APPLICATIONS OF SPECIFICATION AND DESIGN LANGUAGES FOR SOCS

Applications of Specification and Design Languages for SoCs

Selected papers from FDL 2005

Edited by

A. VACHOUX
Ecole Polytechnique Fédérale de Lausanne,
Lausanne, Switzerland

Springer

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

ISBN-10 1-4020-4997-8 (HB)
ISBN-13 978-1-4020-4997-2 (HB)
ISBN-10 1-4020-4998-6 (e-book)
ISBN-13 978-1-4020-4998-9 (e-book)

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

www.springer.com

Printed on acid-free paper

Contents

List of Figures

List of Tables

List of Listings

Preface

The Forum on specification and Design Languages (FDL) is the premier European forum to exchange experiences and learn about new trends in the application of languages and models for the specification and modeling of electronic systems. FDL'05 was organized around four thematic areas that cover essential aspects of system-level design methods and tools: "C/C++-Based System Design" (chaired by Frank Oppenheimer, OFFIS, Germany), "Analog, Mixed-Signal, and Heterogeneous System Design" (chaired by Christoph Grimm, University of Hannover, Germany), "UML-Based System Specification and Design" (chaired by Piet van der Putten, TU Eindhoven, The Netherlands), and "Specification, Design, and Verification Methods" (chaired by Alain Vachoux, EPFL, Switzerland). This book includes a collection of outstanding contributions to FDL'05 that have been carefully selected by the thematic area Chairs and thoroughly revised by the authors. The book has 17 chapters grouped in four parts. Each part groups chapters related to one thematic area and is introduced by its respective FDL'05 Chair:

- Part I, "Specification, Design, and Verification Methods," includes two chapters covering system design issues of growing importance, namely assertion-based design and network-on-chip (NoC) design flow.
- Part II, "C/C++-Based System Design," includes five chapters mostly addressing issues related to the development and the use of SystemC for hardware/software system-level design.
- Part III, "Analog, Mixed-Signal, and Heterogeneous System Design," includes five chapters discussing how the design of mixed-signal/mixed-technology systems may be handled at system level.
- Part IV, "UML-Based System Specification and Design," concludes the book with five chapters exploring modeling methodologies that can map abstract models of complex systems onto efficient implementations.

The book is providing an excellent coverage of recent achievements in the use of languages and models for the specification and the design of systems-on-chip (SoCs). It also highlights the diversity of the issues that have to be tackled

with in today's system designs and the creativity of researchers to develop efficient solutions. Last, but not least, the book shows the importance of the FDL event as the place to discuss issues related to electronic system design.

On a final note, I would like to warmly thank Torsten Mähne for his excellent work in helping me editing the book. Torsten efficiently managed all the subtle LaTeX issues that arose and highly contributed to make all chapters in the book consistent.

Alain Vachoux
FDL'05 General Chair
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
March 2006

I

Specification, Design, and Verification Methods

Introduction

This thematic area of FDL'05 deals with specification-driven designs, formal verification techniques, mixed formal and simulation-based verification techniques, formal languages (B, CTL, Z, temporal logic, etc.), synchronous languages (Esterel, etc.), modeling concepts (e.g., StateCharts, Petri Nets, finite state machines (FSMs), dataflow models, etc.), and models of computation.

In this part, two contributions in this thematic area addressing two important aspects in system design have been selected. The first aspect is assertion-based design methods. Using assertions in design models provides means to write specifications in a formal way that can be unambiguously understood and verified by designers and tools. They also provide a strong link between design and verification activities early in the design flow. One emerging assertion language is the Property Specification Language (PSL). PSL is an IEEE standard that has been designed to unify static (formal) and dynamic (simulation-based) verification and that is currently supported by many commercial electronic design automation (EDA) tools. Chapter 1 by Dominique Borrione, Miao Liu, Pierre Ostier, and Laurent Fesquet, entitled "PSL-Based Online Monitoring of Digital System," presents an original method for generating hardware assertion monitors. Such monitors capture the occurrence of events specified by logical and temporal properties originally written as PSL assertions. The chapter illustrates the approach on an experimental field programmable gate-array (FPGA)-based platform that includes a Nios-embedded processor.

The second aspect deals with the design of systems-on-chip (SOC) architecture and the mapping onto a network-on-chip (NoC) architecture. NoC is an emerging paradigm that aims at providing efficient on-chip communication services capable of supporting large quantities of heterogeneous processing components (e.g., processor cores digital signal processings (DSPs), FPGAs/ application-specific integrated circuits (ASICs), and memories). Such SOCs are one possible solution to sustain the ever increasing complexity of applications and are enabled by the constant improvements in manufacturing technologies. Chapter 2 by Zhonghai Lu, Ingo Sander, and Axel Jantsch, entitled "Refining Synchronous Communication onto Network-on-Chip Best-Effort

Services," presents a novel approach to refine a system model specified with perfectly synchronous communication onto an NoC best-effort communication service. The approach starts from a formal synchronous model of the communication and ends with an NoC architecture providing best-effort communication service class. The use of perfectly synchronous models allows to cleanly separate computation from communication and to better formalize and validate system specifications. The proposed refinement procedure then takes care of deriving an efficient implementation of the communication onto the NoC architecture, for which the perfect synchrony assumption does not hold anymore.

Alain Vachoux
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland, March 2006

Chapter 1

PSL-Based Online Monitoring of Digital Systems

Dominique Borrione, Miao Liu, Pierre Ostier, and Laurent Fesquet
TIMA
46 Avenue Félix Viallet
38031 Grenoble cedex
France

Abstract We present an original method for generating monitors that capture the occurrence of events, specified by logical and temporal properties under the form of assertions in declarative form, written in the PSL standard. The method includes a library of primitive digital components and a technique to interconnect them, resulting in a synthesizable digital module that can be properly connected to a digital system under verification, or to a set of input signals under scrutiny. The complexity of the generation is proportional to the size of the PSL expression. A prototype emulation system has been implemented.

Keywords: property; monitor; PSL.

1. Introduction

Today's increasing design complexity requires innovative methods for verification and debug. With verification consuming up to 70% of the design cycle, assertion-based design (Foster et al., 2003) is viewed as one key method for improving productivity. An assertion is a design property that is declared to be true and should be evaluated by one or more techniques among simulation, emulation, or formal verification. The introduction of new standard languages such as Property Specification Language (PSL) or SystemVerilog has made assertions more easy to write and very powerful. An assertion can also be seen as a high-level functional specification for a circuit intended for monitoring of events over time.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 5–22.

We developed an original method for generating hardware that monitors signals whose behavior is specified by logical and temporal properties under the form of assertions in declarative form. In this chapter, we shall use Accellera's PSL standard (Accellera, 2003, 2004) and assume the reader to be familiar with its basic concepts. The method is founded on a library of primitive digital components and a technique to interconnect them, resulting in a digital module that can be properly connected to the signals of interest. Monitoring can be initialized and started independently from the system under scrutiny; it runs concurrently with the system under verification and notifies its environment when the property checking is terminated with a true or false value or whether the property is still being evaluated, possibly with a transient false value. Properties over finite and infinite state sequences over time are covered by the method. Monitors under this method may be used for design verification by simulation. But their primary use is online checking during either hardware emulation for debug or normal system operation for safety-critical property checking.

1.1 State of the Art

PSL (Accellera, 2003, 2004) is a standard specification language proposed by Accellera, and further standardized by IEEE. It is based upon the Sugar 2.0 property specification language (Formal Methods Group, 2000), and is an extension of the temporal logics: linear temporal logic (LTL) and computation tree logic (CTL). PSL is used to describe properties that are required to hold in a device under verification (DUV). Owing to its formal denotational semantics, PSL provides a means to write specifications that are both easy to read and mathematically precise. Special care has been devoted to the semantic definition. Gordon performed a "deep embedding" of PSL in the High Order Language (HOL) proof assistant. He used this mechanized system to demonstrate theorems about the semantics and to derive correct-by-construction mathematical observers for PSL properties (Gordon et al., 2003). A detailed analysis revealed some inconsistencies in the interpretation of a special class of regular expressions; to help solve this problem, Claessen and Martensson proposed an operational semantic definition (Claessen and Martensson, 2004) to guide the semantic definition with the structure of the PSL formula.

