.

The SystemC core language is built around an event-driven simulation kernel which allows efficient simulation of compiled SystemC models. Processes in SystemC are nonpreemptive, thus one erroneous process can deadlock the simulator. The SystemC library provides abstractions for hardware objects that allow modeling from RTL up to transactional level. These abstractions include:

- Processes for modeling of simultaneously executing hardware units.
- Channels for modeling the communication of processes, as well as ports and interfaces for flexible interchangeability of channels.
- Events for modeling the interaction between processes and channels.
- Modules for modeling the structural and hierarchical composition of the described system models.
- Hardware specific data types like signals, bitvectors, and floating point numbers of fixed and variable width.
- A notion of time is supported with clock objects. Clocks generate timing signals such that events can be ordered in time.

The SystemC library and reference implementation of the simulation kernel are available for free [VA Software Corporation and Open SystemC Initiative, 2004] in source code. Companies are encouraged to provide Intellectual Property (IP) cores in this standardized description language.

Verification extensions for SystemC. Despite the fact that we are already able to handle models with SymC where traditional model checkers like RAVEN are running out of memory, we still run into problems for very large designs. Here, we try to take advantage of simulation, but enrich it with formal methods.

The first step was to extend SystemC models with assertions expressed as temporal logic formulas [Ruf et al., 2001]. Depending on the translation scheme, these assertions are either compiled into a library that is linked against the SystemC executable, or they can be added dynamically during execution. A special intermediate language [Krebs and Ruf, 2003] supports these different translation schemes. With this technique, checking executable system models against formal properties can start at the highest abstraction levels. Also, the same properties can be checked both with SystemC and SymC.

The methodology just mentioned is instrumental in another approach that combines functional and formal verification. Once the system model has been converted into a transition system it can be model checked with SymC or RAVEN. However, for large designs we run into the state explosion problem. In order to still be able to perform limited checking, we support the following technique.

The model given as transition system is translated into an executable SystemC model and a temporal formula is checked against this model during simulation using our checker library. The formula has to conform to this structure (equation 16.1 is the CTL flavor for RAVEN, equation 16.2 the LTL flavor for SymC):

$$\mathsf{AG}(trigger\_state \rightarrow required\_temporal\_behavior) \tag{16.1}$$

$$\mathsf{G}(trigger\_state \rightarrow required\_temporal\_behavior) \tag{16.2}$$

Whenever formula $trigger\_state$ is true during simulation, the current system state is dumped to a state file.

After simulation, we use a formal property checking tool to check formula $required\_temporal\_behavior$ against the original transition system. The dumped state files are used to set up the initial states for the model checker. The checked formula is restricted to a finite time bound t_{max}, thus we avoid the construction of the complete state space. Of course, we can now only argue about the system behavior in this time bounded state space. The state space in this time bound is examined exhaustiveley according to the given property $required\_temporal\_behavior$.

Summarizing, the user guides the verification process by pointing out trigger states from which local state space traversal is performed within a limited time scope. Fig. 16.4 shows the overall flow of this combined approach, using our real-time model checker RAVEN as property checking tool.

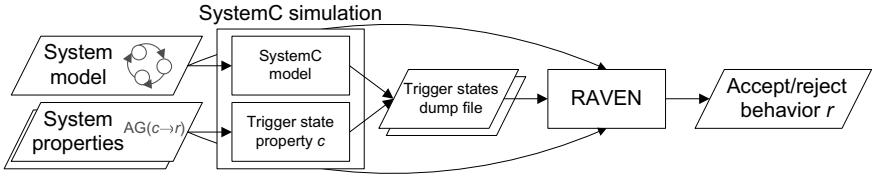


Figure 16.4. Structure of the combined approach using SystemC and RAVEN.

3.3 Experimental Results

In [Ruf et al., 2004] we descibe a holonic material transport system. This system was coded as transition system in the RAVEN input language (RIL). The RIL model was translated automatically into a SystemC and a SymC model. Thus, we have one reference model.

Model. The transport system consists of an input station, three machines, an output station and automatic transport vehicles, the so-called *holons*. Two of the three machines are for workpiece processing, one is for cleaning. All holons are identical. The task of the holons is to move workpieces to the two processing units. After processing, the workpieces have to be moved to the cleaning machine. From this station the workpieces have to be transported to the output station. Effectively, a workpiece travels from the input station to the output station, and visits all machines on the way in order. Transportation of the same workpiece can be accomplished by different holons, because a holon's task is renegotiated after it has dropped a workpiece at a unit. We have a model generator that allows setting the size of the moving area, the position of machines, and the number and positions of holons.

Properties. We check that a work piece is evetually delivered to the output station within a certain time bound (P_1), and that holons never collide[4] (P_2).

Results. We tested models with one (M_1), two (M_2) and three holons (M_3) on a Linux PC with a 2.8 GHz Pentium 4 processor and 1 GB of RAM installed.

We checked property P_1 with RAVEN and SymC and report the results with different time bounds n in Fig. 16.5 and Fig. 16.6.