PSL is designed to unify static verification (e.g., model checking) and dynamic verification (e.g., simulation). All properties expressed in PSL can be verified by formal verification methods that consider all possible states of a DUV, traversing them in a branching or parallel behavior. In contrast, some of these properties cannot be easily evaluated in simulation or execution of the DUV, because time advances monotonically along a single path. So a *simple subset* of PSL is defined that conforms to the notion of *monotonic advancement*

of time, left to right through the property, which in turn ensures that properties within the subset can be evaluated along the execution of the DUV.

Many products support PSL. Long lists of companies and software names may be found, together with their websites (Cohen et al., 2004; IBM, 2005). Without any attempt to exhaustivity, the main tools may be partitioned into:

1. **Formal verification of circuit properties by model checking:** The property, written in PSL, is evaluated over the reachable states of the finite state machine (FSM) extracted from the design. Further to RuleBase from IBM (Beer et al., 1996), let us mention Verix from Real Intent, Safelogic Verifier, imPROVE-HDL from TransEDA, Verity-Check from Veritable, Archer-SF from 0-In Design Automation, etc.
2. **Automatic generation of test bench:** Tools like Designer from HDL and Specman Elite from Verisity generate full-coverage test bench for PSL assertions automatically.
3. **Simulators supporting standard hardware description languages (HDL's) and PSL assertions at the same time:** During simulation, the evaluation of properties is executed on the fly and the evaluation results are shown as waveforms together with other signals of the design. Examples of such tools include Cadence Incisive or Modelsim from Mentor Graphics.
4. **Plug-in products working together with leading HDL simulators add the assertion support to these simulators:** Examples are Auriga from FTL Systems, Safelogic Monitor, VN-Property from TransEDA, and Archer-CDV from 0-In Design Automation. Novas Software developed a series of products as an assertion-based debug system. Esterel Technologies has integrated assertion checking into Esterel Studio.
5. **Generation of property monitors:** IBM Corporation has developed a tool named FoCs (Abardanel et al., 2000), which generates HDL checkers for PSL properties that can subsequently be linked to the design and loaded into a verification environment. By this method, assertions can be verified by simulation as well as formal methods without change. Summit has combined FoCs into its C/C++/SystemC simulation tool Visual Elite, in which Sugar assertions are transformed into efficient assertion-checking code.

1.2 PSL as a Design Language

From the information and publications available to the authors, PSL properties are largely seen as an expression of the intended behavior of a design expressed in another language. PSL properties are what you want to verify on your design, rather than synthesizing the specification of an interesting observer. Yet,

in safety critical systems, it may be of utmost importance to design a system together with one or more monitors that can detect the occurrence of possibly complex sequences of events, particularly when such events are the result of transient errors or when such events are rare and the size of the design defeats formal verification methods. In that case, synthesizing a hardware monitor that checks a PSL property becomes a design activity.

IBM's FoCs partially address this objective, by producing a source HDL process from a PSL statement. But the presence of error-reporting statements in the resulting code is clearly verification-oriented, and the user is limited to a very strict subset of primitive PSL. Moreover, the lack of information about the principles for constructing the monitors prevents incremental enhancements. The situation with other products is even worse; no information about the generated property monitors is made visible to the user. The motivation for our research was thus to find a method that would allow the synthesis of monitors in an incremental way, in order to easily adjust to any semantic change between PSL language versions, and that would remain of a tractable complexity.

Many works have been published in the context of model checking of LTL formulae. A standard technique translates the formula into an automaton that recognizes all the acceptable sequences of values for the operand signals in the formula (Daniele et al., 1999; Gastin and Oddoux, 2001). The problem is that the resulting automaton is nondeterministic, which is inappropriate for a hardware implementation, and the transformation into a deterministic one is exponential in the number of nondeterministic decision states (Sebastiani and Tonetta, 2003). Considering that a syntactically simple PSL formula can easily expand into an LTL formula with a large number of temporal operators, the automata-theoretic approaches appear too inefficient.

In contrast, we are not interested in generating an automaton, but rather a sequential circuit whose state space is equivalent to that of an accepting automaton for the formula. Taking a hardware design approach guided by the syntactic structure of the PSL formula, it is possible to directly build a monitor as a structural interconnection of primitive components. More precisely, we defined a library of primitive components, one for each PSL operator, whose behavior is based on the operator semantics. Both the "weak" and the "strong" version of the PSL operators are supported. For a PSL formula, all involved operator monitors are combined together with dedicated rules to build the monitor for that formula. Monitors generated by our method have the same structure as the PSL formulae, and are thus easy to understand, to debug, and to reuse. This approach is flexible and easy to extend, should PSL be extended in a future revision of the standard. The validity of the method depends on the correctness of the individual library elements and of the composition method; this point will be discussed in Section 3.

This chapter describes the monitors for the "foundation language" operators. Monitors for the "sequential extended regular expressions" (SEREs) are described in (Gascard, 2005).

2. Monitor Construction: Principles

We provide a mechanism to automatically produce a hardware monitor that checks a property specified in any (linear temporal) PSL that ensures monotonic advancement of time, left to right through the property. The "simple subset" of PSL has this property, although it may be extended. In the following, we use the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) flavor of PSL and components are described in VHDL, but the principles apply to other syntax as well. In the text and the figures, 0 and 1 are used both for bits `'0'` and `'1,'` and for Booleans `false` and `true`.

2.1 Property Satisfaction

PSL defines four levels of satisfaction of a property on a finite execution path of the DUV, which we recall as follows:

Holds strongly. No bad states have been seen. All future obligations have been met. The property will hold on any extension of the path.

Holds (but does not hold strongly). No bad states have been seen. All future obligations have been met. The property may or may not hold on any given extension of the path.

Pending. No bad states have been seen. Future obligations have not been met. The property may or may not hold on any given extension of the path.

Fail. A bad state has been seen. Future obligations may or may not have been met. The property will not hold on any extension of the path.

The key idea to the monitor construction is based on this observation. In practice, it is sufficient to obtain the answer about a property satisfaction one or two clock cycles after this answer is known in theory. This is quite different from the results of model checking, which can tell that a formula is true/false in a state even when that formula references future values.

Example: Assume signals `A`, `B`, `C` take values as shown in Figure 1.1 (vertical dotted lines stand for the default clock edges), and `PR` is defined as:

```
PSL property PR is A implies (next[2](B until C));
```

`PR` is true in the initial state of the system (`A = 1`, `B = 0`, `C = 0`), but the answer depends on the true value of (`B until C`) two cycles later, which is known only when `C` takes value 1.

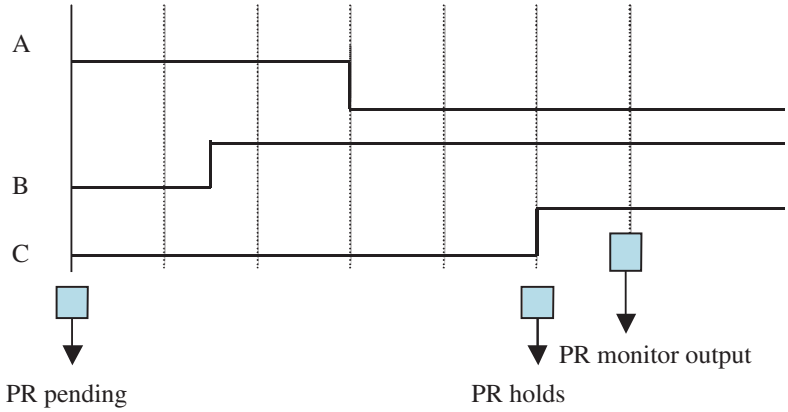


Fig. 1.1 Property status and monitor output

The monitor outputs have the following significance, some of which are for the strong version of operators only. Internal monitors, and in particular primitive components, have the first two outputs only.

Checking: indicates when output valid is effective.

Valid: provides the evaluation result (1 for absence of error, 0 for error).

Pending: takes value 0 if the obligations have been met, 1 otherwise.

Fail: takes value 1 if bad states have happened, 0 otherwise.

Outputs `checking` and `valid` are generated by the last monitor, and they can be used to generate appropriate actions when a valuation of the property has been obtained. The combination of outputs `pending` and `fail` indicate the evaluation result of the property (00 for hold, 01 or 11 for fail, 10 for pending).

Example: Let `P1` be the following invariant property (it must always hold), synchronized on the rising edge of clock `clk`. `P1` states that one clock cycle after receiving a request on signal `req`, a `done` signal must be eventually received within one or two cycles and remain 1 until `grant` is received. Figure 1.2 displays the monitor generated for `P1` defined as:

```
property P1 is always (req->next((next_e 1:2(done))
                      until grant)) @rising_edge(clk).
```

2.2 Library of Primitive Components

A primitive monitor is defined for each operator of the "foundation language" of PSL. Table 1.1 lists all the components of our library of primitive monitors: column "Monitor Name" lists their names for further reference in this

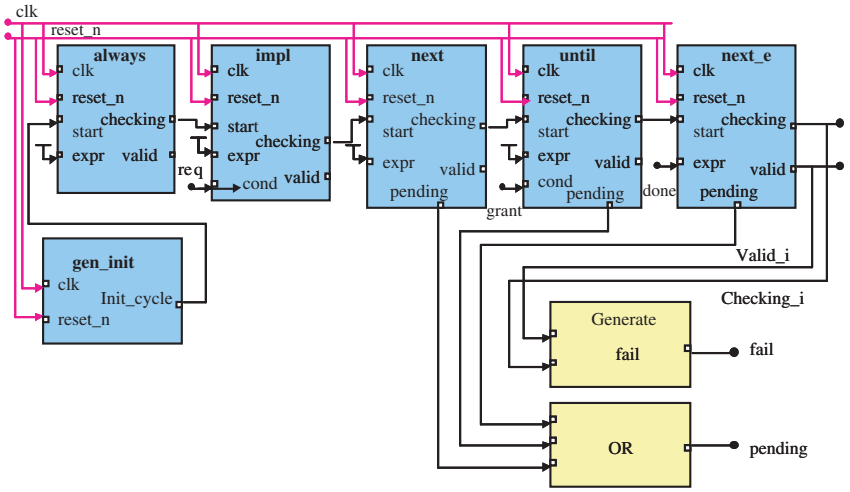


Fig. 1.2 Property monitor for P1

Table 1.1 Components in the library

Monitor name	*Corresponding foundation language operator*
`gen_init`	Initial state generator
`mnt_always`	**`always`** `(FL_Property)`
`mnt_never`	**`never`** `(Boolean)`
`mnt_next`	**`next`** `[Number] (FL_Property)`
`mnt_next_a`	**`next_a`** `[finite_Range] (FL_Property)`
`mnt_next_e`	**`next_e`** `[finite_Range] (Boolean)`
`mnt_next_event`	**`next_event`** `(Boolean) [positive_Number] (FL_Property)`
`mnt_next_event_a`	**`next_event_a`** `(Boolean) [finite_postive_Range] (FL_Property)`
`mnt_next_event_e`	**`next_event_e`** `(Boolean) [finite_postive_Range] (Boolean)`
`mnt_abort`	`FL_Property` **`abort`** `Boolean`
`mnt_before`	`Boolean` **`before`** `Boolean, Boolean` **`before_`** `Boolean`
`mnt_until`	`FL_Property` **`until`** `Boolean, FL_Property` **`until_`** `Boolean`
`mnt_impl`	`Boolean -> FL_Property`
`mnt_iff`	`Boolean <-> Boolean`
`mnt_and`	`FL_Property` **`and_op`** `FL_Property`
`mnt_or`	`Boolean` **`or_op`** `FL_Property`
`mnt_eventually`	**`eventually!`** `Boolean`

document, and column “Corresponding Foundation Language Operator” lists the PSL operator or set of operators that each component implements. For all monitors that have a strong and a weak version, both versions are supported (not repeated in the table).

Table 1.2 Parameters

Name	*Type*	*Explanation*
OP_TYPE	INTEGER	Type of operator: 0: weak type operator or weak exclusive operator 1: strong type operator or strong exclusive operator 2: weak inclusive operator 3: strong inclusive operator
NUM_CLK	INTEGER	Number of clock cycles
LOW_CLK	INTEGER	Lower bound of clock range
HIGH_CLK	INTEGER	Higher bound of clock range
NUM_COND	INTEGER	Number of condition occurrences
LOW_COND	INTEGER	Lower bound of the range of condition occurrences
HIGH_COND	INTEGER	Higher bound of the range of condition occurrences

The primitive monitors are defined as generic modules; they must be instantiated with parameters indicating the strength and numeric attributes of the particular operator in use (see Table 1.2). A particular operator variation belonging to a single operator family, and implemented within one monitor, can be chosen by the parameter OP_TYPE. For example, the family of operators next includes next, next!, next[num], next![num]; the family of operators until includes until, until!, until_ and until_!.

In addition, two primitive monitors have different interfaces and connection rules; they correspond to operators abort and operator and when its operands are temporal expressions (an extension to the simple subset; Liu, 2004). For reasons of space, these monitors are not further discussed.

2.3 Structure of a Primitive Monitor

The main structure of a primitive monitor is shown in Figure 1.3. Two main blocks can be identified:

- The checking window block generates the temporal window for the evaluation of the operands, and it sets an internal check enabling check_en signal, based on the evaluation requirement (start input signal) and the semantics of the operator. For the operators that include an overlap of evaluation windows, due to multiple activations of the start input signal, a shift register checkbit_reg is included. Defined as a BIT_VECTOR(HIGH_CLK downto 0), checkbit_reg shifts from the lowest bit to the highest bit at each time unit/clock cycle.

- The evaluation block executes the checking of the operands, when the checking-enable signal is '1.' When reset is active, the monitor stays in its reset state. Otherwise, when check_en is active, the operand is

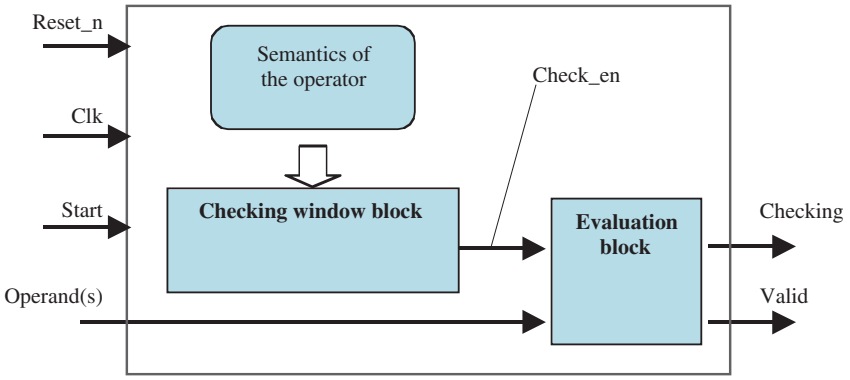


Fig. 1.3 Structure of a primitive monitor

checked and output `valid` represents the result. When `check_en` is inactive, execution is stopped and output `valid` stays in its default value "`1.`"

2.4 Construction of Complex Monitors

To generate the monitors for complex properties, primitive single-operator monitors are interconnected to construct a complex property monitor. The method is based on the syntax tree of the property. A node in the tree represents a PSL operator, a leaf represents a basic operand (signal or constant value), and the edges connect an operator with its operands. Some operators have two operands, whereas some have only one. The root node of a property tree is the operator with lowest precedence, i.e., the operator executed last Figure 1.4.

Initial-State Generation. Normally, a PSL property refers to the initial state of the design, i.e., the state at power on. If the property needs to be evaluated at states other than the initial state, it should be enclosed in some temporal operator, such as `always`, `never`, `until`. We use a module `gen_init` to generate the initial-state signal `init_cycle`, which is active one cycle immediately after power up reset. This signal is used as the evaluation requirement for the whole property, and it is fed into input `start` of the root monitor.

Operators. For each node in the tree structure of a property, a corresponding operator monitor is needed. All components available in the PSL monitor library and the relationship with PSL operators are listed in Table 1.1.