We see that RAVEN is unable to check properties for models with more than one holon in the system. However, once all preparations for model composition have finished, the actual model checking does not vary significantly depending on time bound n.

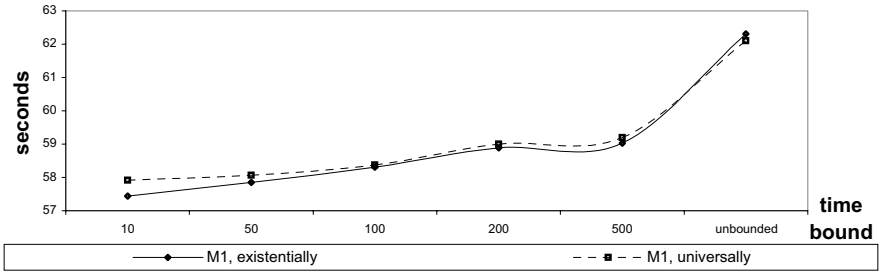


Figure 16.5. Results for checking P_1 with RAVEN.

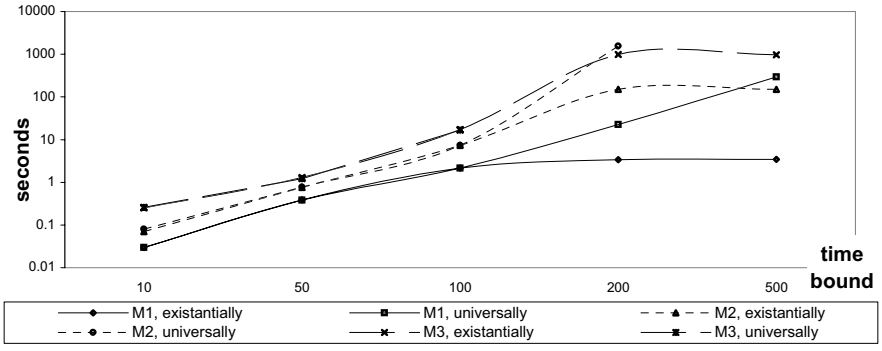


Figure 16.6. Results for checking P_1 with SymC with logarithmic time scale.

We are able to check properties with SymC against all three models within given time bounds. In most cases, SymC outperforms RAVEN. However, once SymC has to traverse a huge state space for formulas that are still pending RAVEN may overtake again, e.g. universal quantification with a time bound of 500 steps. For results of checking P_2 refer to [Ruf et al., 2004].

The experiments show that depending on the checked properties and the requirements on the checks, different verification tools excel in different areas. Model checking gives the user complete coverage of the model, however it suffers from the state explosion problem. Here, semi-formal approaches can help to validate properties. An important task of the verification engineer is to select the appropriate tool to handle a specific verification problem.

4. Summary and Future Work

In this paper we have emphasized the importance of formal property specification as a key ingredient for enhancing modern design flows. These specifications can be reused in various phases of the development process both by funtional and formal verification tools.

Property checking enhances system reliability by detecting functional errors early during system design and implementation. Therfore, it has great potential to speed up development and regain the investment by creating more reliable systems. Two property checking techniques were explained in more detail.

We are currently enhancing our property checker SymC in two directions. A parallel version should allow checking larger models as well as shortening the verification time. Furthermore, we also add a SAT-based engine to the already existing BDD-based proof machinery.

Notes

1. Fusing two SEREs means that the first sequence and the second sequence overlap by one clock cycle.
2. SymC is available at `www-ti.informatik.uni-tuebingen.de/~fmg/symc`.
3. However, our property checking algorithms are quite different.
4. The holonic system does not contain collision freeness by design.

References

Accellera Organization (2004). Property Specification Language (PSL), version 1.1. `www.eda.org/vfv`.

Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M., Strichman, Ofer, and Zhu, Yunshan (2003). Bounded model checking. In Zelkowitz, Marvin, editor, *Highly Dependable Software*, volume 58 of *Advances in Computers*. Academic Press.

Clarke, Edmund M., Grumberg, Orna, and Peled, Doron E. (1999). *Model Checking*. The MIT Press.

Coelho, Claudionor Nunes Jr. and Foster, Harry D. (2004). Asserstion-based verification. In Drechsler, Rolf, editor, *Advanced Formal Verification*, pages 167–204. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Damm, Werner and Harel, David (2001). LSCs: Breathing life into message sequence charts. *Journal on Formal Methods in System Design*, 19(1):45–80.

Dwyer, Matthew B., Avrunin, George S., and Corbett, James C. (1999). Patterns in property specifications for finite-state verification. In *21. International Conference on Software Engineering*, pages 411–420. ACM Press.

Emerson, E. Allen, Mok, Aloysius K., Sistla, A. Prasad, and Srinivasan, Jai (1991). Quantitative temporal reasoning. In Clarke, Edmund M. and Kurshan, Robert P., editors, *Computer Aided Verification, 2nd International Workshop*, volume 531 of *Lecture Notes in Computer Science*, pages 136–145. Springer.