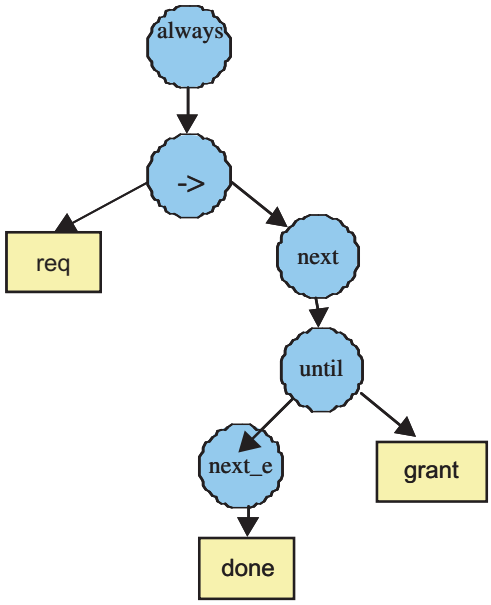


Fig. 1.4 Tree structure of PSL property P1: always (req -> next((next_e[1:2](done)) until grant)) @ rising_edge(clk)

For usual Boolean expressions written with Boolean operators and Boolean variables, no special processing is needed. They are kept unchanged within the monitor and synthesized into usual gate networks.

Example: `(not a), (a or b), not (a and b)`, etc.

Operand Properties of Boolean and Temporal Operators. If the operand of an operator is a Boolean expression, just connect it to the corresponding operand input of the operator monitor. If the operand is a formula composed of other PSL temporal operators, connect value '1' to the corresponding operand input of the operator monitor in order to disable the evaluation function within the monitor. This is Because the evaluation of other operands depends on other monitors and cannot be obtained within this single monitor (Figure 1.3).

Example: `assert always (req -> next ack)`

The second operand of operator -> is a formula (`next ack`) with a temporal operator. So, for the monitor of operator ->, the second operand input `expr` is connected to '1' to disable the evaluation function within the monitor. The connection of monitor -> is as follows: `clk`, `reset_n`, `req` are primary inputs; input `start` is connected to the output `checking` of the monitor `always`; and

the outputs `checking` and `valid` are connected to the corresponding primary outputs of the overall monitor.

Connection of Consecutive Operators. For two operators `N1` and `N2` such that `N1` is the father node of `N2` in the tree structure (`N2` is an operand of `N1`), the two monitors are connected in the following way (see Figure 1.2 for an illustration):

- Output `checking` from monitor `N1` is fed to input `start` of monitor `N2`.
- Output `valid` from monitor `N1` is useless, and left unconnected.
- The clock and reset signals are shared by the two monitors.

3. Validation

As a first validation step, the results of our prototype monitor synthesizer have been compared (Liu, 2004) with those provided by FoCs, version 2.02. The comparison method is shown in Figure 1.5. We used RuleBase to check the equality between the monitors generated by our compiler and by FoCs.

3.1 Functional Comparison

First, simple properties were written to verify each library component individually. Because model checking cannot verify parameterized modules, all parameters were fixed with (small) constant values. All the weak versions of the monitors in our library were shown equivalent to the process generated by FoCs, on the properties listed in Table 1.3, except the monitor for operator `abort`, which is not supported by FoCs.

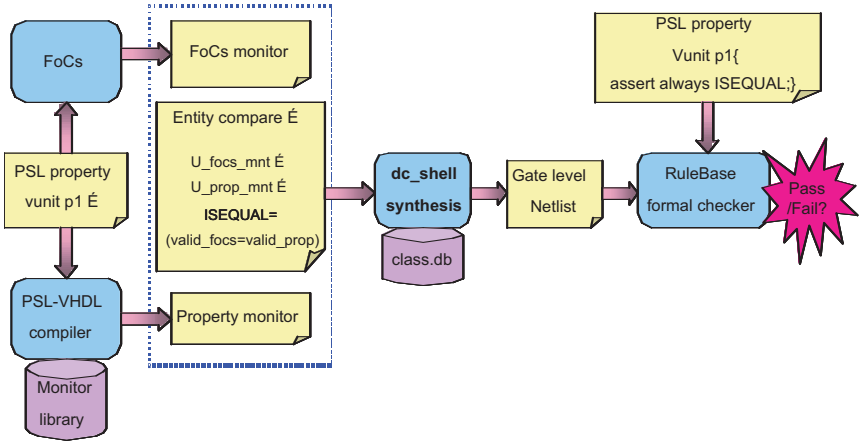


Fig. 1.5 Comparison flow

Table 1.3 List of properties for library validation

PSL property
`assert always (Req);`
`assert never (a);`
`assert always (a -> next (b));`
`assert always (a -> next [4] (b));`
`assert always (a -> (next_a [1 to 3] (b)));`
`assert always (a -> (next_e [1 to 2] (b)));`
`assert always (a -> next_event(b) (c));`
`assert always (a -> next_event(b) [4] (c));`
`assert always (a -> next_event_a(b)[1 to 3] (c));`
`assert (always a) abort b;`
`assert always (a -> (b before c));`
`assert always (a -> (b before_ c));`
`assert always (a -> (b until c));`
`assert always (a <-> b);`
`assert always ((a -> (b until c)) and (d -> next(e)));`
`assert always ((a) or (next_a [1 to 3] (b)));`
`assert always (a->next_event_e(b) [1 to 6] (c));`

Then we wrote compound PSL properties, involving several nested FL temporal operators, and performed the comparison of the monitor result with the process generated by FoCs. On all cases when we limited our properties to the subset supported by FoCs, we proved the functional equality. Examples of such properties are given in Table 1.4. This experiment, although not a thorough verification, gave us some initial confidence in our complex monitor construction method.

Note that some monitors could not be checked in this way. For instance, in FoCs, operator `until_` has the limitation that both operands should be Boolean. So we could not compare our monitor for a property such as: `always ((next (a)) until_ b);`

3.2 Area Comparison

We used Synopsys "design compiler" to synthesize the equivalent monitors generated by our PSL-VHDL compiler and by FoCs V2.02, and compared the area efficiency of the two results. The circuit area is computed on the basis of Synopsys generic library `class.db`: the unit is the area of the ND2 (two-input inverted and gate) cell. The results are shown in Table 1.5.

It is interesting to see that for simple properties with small-value parameters, the area of FoCs monitors is smaller than our monitors; but for complicated properties with larger parameters, such as `Prop1-4`, the area of FoCs monitors is much larger than ours (see boldface numbers).

Table 1.4 List of complex properties with nested temporal operators

Name	*PSL property*
Prop1	always (a-> next(next_a[2 to 10] (next_event(b) [10] ((next_e [1 to 5] (d)) until (c)))));
Prop2	always (a->(next_event_a(b) [1 to 4] (next((d before e) until (c)))));
Prop3	always (a->(next_event(c) ((next_event_e(d)[2 to 5](e)) until (b))));
Prop4	(always (a->next(next [10] (next_event(b) ((next_e[1 to 5](d)) until (c)))))) and (always (e->(next_event_a(ff)[1 to 4] (next((gg before h) until (i))))));
Prop5	always ((a-> next(next[10] (next_event(b) ((next_e[1 to 5](d)) until (c))))) or e);

Table 1.5 Area comparison results

PSL property	*Combinational area*		*Sequential area*		*Total area*	
	Ours	*FoCs*	*Ours*	*FoCs*	*Ours*	*FoCs*
Prop1	94	331	308	2172	**402**	**2503**
Prop2	55	73	131	450	**186**	**523**
Prop3	60	120	141	618	**201**	**738**
Prop4	120	114	371	639	**491**	**753**
Prop5	67	40	256	198	323	238
G(a->next b)	12	6	50	30	62	36
G(a->next[10]b)	23	15	111	93	134	108
G(a->next_a[1 to 3](b))	20	11	62	58	82	69
G(a->next_a[5 to 20](b))	58	50	181	268	**239**	**318**
G(a->next_e[1 to 2](b))	20	8	76	37	96	45
G(a->next_event_e(b) [1 to 6](c))	57	36	123	184	**180**	**220**
G(a->(b before_ c))	26	12	62	44	88	56

4. Implementation of Monitors

4.1 Design Flow with Assertion Monitors

To implement PSL assertions in a digital system, the designer follows the standard design flow (HDL description, synthesis, place, and route) presented in Figure 1.6. The PSL assertions are extracted from the system specification. The extraction can be performed either from a high-level system description, from the hardware description, or from the software (see dashed lines in Figure 1.6). In the last case, the software is executed on a processor core that is instrumented at hardware level with the monitors. Once the PSL assertions

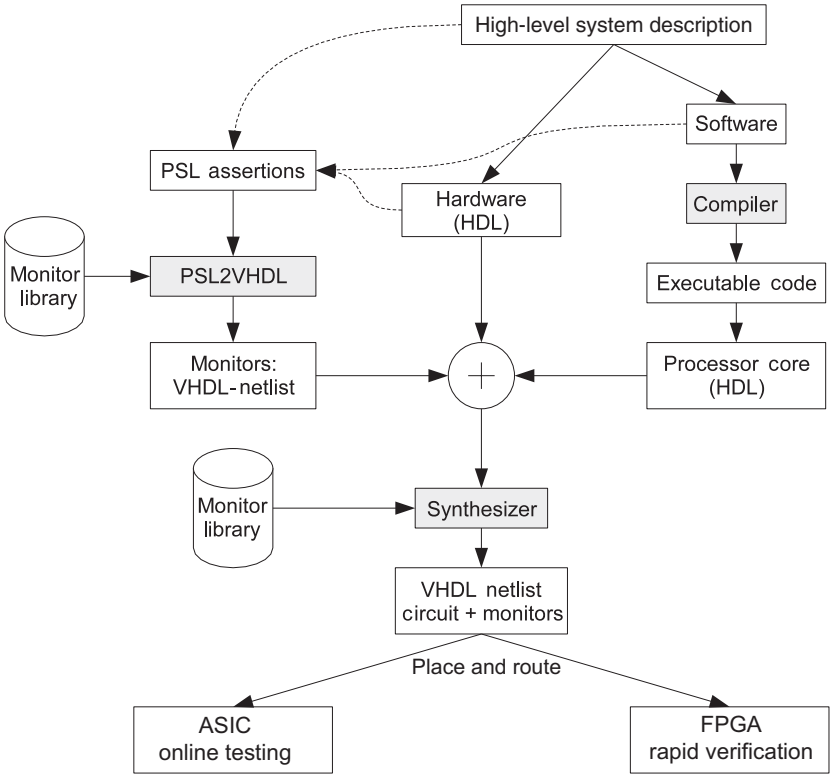


Fig. 1.6 Design flow to implement assertion monitors

have been extracted, the monitors are automatically generated by a dedicated synthesizer (PSL2VHDL), resulting in a netlist of property monitors written in synthesizable RT-level VHDL. This netlist is then merged with other hardware cores, to be synthesized with a commercial tool. At this point, the only assumption made is that the signal names in the hardware core description and in the PSL language are the same. The monitor output signals can be observed directly if connected to the pins. Otherwise, a dedicated monitor interface can be more convenient to read the monitor state and to observe the system under verification.

After synthesis, the generated circuit (hardware cores and monitors) follows the standard steps of a design flow to be targeted either in an ASIC or in an FPGA, depending on the purpose of the monitors. Monitors implemented in an ASIC are primarily devoted to online testing of the circuit in operation. If the FPGA implementation is chosen, the monitors can be used to detect design errors at the hardware or software level, the primary interest being several orders of magnitude in the verification speed compared with a simulation execution.

4.2 Example: A Bus Snoop System for Software Verification

To demonstrate the hardware monitor principles on a real system, an experimental platform, based on an Altera FPGA (a Stratix 1s40), has been designed. The implemented architecture is described in Figure 1.7. The Nios-Avalon architecture is based on a standard Avalon bus and has an Universal Asynchronous Receiver–Transmitter (UART) serial interface, a Nios processor with a RAM, and a boot ROM. The hardware monitors are connected to the bus through a small interface in order to snoop the data transactions about which the PSL properties are written. The interface also allows the Nios processor to scan the state (pending, hold, fail) of the monitors. Figure 1.7 displays an experiment in which three PSL properties are compiled through our PSL2VHDL compiler and are then synthesized on the FPGA.

The host computer is used to load the hardware on the FPGA (with a Joint Test Action Group (JTAG) link not represented in Figure 1.7). Then, the software is downloaded through the UART link and executed on the Nios processor. A typical Nios executable code results from the compilation of a program that performs read/write operations on an external device, thus activating bus transactions.

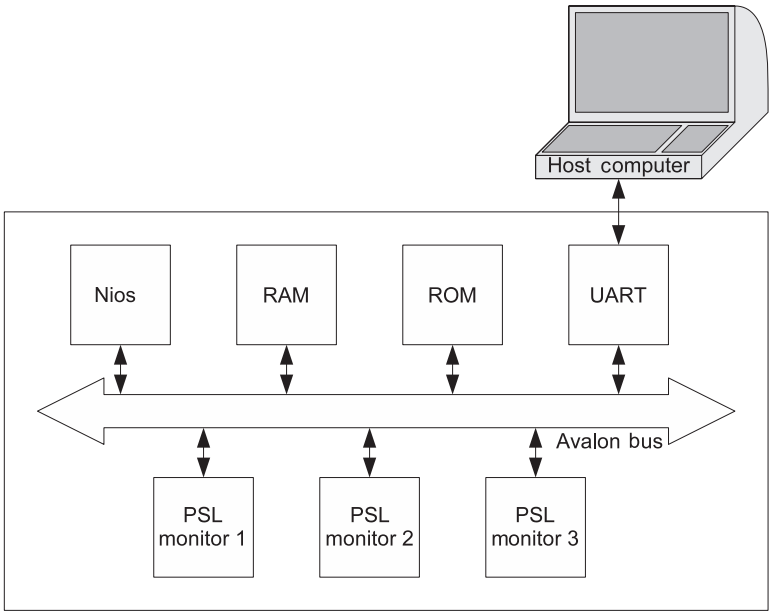


Fig. 1.7 The bus snoop system implemented on an FPGA with assertion monitor

Each monitor snoops its own set of signals on the Avalon bus and evaluates a particular property. The dedicated bus–monitor interfaces are individually controllable by software, which provides great flexibility. After one or more monitors are started, as long as all hardware monitors are in pending state, the Nios executes its program normally. When a monitor detects a hold or fail condition, an interrupt is generated and the Nios processor executes an exception handler. The interrupt routine performs appropriate actions for debug, e.g., read the state of the implied monitor and display it on the host computer.

As an illustration, the prototype has been demonstrated at the DATE'05 exhibition. Hardware monitors for the properties `P1` to `P4` listed below were exercised by stimuli sent by a program running on the Nios; the user could select the stimuli via the screen and keyboard of the host computer through the UART link:

```
vprop P1 assert always (Req -> (Ack before! Grant));
vprop P2 assert (always (Req -> next_a [1 to 3] (Ack)));
vprop P3 assert (always (Req -> next_event (Ack)[3](Grant)));
vprop P4 assert (always (Req -> (Req until Ack) before Grant));
```

5. Conclusion

In this chapter, we described a new efficient methodology to generate monitors from PSL assertions. The technology is based on a library of primitive components that implement the PSL logical and temporal operators and on a structural method to interconnect them. Our prototype implementation automatically synthesizes monitors from a PSL property, as an RT-level VHDL component netlist. Our practical experiments show that the area of the monitors thus obtained increases gracefully with the number of nested PSL operators, and the upper bound of the observation window of the (`next`, `next_event`) operators. Various applications are foreseen:

Design verification: The monitor is implemented on FPGA and serves as a fast verification tool connected to a hardware prototype of the DUV.

Safety critical application insurance: The monitor is implemented together with the DUV and permanently snoops on its critical outputs or state elements, to catch possible errors that would violate an essential property.

Synthesis of special-purpose hardware modules: These modules are specially dedicated to the observation of events or event sequences specified in PSL. Such monitors, directly produced through the application of our method, could trigger the monitored design for wake up (power-saving applications) or special behavior initiation (alarm, reconfiguration, online testing, special processing, etc.).

Current work aims at removing some limitations of the current library, which is strictly limited to scalar bit and Boolean operands. Also, the implementation of a term rewriting preprocessing step should enlarge the set of supported temporal expressions involving SEREs.

In another direction, a complete formal proof of the library components is being performed. Theorem-proving techniques are applied to obtain a proof that is independent of the valuation of the temporal parameter (number of cycles or time interval) present in many operators.

Acknowledgments

The authors thank Y. Wolfsthal, E. Zarpas, and G. Shapir from the IBM Haifa Research Laboratory for giving free access to RuleBase and FoCs.