Flake, Stephan, Müller, Wolfgang, and Ruf, Jürgen (2002). Structured english for model checking specification. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 3. GI/ ITG/GMM Workshop, pages 99–108. VDE Verlag.

Flake, Stephan, Müller, Wolfgang, Pape, Ulrich, and Ruf, Jürgen (2004). Specification and Formal Verification of Temporal Properties of Production Automation Systems. In: Ehrig, Harmut et al., editors, *Integration of Sofware Techniques for Applications in Engineering*, Volume 3147 of *Lecture Notes in Computer Science*, pages 206–226. Springer Verlag.

Grötker, Thorsten, Liao, Stan, Martin, Grant, and Swan, Stuart (2002). *System Design with SystemC*. Kluwer Academic Publishers.

ISO/IEC (2003). *Programming Languages – C++*. Number 14882:2003 in JTC1/SC22 – Programming languages, their environment and system software interfaces. International Organization for Standardization, 2. edition.

Iwashita, Hiroaki and Nakata, Tsuneo (1997). Forward model checking techniques oriented to buggy designs. In *Proceedings of the 1997 IEEE/ACM International Conference on CAD*, pages 400–4004. ACM and IEEE Computer Society Press.

Klose, Jochen, Kropf, Thomas, and Ruf, Jürgen (2002). A visual approach to validating system level designs. In *15th International Symposium on Systems Synthesis*, pages 186–191. ACM Press.

Krebs, Andreas and Ruf, Jürgen (2003). Optimized temporal logic compilation. *Journal of Universal Computer Science, Special Issue on Tools for System Design and Verification*, 9(2):120–137.

Object Management Group (OMG) (2003). Unified Modeling Language (UML), Version 1.5. `www.omg.org`. Document formal/03-03-01.

Ruf, Jürgen, Hoffmann, Dirk W., Kropf, Thomas, and Rosenstiel, Wolfgang (2001). Simulation-guided property checking based on a multi-valued AR-

automata. In Nebel, Wolfgang and Jerraya, Ahmed, editors, *Design, Automation and Test in Europe, DATE 2001*, pages 742–748. IEEE Press.

Ruf, Jürgen and Kropf, Thomas (1999). Modeling and checking networks of communicating real-time process. In Pierre, Laurence and Kropf, Thomas, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 256–279. Springer.

Ruf, Jürgen and Kropf, Thomas (2003). Symbolic verification and analysis of discrete timed systems. *Journal on Formal Methods in System Design*, 23(1):67–108.

Ruf, Jürgen, Peranandam, Prakash M., Kropf, Thomas, and Rosenstiel, Wolfgang (2003). Bounded property checking with symbolic simulation. In *Forum on Specification and Design Languages*.

Ruf, Jürgen, Weiss, Roland J., Kropf, Thomas, and Rosenstiel, Wolfgang (2004). Modeling and formal verification of production automation systems. In Ehrig, Hartmut et al., editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 541–566. Springer Verlag.

VA Software Corporation and Open SystemC Initiative (2004). Open SystemC Initiative. `www.systemc.org`.

Vardi, Moshe Y. (2001). Branching vs. linear time: Final showdown. In *European Joint Conferences on Theory and Practice of Software (ETAPS 2001)*. Invited paper.

Chapter 17

AUTOMOTIVE SYSTEM DESIGN AND AUTOSAR

Georg Pelz[1], Peter Oehler[2], Eliane Fourgeau, Christoph Grimm[3]
[1]*Infineon Automotive & Industrial,* [2]*Continental Teves,* [3]*University of Frankfurt*

Abstract Automotive system design demands for new solutions for management of complexity and heterogeneity. Heterogeneity in automotive systems does not only cover different physical domains that are combined in a complex system. Heterogeneity in automotive systems also means optimized, proprietary solutions and interfaces. The fact that many tiers are involved, and many configurations have to be maintained is an increasing challenge. This chapter gives an overview of an approach to hide the heterogeneity behind a common and unique interface: AUTOSAR.

Keywords: Automotive System Design, AUTOSAR

1. Introduction

The design of automotive systems has changed dramatically. In the last years, electric and electronic (E/E) components have enriched mechanical and hydraulical components with additional features. Typical examples are the electronic brake systems ABS or ESP. In future systems, software components will provide an increasing share of functions and features in automotive systems.

The embedded software components are optimized for electronic hardware platforms and their physical environment. The close relation between hardware and software development is necessary in order to optimize performance and resource allocation [Salzmann et al.]. This leads to architectures that are characterized by proprietary solutions.

Both complexity and heterogeneity of automotive systems make their design to a cost intensive and time consuming challenge that demands for

- means for the management of complexity and heterogeneity,
- re-use of all kinds of intellectual property, and

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 293–305.

- scalability of existing solutions within and across product lines.

This chapter describes current problems of automotive system engineering and gives an overview of AUTOSAR, a development partnership for an open standard for automotive E/E architecture. Section 2 gives an overview of the design process of automotive systems, and introduces the tiers involved in the design process. Sections 3 and 4 motivate and give an overview of a new approach for hiding the complexity and heterogeneity of automotive systems behind a standardized, function-oriented interface: AUTOSAR.