References

Abardanel, Y., Beer, I., Gluhovsky, L., Keidar, S., and Wolfsthal, Y. (2000) FoCs: automatic generation of simulation checkers from formal specifications. In: Emerson, E.A. and Sistla, A.P. (eds) *Computer Aided Verification 12th International Conference, CAV 2000*, vol. 1855 of *Lecture Notes in Computer Science*. Springer, Chicago, IL ISBN: 3-540-67770-4, also available from `http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/ps/checkers.ps`.

Accellera (2003) *Property Specification Language Reference Manual*. Accellera, version 1.01 edition.

Accellera (2004) *Property Specification Language Reference Manual*. Accellera, version 1.1 edition.

Beer, I., Ben-David, S., Eisner, C., and Landver, A. (1996) RuleBase: an industry-oriented formal verification tool. In: *Proceedings of the 33st Conference on Design Automation*. ASM Press, Las Vegas, NV, pp. 655–660.

Claessen, K. and Martensson, J. (2004) An operational semantics for weak PSL. In: Hu, A.J. and Martin, A.K. (eds) *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004*, vol. 3312 of *Lecture Notes in Computer Science*. Springer Austin, TX, pp. 337–351.

Cohen, B., Venkataramanan, S., and Kumari, A. (2004) *Using PSL/Sugar for Formal and Dynamic Verification*. Los Angeles, CA. VhdlCohen Publishing, 2nd edn.

Daniele, M., Giunchiglia, F., and Vardi, M. (1999) Improved automata generation for linear temporal logic. In: Halbwachs, N. and Peled, D. (eds). *Computer Aided Verification: 11th International Conference, CAV'99*, vol. 1633

of *Lecture Notes in Computer Science*. Springer, Trento, Italy, pp. 249–260. ISBN: 3-540-66202-2.

Formal Methods Group (2000) Guide to Sugar formal specification language. Technical Report Version 1.3.1, IBM Haifa Research Laboratory, `http://www.haifa.il.ibm.com/projects/verification/sugar/literature.html`.

Foster, H., Krolnik, A., and Lacey, D. (2003) *Assertion-Based Design*. Kluwer Academic Publishers.

Gascard, E. (2005) From sequential extended regular expressions to deterministic finite automata. In: *Proceedings of the ITI 3rd International Conference on Information & Communications Technology (ICICT 2005)*. IEEE, Cairo, Egypt.

Gastin, P. and Oddoux, D. (2001) Fast LTL to büchi automata translation. In: Berry, G., Comon, H., and Finkel, A. (eds) *Computer Aided Verification: 13th International Conference, CAV 2001*, vol. 2102 of *Lecture Notes in Computer Science*. Springer, Paris, France, pp. 53–65.

Gordon, M., Hurd, J., and Slind, K. (2003) Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In: Geist, D. and Tronci, E. (eds) *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, vol. 2860 of *Lecture Notes in Computer Science*. Springer, L'Aquila, Italy, pp. 200–215.

IBM (2005) *Sugar-Based tools*. IBM Haifa Research Laboratory, `http://www.haifa.il.ibm.com/projects/verification/sugar/tools.html`.

Liu, M. (2004) Generation of a verification environment for the standard property specification language. Unpublished research report, TIMA, Grenoble, France.

Sebastiani, R. and Tonetta, S. (2003) More deterministic vs. smaller büchi automata for efficient LTL model checking. In: Geist, D. and Tronci, E. (eds) *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, vol. 2860 of *Lecture Notes in Computer Science*. Springer, L'Aquila, Italy, pp. 126–140.

Chapter 2

Refining Synchronous Communication onto Network-on-Chip Best-Effort Services

Zhonghai Lu, Ingo Sander, and Axel Jantsch
Department of Electronic, Computer and Software Systems
Royal Institute of Technology
Sweden

Abstract We present a novel approach to refine a system model specified with perfectly synchronous communication onto a network-on-chip (NoC) best-effort communication service. It is a top-down procedure with three steps, namely, *channel refinement*, *process refinement*, and *communication mapping*. In channel refinement, synchronous channels are replaced with stochastic channels abstracting the best-effort service. In process refinement, processes are refined in terms of interfaces and synchronization properties. Particularly, we use *synchronizers* to maintain local synchronization of processes and thus achieve *synchronization consistency*, which is a key requirement while mapping a synchronous model onto an asynchronous architecture. Within communication mapping, the refined processes and channels are mapped to an NoC architecture. Adopting the *Nostrum* NoC platform as target architecture, we use a digital equalizer as a tutorial example to illustrate the feasibility of our concepts.

Keywords: synchronous model; communication refinement; network-on-chip.

1. Introduction

For system design, a synchronous design style is attractive since it allows us to separate timing from function. The designer can focus on the design of the system functionality without being distracted by unnecessary low-level communication details. This also facilitates the verification task, which is a key activity at the system level. Later, *refinement* explores the implementation space under constraints, making design decisions and filling in implementation details. Network-on-Chip (NoC) is an emerging system-on-chip (SoC) paradigm aimed to cope with the scalability problem of various buses in order to connect tens or perhaps even hundreds of microprocessor-sized heterogeneous resources, such as processor cores, digital signal processings (DSPs),

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 23–42.

field programmable gate-arrays (FPGAs) application-specific integrated circuits (ASICs), and memories. The complex integration is desired by ever-increasing functionality and enabled by the steady technology scaling. Nostrum (Millberg et al., 2004; Nilsson et al., 2003; Thid et al., 2003) is our NoC architecture offering a packet-switched communication platform. To satisfy different performance/cost requirements, Nostrum provides two classes of communication services: best effort (BE) and guaranteed bandwidth (GB) services. The BE service is connectionless where packets are routed without resource reservation. The GB service is connection-oriented where packets are delivered after enough bandwidth is reserved. It achieves better performance at the expense of higher cost.

In this chapter, we are interested in mapping a system specified as a synchronous model onto an NoC. To this end, we propose an NoC design flow shown in Figure 2.1 where we concentrate on the communication problem. There are three communication-related tasks: *clustering and resource allocation*, *communication refinement*, and *synthesis*. The clustering flattens the hierarchy in the model and groups processes into new processes with coarser granularity. With resource allocation, the grouped processes are allocated to network nodes, either hardware (HW) or software (SW) execution resources. Communication refinement bridges the gap between the communication model in the specification and the NoC communication implementation by adapters. With synthesis, these processes and adapters are synthesized into HW and/or SW.

We address the *communication refinement* that starts from a synchronous communication model and ends with the Nostrum NoC best-effort communication services. With the specification model, communication is perfectly synchronous with a global logical clock and cleanly separated from computation. With the NoC communication service, communication introduces variable delays and crosses multiple clock domains connected by a packet-switched network. Clearly, the communication in the implementation domain is not synchronous, and thus not consistent with that in the specification domain.

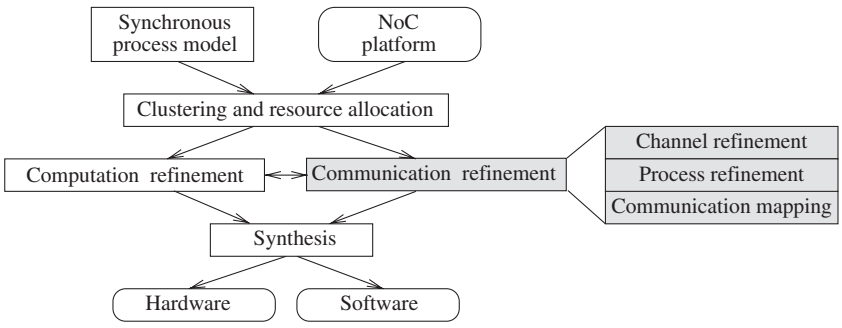


Fig. 2.1 An NoC design flow

Our contributions are (1) a novel approach to realize this communication refinement; (2) a classification of process synchronization properties such as *strict*, *nonstrict*, *strong*, and *weak* synchronization in order to formally analyze the processes' local synchronization requirement(s) (Section 5.2); (3) using *synchronizers* (synchronization adapters) to maintain synchronization consistency during refinement (Section 5.3). We will focus on the synchronization issue while keeping the process computation untouched. Note that this synchronization issue lies at the system-modeling level and not at the lower implementation levels such as shared memory synchronization using locks or semaphores, as well as message-passing synchronization using blocking or nonblocking semantics. We assume that, after a clustering, the resulting processes, more precisely the process networks, are top-level entities. Each process may comprise a hierarchy of subprocesses, which are intended to reside in a synchronous implementation domain. Besides, we consider that a resource maintains a local synchronous region. Consequently, a process is to be mapped to one resource and one resource hosts exactly one process.

2. Related Work

Based on the isolation of communication from computation, a large body of work on communication refinement exists. Through the virtual component interfaces (VCI) of the VSI Alliance (Lennard et al., 2000), the COSY-VCC design flow (Brunel et al., 2000) supports communication refinement from specification to performance estimation and to implementation. IPSIM (Coppola et al., 2003) developed on top of SystemC 3.0 supports an object-oriented methodology and establishes two intermodule communication layers. The message box layer concerns generic and system-specific communication, whereas the driver layer implements higher level application-dependent communications. The SpecC methodology defines four levels of abstraction, namely, at the specification, architecture, communication, and implementation level, and the refinement transformations between them (Dömer et al., 2002). These works do not assume a synchronous specification.

With synchronous communication, latency insensitive theory (Carloni et al., 2001) targets synchronized HW design where synchronization can still be achieved using relay stations even if interconnecting synchronous image processing (IP) blocks experiences indefinite wire latencies. Desynchronization for SW design was addressed in Benveniste et al. (2000). Furthermore, some mathematical frameworks were developed to support refinement-based design methods. Benveniste et al. present a theoretical framework for modeling heterogeneous systems, and derive sufficient conditions to maintain semantic-preserving transformations when deploying a synchronous specification onto GALS and the loosely time-triggered architectures (Benveniste et al., 2003).

Another framework is proposed by Guernic et al. (2003) concerning the refinement of a polysynchronous specification, which allows the existence of multiple clocks instead of a single clock. All these works are complementary to ours but none of them provides a detailed refinement approach targeting an NoC platform.

3. Refinement Overview

3.1 The Perfectly Synchronous Model

The synchronous modeling paradigm is based on an elegant and silmple mathematical model, which is the ground of synchronous languages such as Esterel, Signal, Argos, and Lustre. The basis is the perfect synchrony hypothesis, i.e., both computation and communication take no observable time. A system is modeled as a set of concurrent communicating processes through signals. Processes use ideal data types and assume infinite buffers. Signals are ordered sequences of events. Each event has a time slot as a slot to convey data. If the data contains useful information, the event is *present* and is called a *token*. Otherwise, the event is *absent* and modeled as a $\sqcup$ representing a clock tick. Each signal can be related to the time slots of another signal in an unambiguous way. The output events of a process occur in the same time slot as the corresponding input events. Moreover, they are instantaneously distributed in the entire system and are available to all other processes in the same slot. Receiving processes in turn consume the events and emit output events again in the same time slot. The medium through which a signal passes can thus be viewed as an ideal communication channel that has no delay for any event data types (unlimited bandwidth). A process specified in the synchronous paradigm is a synchronous process. For feedback loops, the perfect synchrony creates cyclic dependency between output and input, and thus leads to deadlock, which can be resolved with initial events in the specification. A synchronous model is deterministic, i.e., if given the same input streams, it generates the same output streams.

As a tutorial example, Figure 2.2 illustrates an equalizer model. It adjusts the bass and treble volume of the audio stream according to button control levels. In addition, it prevents the bass level from exceeding a predefined threshold to

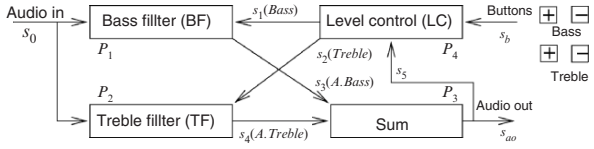


Fig. 2.2 The digital equalizer

avoid damaging the speakers. Its function can be described by the following set of equations, where the initial value "1" is used to resolve the feedback loops:

$$
\begin{array}{lcl}
\mathit{AudioOut} & = & \mathit{Equalizer}(\mathit{Buttons}, \mathit{AudioIn}) \\
\quad \text{where} & & \\
\quad \mathit{AudioOut} & = & \mathit{Sum}(\mathit{AudioBass}, \mathit{AudioTreble}) \\
\quad (\mathit{Bass}, \mathit{Treble}) & = & \mathit{LevelControl}(\mathit{Buttons}, \mathit{AudioOut}) \\
\quad \mathit{AudioBass} & = & \mathit{BassFilter}(\mathit{AudioIn}, \mathit{init} : \mathit{Bass}) \\
\quad \mathit{AudioTreble} & = & \mathit{TrebleFilter}(\mathit{AudioIn}, \mathit{init} : \mathit{Treble}) \\
\quad \mathit{init} & = & 1
\end{array}
$$

This model is specified in the functional language Haskell and is executable.

3.2 Nostrum Communication Services

In Nostrum, each resource R_i $(i = 1, 2, \ldots, n)$ is equipped with a resource-network-interface (RNI) in order to access the network, as shown in the lower part of Figure 2.3. The RNI and the network belong to the Nostrum protocol stack. Nostrum provides a message-passing platform with two communication services, i.e., BE and GB. The BE service (Nilsson et al., 2003) is connectionless. Packets are routed in the network without reserving network resources such as storage and link bandwidth. The end-to-end flow control, reordering, packetization, and packet admission control are performed by RNIs. The BE service maintains message order, and is lossless and corruptless. It has no guarantee on timely delivery, but must have an upper bound on delivery time. To this end, we assume that the communication protocols can prevent the network from saturation and guarantee bounds on delay. The GB

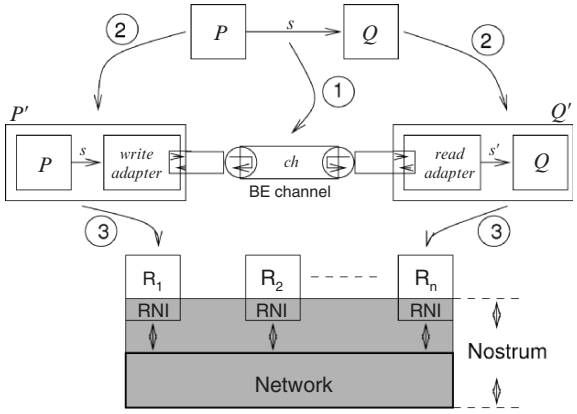


Fig. 2.3 Communication refinement overview

service is connection-oriented. Bandwidth is negotiated during a connection-establishment phase. Packets are delivered after a connection is established. The GB service is implemented by using looped containers and temporally disjoint networks (Millberg et al., 2004). The RNIs hide the service implementation details and make the services *transparently* accessible to applications. The access methods are communication primitives offered to the higher layer.

Within Nostrum, we define a set of basic communication primitives for message passing as follows:

int open(int src, int dst, int service, struct bandwidth): it opens a simplex channel between a source `src` process and a destination `dst` process. The `service` denotes the channel service class: 0 for the BE service, 1 for the GB service. The `bandwidth` is a user-defined record with three fields `{int min_bw, int avg_bw, int max_bw}` which specifies the minimum, average, and maximum bandwidth (bytes/second) requirement of the channel. The method returns a unique channel identity number (`cid`) upon successfully opening the channel. Otherwise, it returns various reasons of failure, such as a destination invalid or performance not satisfied.

bool write(int cid, void msg): it writes `msg` to the specified channel `cid`. The size of messages is bounded. It returns the status of the write.

bool read(int cid, void *msg): it reads channel `cid` and writes the received data to the address starting at `msg`. It returns the status of the read.

We have implemented these primitives with the BE service using SystemC in our layered NoC simulator *Semla* (Thid et al., 2003). The `write()` and `read()` are presently implemented with nonblocking semantics. Semla is programmable as to network topology, process-to-resource mapping, routing algorithm, and traffic pattern. The current implementation opens channels statically during compile time and the opened channels are never closed during simulation.

3.3 The Refinement Procedure

Given a synchronous system specification, our objective is to refine the synchronous communication onto the Nostrum BE service. For this communication refinement, we propose a three-step procedure: *channel refinement*, *process refinement*, and *communication mapping*. We illustrate the procedure by a pair of producer–consumer processes in Figure 2.3. The three steps are marked by a circle with a step number inside it.