2. Automotive System Design

2.1 Application Fields of Automotive Systems

The complete system of a car can be subdivided into quite a number of subsystems performing functions in the domains of safety, power train, body and navigation/telematics. These fields trigger much different requirements. For instance, the power train functions often are located close to the engine resulting in much higher ambient temperatures, which has to be taken into account when designing the respective subsystems. Also the weighting of performance vs. price varies from domain to domain. Moreover, the driving capabilities of the power electronics may be heavily different from application to application. In some cases, high-voltage CMOS processes may be sufficient to fabricate the respective chips, while for others fully-fledged BCD processes (bipolar, CMOS, DMOS) offer the best solution. Last but not least, in safety critical applications, the quality requirements may be even more strict, than e.g. for navigation.

Even after this short overview, it should be clear, that environment, requirements, basic technologies etc., are very different from application to application. Let's now take a closer look at the four above domains.

Safety. The safety features include adaptive restraint systems with occupant detection, belt pretensioners and - of course - quite a number of airbags, which depending on the crash scenario may or may not be fired. Furthermore, we have antilock braking system (ABS) and electronic stability program ESP and many other safety-relevant features.

Power Train. The power train part contains all motor control related features together with exhaust fume optimization, but also the brains of gearbox or automatic transmission.

Body. The applications of body electronics are for instance power-adjustable external mirrors, window lifters, HVAC (heating, ventilation, air-condition), centralized door locking, keyless entry and many more.

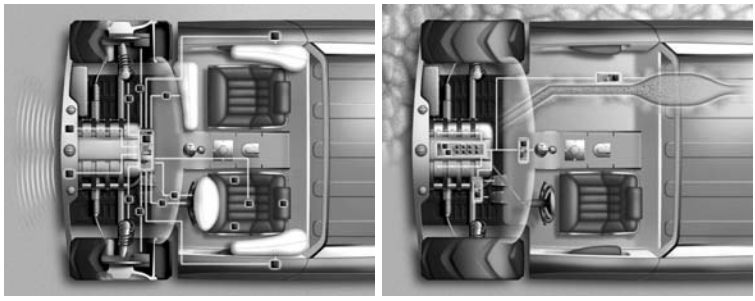


Figure 17.1. Electronics for automotive safety features (left) and power train features (right). (c) Infineon Technologies AG.

Navigation and Telematics. Navigation is about the combination of GPS position information with electronic maps. This can be refined with traffic information. Another feature might be an automatic rescue call in case of an accident.

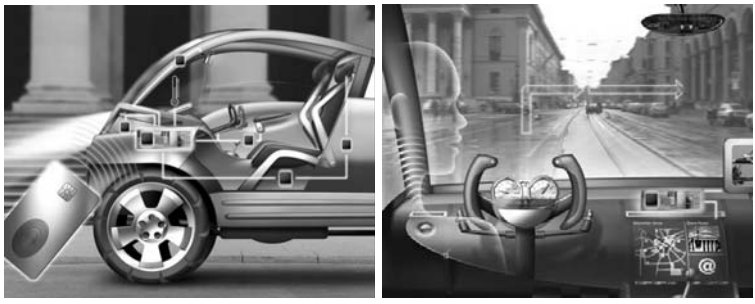


Figure 17.2. Electronics for automotive comfort functions of the car body (left) and Automotive navigation and telematics (right). (c) Infineon Technologies AG.

2.2 What is Difficult in Automotive Electronics

Heterogeneity. One of the most demanding challenges in automotive electronics comes with the plethora of physical and logical domains to be managed. In general, we have both analog and digital electronics. When digital electronics forms microcontrollers, also embedded software has to be taken care of. As soon as power comes into play, also thermal aspects have to be taken into account as well. Moreover, mechanics, hydraulics, pneumatics, magnetics and many other domains can also not be neglected. For all of these domains, different description formats are used and a legacy of single-domain software tools, e.g. simulators, is at hand. The problem is to reconcile descriptions and software tools to allow for complete system design.

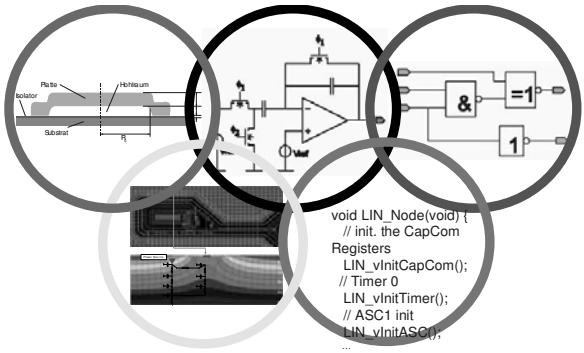


Figure 17.3. Almost Olympic challenge to reconcile the plethora of domains in automotive engineering.

Environment. Another big challenge in automotive is the rough environment in which the circuitry has to be operational. Temperature ranges are often from -40 °C to 150 °C ambient temperature. At the transistor junction, typically even higher temperatures are observed. Also the supply voltage in a car is 12 V nominal, which immediately kills modern CMOS interface circuitry. Even more, in some cases, the supply voltage may rise up to 60 V and higher.

Quality. The quality requirements are extremely tough. A typical quality requirement may be that only 500 ppm may fail within a given timeframe. Taking into account, that a modern car has some 50 applications like ABS or airbag, we get 10 ppm for each application. Each of these applications may have 300 components, like microcontrollers, power switches, passive devices but also motors, sensors, mechanical transmissions and many more. If we distribute the remaining 10 ppm to 300 components, which individually may cause the failure of the application (otherwise the components would not be in!), we finally end up with a requested quality of way below 1 ppm. This leads us to the common quality requirement for automotive components of zero ppm.

2.3 Roles and Cooperation in the Design Flow

The challenge of system design is not addressed by the car manufacturers alone. Usually, subsystems are developed by Tier-1 companies, e.g. Bosch or Continental Teves, which sometimes include Tier-2 companies, i.e. component suppliers, which design and manufacture microelectronic circuits (e.g. TI, Infineon), but also all the other components from electronics, mechanics, software and many other domains.

One major source of problems is the interfaces between the partners in the design flow as shown in Fig 17.4. Today, information is transported on the left wing through paper specs and on the right wing through samples. In the past, this worked up to some degree, but the current problems in complexity and heterogeneity cannot be solved in this way.

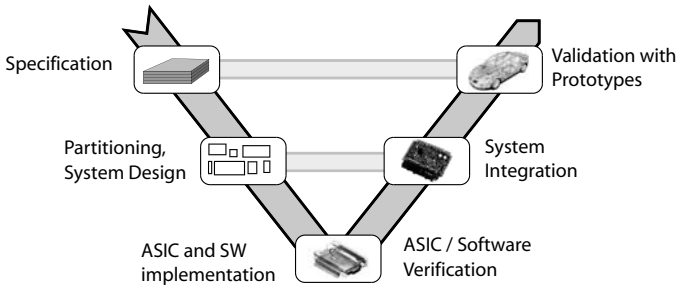


Figure 17.4. Design flow according to V-model

2.4 Design Methodology

Executable Specifications. To improve the interfaces on the left hand wing, paper specs are supplemented with behavioral models forming executable, i.e. simulatable, specs. These behavioral models can be simulated to illustrate the specified behavior. In this way, the partners can discuss on waveforms rather than about an operating point as in the paper specs. Also the ability to simulate helps a lot in validating the specification, while paper specs are at best checked manually. Moreover, it is much easier to make sure that an executable spec is complete and consistent, than it is for a paper spec.

Virtual Prototypes. On the right hand wing, the samples are supplemented with virtual prototypes, which again are formed out of behavioral models [Meise et al.]; [Pelz et al.]. They may be available even long before silicon. Moreover, simulations at least in some cases provide more insight than measurements, as models are completely controllable and observable. Measurements take much more effort and provide only limited access. Let's look at the example of a power switch observing a short circuit. The self-heating will quickly trigger the heat protection of the switch. This can be simulated in an electro-thermal way with limited effort, while meaningful measurements of the quick thermal transients are extremely difficult.

Partitioning and Design. An important issue of design is the system partitioning. System partitioning distributes tasks to resources and thereby determines costs, quality and performance. However, there are no tools available

which support the task of partitioning of complex, heterogeneous systems, so called multi-domain systems, in general. Tools that support the partitioning of mixed-signal systems (analog and digital electronics) are emerging in academic research, but they are still far away from industrial applicability at the moment. Nonetheless, what helps a lot is to use the behavioral models as a platform to assess partitioning- and design-alternatives. Even more, optimizations on parameters may even be carried out automatically.

If the scope is restricted to embedded hardware/software systems, partitioning here often is the decision about what to implement in hardware and what in embedded software. This task is the domain of classical hardware/software co-design. Tools are available but still far away from working in a fully automatic way. If we restrict the system on functions which are running on a microcontroller the focus is at embedded software. Then, partitioning means to decide which functions shall run on which microcontroller. This is still a challenging task because it is influenced by the problem to distribute software functions between different available microcontrollers and the communication mechanisms realized between them. However, the chances to reach satisfying results are high because of the homogeneity compared with the former tasks.

3. AUTOSAR: Aims and Objectives

Automotive systems like an electronic brake system (ABS, ESP, ...) are very heterogeneous. They consist of electric/electronics and a mechanic and/or hydraulic environment. Software components in automotive systems are developed by automotive OEMs, suppliers, and independent software companies. This leads to a number of problems:

- Current Automotive electric/electronic architectures are characterized by proprietary solutions that seldom permit the exchange of applications between automotive OEMs and their suppliers.

- Microcontrollers cannot be exchanged without need for adjustments in SW functions/applications.

The AUTOSAR development partnership. One approach to deal with the above mentioned problems is encapsulation of functions in well defined and standardized Interfaces between Applications and Programs (Application Program Interface, API). The encapsulation creates independence from communication technology at different levels. Standardized APIs could allow all tiers to exchange hardware and software components. This would permit a concentration on functions with competitive value, freeing valuable resources to focus on innovative new functionalities.

For the development of such a standardized *Aut*motive *O*pen *S*ystem *AR*chitecture, a three tier structure, proven in similar initiatives, has been formed in mid of 2003. Appropriate rights and duties are allocated to the various tiers:

- Core Partner (OEM & Tier 1 Supplier) are BMW Group, Bosch, Continental AG, DaimlerChrysler, Ford, Opel/GM, PSA Peugeot Citroën, Siemens VDO Automotive, Toyota and Volkswagen AG,
- 35 Current Premium Members (incl. Tool Manufacturers),
- 7 Current Associate Members

The Core Partner have organizational and technical control. Together with the Premium Partners they can make technical contributions and lead of or involve in Working groups, respectively. Premium Members have access to current information where Associate Members have access to finalized documents and can utilize the standards before release. Other support roles are the Development Member Agreement which allows to participate/cooperate in the Working groups free of charge. An Attendee Agreement allows close connections to other institutions.

Hence the AUTOSAR partnership is an industry wide alliance of OEM manufacturers and Tier suppliers working together to develop and establish a de-facto industry standard for automotive E/E architecture which will serve as a basic infrastructure for the management of functions within both future applications and standard software modules. The AUTOSAR milestone plan was released mid 2003 and foresees the completion of the test and verification phase in August 2006.

AUTOSAR objectives. In brief, the primary goals are the standardization of basic system functions and functional interfaces, the ability to integrate and transfer functions as well as to substantially improve software updates and upgrades over the vehicle lifetime. The AUTOSAR project goals will be met by specifying and standardizing the central architectural elements across functional domains, allowing industry competition to focus on implementation, while cooperating on standard. Figure 17.5 [Autosar] gives an overview of the AUTOSAR project objectives, and the involved functional domains.

Redundancy activation would mean, for example, that a system from one supplier could use/read sensors from other suppliers for failsafe reasons.

Topics for the integration of basic system functions and technologies include:

- Bus Technologies (CAN/LIN, FlexRay, ...)
- Operating Systems (OSEK, ...)
- Communication Layer (OSEK/COM, OSEK fault tolerance, ...)

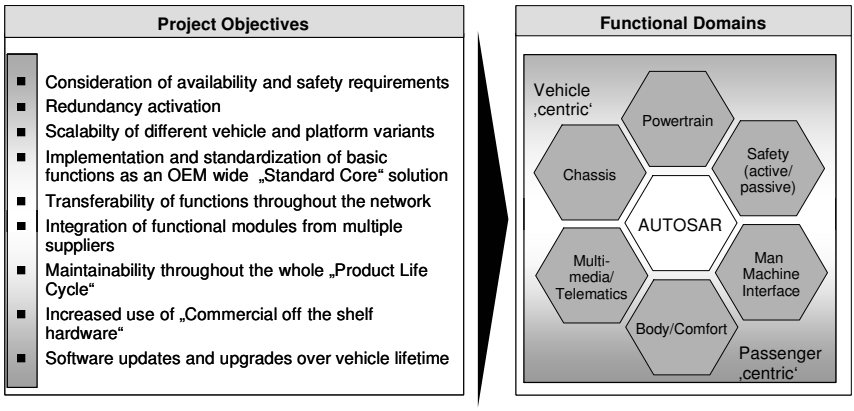


Figure 17.5. AUTOSAR high level project objectives [Autosar].

- HW Abstraction Layer
- Memory Services (NVRAM manager)
- Mode Management (ECU states, sleep mode, error manager, watchdog, . . .)
- Middleware/Interfaces - APIs
- Standard Library Functions (CRC checksum, cast operations, mathematical functions, . . .)

All this leads to the main focus of AUTOSAR: Cooperate on standards, compete on implementation.

Transferability and exchangeability. In the development partnership an architecture shall be developed which fulfills the system requirements of the functional domains shown in figure 17.5. AUTOSAR will serve as a platform from which future vehicle applications will derive and upon which they will be implemented. The availability of a standardized, functional view of the functional domains enables the transferability of functions and/or features between the domains. For example, a brake system could use sensor data available from another domain. Figure 17.6 [Autosar] gives an overview of the intended transferability of functions with AUTOSAR.

AUTOSAR's vision is an improved complexity management of highly integrated E/E architectures through an increased re-use and exchangeability of SW modules between OEMs and suppliers. This means that, for example, software modules developed based on an ECU from supplier A should as well run

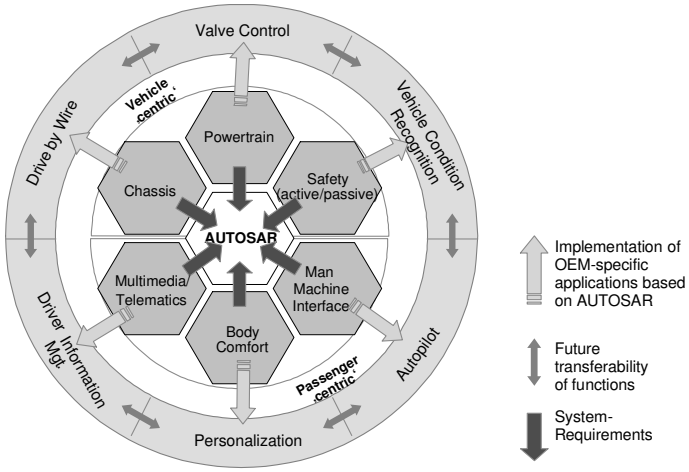


Figure 17.6. Future transferability of functions with AUTOSAR [Autosar].

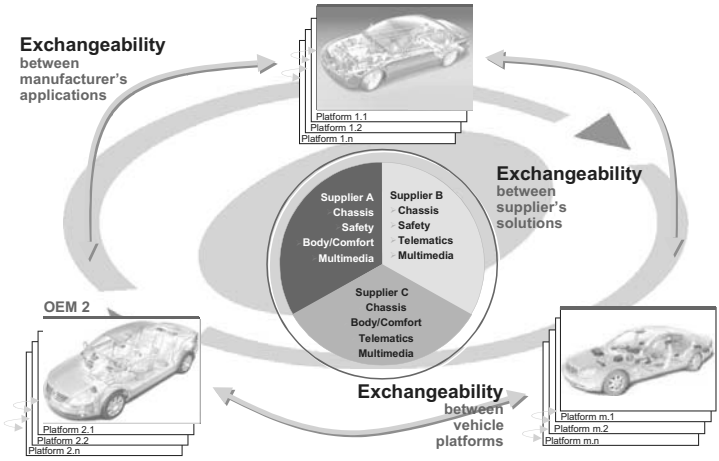


Figure 17.7. Exchangeability of functions between OEM and suppliers [Autosar].

with an ECU from supplier B. Figure 17.7 gives an overview of the intended exchangeability of functions between OEM and suppliers.

This transferability of application layer software components is enabled through the AUTOSAR Run Time Environment (RTE, figure 17.8). The AUTOSAR runtime environment is abstracted to a Virtual Function Bus (VFB) that acts as a communication center for inter and intra ECU information exchange. All communications run through the AUTOSAR VFB, which provides a communication abstraction to software components attached to it by provid-

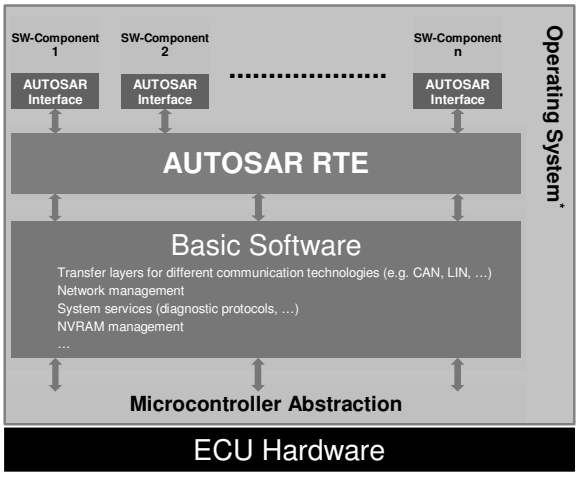


Figure 17.8. Schematic view of AUTOSAR software architecture [Autosar].

ing the same interface and services whether inter ECU communications are used (such as CAN, LIN, FlexRay, MOST, ...) or intra ECU communication channels. As the communication requirements of the software components running on top of the RTE are application dependent, the RTE needs to be tailored to these communication requirements.

Methodology. The desired transferability and exchangablility also have an impact on modeling and design methodology. In AUTOSAR, an E/E system consists of three main parts:

- software components,
- system constraints (e.g. bus systems / attributes like data rates; clustering/mapping of software components), and
- ECU resources (physical and electronic attributes).

The AUTOSAR methodology combines these three main parts in a process of five steps. In a first step software components, system constraints and ECU resources are specified. In a second step the SW components are distributed or partitioned to the available ECUs considering the system constraints. In a third step the available information is separated into software tasks for each single ECU. In step 4, compiler and linker are applied to get binary code. Finally, system integration permits validation and improvements of the system in step 5.

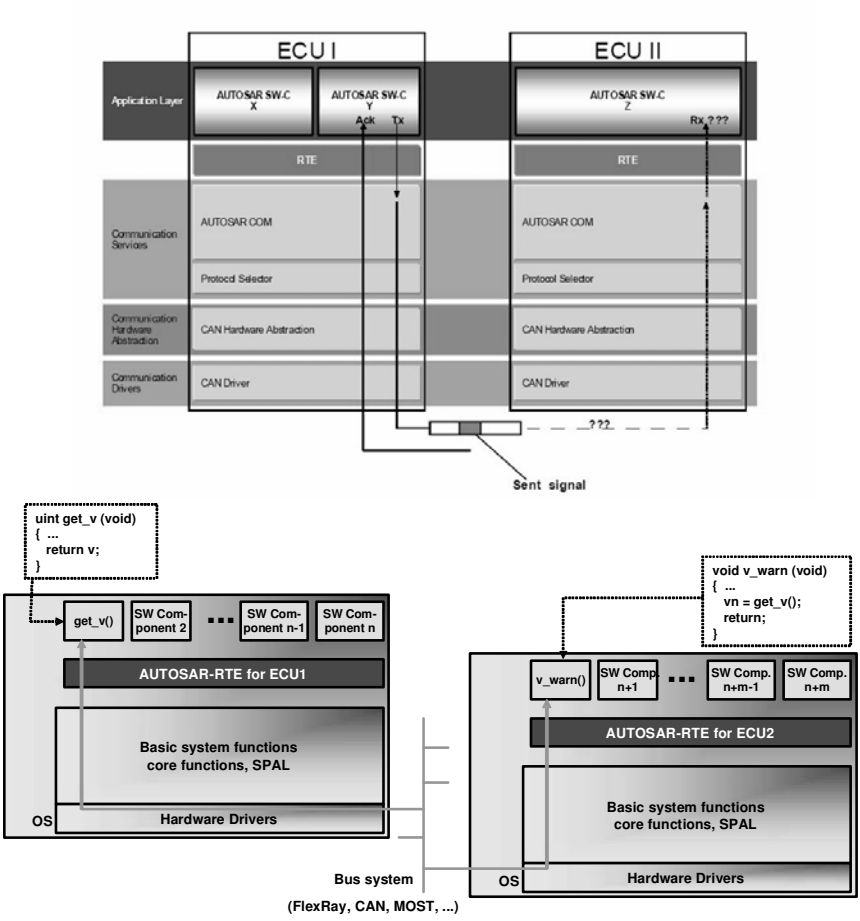


Figure 17.9. Implementation on 2 ECUs (upper image not from AUTOSAR). Implementation of functions independent on distribution on different ECUs as communication will be done via ECU individual AUTOSAR RTE exclusively.

4. Status Quo of AUTOSAR

The introduction of the AUTOSAR standard is striving to place the future of E/E development in automotive industry on an industry wide accepted and stable basis. This will be a key element in order to cope with the functional and legal requirements in next generation vehicle architectures. It will also be instrumental in securing market attractiveness and opening new and different business opportunities for OEMs and their suppliers alike, pushed by increasingly demanding legal and customer requirements, which are often conflicting:

- Legal enforcement (environmental aspects, safety requirements . . .)

- Passenger convenience and service requirements from the comfort and entertainment functional domains
- Driver assistance and dynamic drive aspects (navigation in high density traffic surroundings, detection and suppression of critical dynamic vehicle states ...)

Unlike past attempts where the automotive industry reacted in response to major technological advances, now a technological breakthrough has been started in order to keep pace with these requirements and enable industry actors to face, well armed, their challenging future.

The development of the AUTOSAR standard is organized into a number of executive technical work packages; any individual work package is the responsibility of an associated working group. They pertain to the following steps:

- Specification of the mechanisms and interfaces of the virtual functional bus
- System generation including specification of the descriptions format and contents and the associated set of tools.
- AUTOSAR ECU configuration delivering configuration files of the particular run time environment modules
- ECU software generation: the process of generating software executables out of the ECU configuration files.
- Test and integration including a prototype implementation supporting representative applications, which will be developed alongside standard definition.
- Data description delivering the formulation of unified functional interfaces of all vehicle domains: Body/comfort, Power train, Chassis/driver dynamics, Safety, Telematics/multimedia, Man-machine interface
- Enabling of AUTOSAR exploitation: definition of conformance test and licensing procedures, version control management and continuous maintenance of the AUTOSAR standard.

5. Outlook

Automotive system design will face a number of new techniques in the next years. Considering design methodology, executable specification and virtual prototyping are going to find their way into industrial application. However, this requires new techniques for modeling and simulation, such as SystemC-AMS [Vachoux et al.].

System partitioning is still a demanding task, which will still require interaction with a designer. A major breakthrough might come with standardized APIs and/or middleware such as AUTOSAR. This allows single components an access to features or functions provided by other components in the car and enables both the development of new features and a reduction of costs.

References

The AUTOSAR development partnership. Web Page. `www.autosar.org`, Dec. 2004.

Christian Meise and Christoph Grimm. A SystemC Based Case Study of a Sensor Application using the BeCom Modeling Methodology for Virtual Prototyping, In *17th Symposium on Integrated Circuits and Systems Design (SBCCI '04)*, Porto de Galinhas, Pernambuco, Brazil, IEEE Press 2004.

Georg Pelz. Mechatronic Systems: Modelling and Simulation with HDLs, John Wiley & Sons, 2003.

Christian Salzmann and Thomas Stauner. Automotive Software Engineering. In *Languages for System Specification*, Kluwer Academic Publishers, June 2004.

Alain Vachoux, Christoph Grimm and Karsten Einwich. Extending SystemC to support Mixed Discrete-Continuous System Modeling and Simulation (invited paper), In *International Symposion on Circuits and Systems 2005 (ISCAS '05)*, Kobe, Japan, IEEE Press 2005.