Step 1: With *channel refinement*, we first abstract the behavior of the Nostrum BE service as that of stochastic channels which are then used to replace

the ideal communication channels for passing signals. In Figure 2.3, the ideal channel for signal s between producer P and consumer Q is refined to a BE service channel *ch*. After being delivered through the service channel, signal s turns into signal s', which is a derived version of s. Furthermore, s and s' are not synchronous since different clock domains are involved in the service channel.

Step 2: With *process refinement*, we discuss how to connect a process to the service interface and how its synchronization property can be met by using adapters to wrap the process. Particularly, to guarantee a correct refinement, the process synchronization property must be consistent from the specification to the refined model. We classify and analyze the synchronization property of processes and then discuss how to maintain *synchronization consistency*. The process synchronization property can be annotated by designers on processes to enable automatically instantiating *synchronizers* to achieve synchronization consistency in the process refinement. Moreover, we consider design decisions to handle feedback loops, by which the process synchronization may be relaxed in order to optimize performance since a synchronous specification may overspecify the system. In Figure 2.3, P and Q are wrapped with a write and a read adapter, respectively. Note that an adapter contains both a component to interface with the service channel (writer/reader) and component(s) to achieve synchronization consistency (synchronizers) whenever necessary.

Step 3: Finally, together with a process-to-resource allocation scheme, the *communication mapping* is to implement the adapters and map the service channels on an NoC, in this case, the Nostrum simulator Semla. In Figure 2.3, the refined processes P' and Q' are mapped to the resources R_1 and R_n, respectively. Accordingly, the service channel *ch* is implemented through the interfaces provided by the RNIs of the resources R_1 and R_n.

4. Channel Refinement

The Nostrum BE service provides in-order, lossless, and bounded-in-time communication between processes. However, its performance is *nondeterministic* since the message delivery experiences dynamic contentions in the RNIs and network. To capture the characteristics of the BE service, we resort to a stochastic approach. Formally, we develop a unicast BE service channel as a point-to-point *stochastic* channel: given an input signal of messages $\{m_1, m_2, \ldots, m_n\}$ to the service channel, the output signal is $\{d_1, m_1, d_2, m_2, \ldots, d_n, m_n\}$, where message m_i $(i = 1, 2, \ldots, n)$ is bounded in size; d_i denotes the delay of m_i, which may be expressed as the number of absent ($\sqcup$) values and

is subject to a distribution with a minimum $d_{i,min}$ and maximum $d_{i,max}$ value. The actual distribution, which may differ from channel to channel, is irrelevant. We do not make any further assumptions about this. If $d_i = n$ (n is a positive integer), it means there are n absent values between m_{i-1} and m_i. We can identify two important properties of the generic service channel behavior: (1) d_i is varying and (2) d_i is bounded. This behavior is purely viewed from the perspective of application processes and its implementation details are hidden.

Replacing the ideal channel (zero delay and unlimited bandwidth) with a stochastic channel (varying delay and limited bandwidth) leads to the violation of the synchrony assumption. In the specification, a channel is ideal so that we can use a *single* signal s to connect a producer to a consumer process. After replacing the ideal channel with a service channel, the signal s can be seen as being *split* into a pair of signals, the original signal s and its derived signal s', as shown in Figure 2.3. For a process with two synchronous input signals, e.g., the *Sum* process of the equalizer (Figure 2.2), if both signals s_3 and s_4 are delivered through a service channel, they are split, resulting in two derived signals s'_3 and s'_4, which are now the input signals to the *Sum* process. Apparently, the two pairs of signals, s_3 and s'_3, s_4 and s'_4, and the two derived signals s'_3 and s'_4 are not synchronous. A synchronous system becomes globally asynchronous, leading to a possibly nondeterministic behavior that deviates from the specification. It is therefore important for a refinement to maintain synchronization consistency for functional correctness.

5. Process Refinement

We first describe how to interface with the service channels in general, and then discuss the synchronization property of processes followed by methods to achieve synchronization consistency. At the system level (a composition of processes), we discuss feedback loops.

5.1 Interfacing with the Service Channels

Once an ideal channel is replaced by a service channel, the processes can not be directly connected to the interface of the service channel. They must be *adapted* in terms of data and control because (1) the input/output data type of a service channel is of a bounded size whereas a signal in the specification assumes an ideal data type, whose length is finite but arbitrary, e.g., a 32/64-b integer, a 64-b floating point, or a user-defined 256-b record type, etc.; or (2) the service channel has bounded buffers and limited bandwidth whereas a signal uses unlimited resources. The sending and receiving of messages use shared resources and thus control functionality has to be added to allocate shared resources, schedule multiple threads, and achieve thread-level synchronization. These adaptations are achieved by a writer and reader process.

Specifically, to interface with the service channels, a producer needs to be wrapped with a *writer* and a consumer with a *reader*.

5.2 Process Synchronization Property

In the system model, all signals of processes are synchronous. However, whether or not the input signals of a process must be synchronous is subject to the evaluation condition of processes, specifically, the local condition(s) to *evaluate* the input events. Because of the tight synchronization in the model, some processes may be overspecified, limiting the implementation alternatives. During the refinement, the designer(s) must inspect and determine the synchronization property of the processes.

Inspired by Lee and Parks (1995), we use *firing rules* to discuss the synchronization property of *synchronous processes*. For a synchronous process with n input signals, PI is a set of N input patterns, $PI = \{I_1, I_2, \ldots, I_N\}$. The input patterns of a synchronous process describe its firing rules, which give the conditions of evaluating input events at each event cycle. I_i ($i \in [1, N]$) constitutes a set of event patterns, one for each of n input signals, $I_i = \{I_{i,1}, I_{i,2}, \ldots, I_{i,n}\}$. A pattern $I_{i,j}$ contains only one element that can be either a token wildcard $*$ or an absent value $\sqcup$, where $*$ does not include $\sqcup$. Based on the definition of firing rules, we propose four levels of process synchronization properties as follows:

Strict synchronization. All the input events of a process must be present before the process evaluates and consumes them. The only rule that the process can fire is $PI = \{I_1\}$, where $I_1 = \{[*], [*], \ldots, [*]\}$.

Nonstrict synchronization. Not all the input events of a process are absent before the process fires. The process *cannot* fire with the pattern $I = \{[\sqcup], [\sqcup], \ldots, [\sqcup]\}$.

Strong synchronization. All the input events of a process must be either present or absent in order to fire the process. The process has only two firing rules $PI = \{I_1, I_2\}$, where $I_1 = \{[*], [*], \ldots, [*]\}$ and $I_2 = \{[\sqcup], [\sqcup], \ldots, [\sqcup]\}$.

Weak synchronization. The process can fire with any possible input patterns. For a two-input process, its firing rules are $PI = \{I_1, I_2, I_3, I_4\}$, where $I_1 = \{[*], [*]\}$, $I_2 = \{[\sqcup], [\sqcup]\}$, $I_3 = \{[*], [\sqcup]\}$, and $I_4 = \{[\sqcup], [*]\}$.

We can identify processes with a *strict*, *strong*, and *weak* synchronization property in the equalizer (Figure 2.2). The *bass filter* (s_0 and s_1) and *treble filter* (s_0 and s_2) have a strict synchronization. Both filters are composed of a finite impulse response (FIR) filter and an amplifier. The FIR filter is specified as a finite state machine (FSM), whose state transition is sensitive to time; thus a $\sqcup$ value in an audio stream can change the values of its output sequence. Meanwhile, the amplifier must have an amplification level, thus

a $\sqcup$ value makes the amplifier undefined. The *Sum* process (s_3 and s_4) has a strong synchronization. It is a combinational process and thus tolerable to events with a $\sqcup$ value. However, the two events of s_3 and s_4 must be synchronized before being processed since they represent the low- and high-frequency components of the same audio sample. The *level control* (s_b and s_5) process has a weak synchronization. It can fire even when either or both of the events of s_b and s_5 are absent since pressing buttons happens irregularly and the bass level surpassing the threshold occurs only aperiodically.

5.3 Achieving Synchronization Consistency

Apparently, for processes with a strict or strong synchronization, their synchronization properties cannot be satisfied if any of their input signals passes through a service channel since the delays through the channel are stochastic. Although globally asynchronous, the processes can be locally synchronized by using *synchronizers* to satisfy their synchronization properties. To achieve strong synchronization, we use an align-synchronization process *sync*; to achieve strict synchronization, we use three processes, *sync*, *deSync*, and *addSync*. We use a two-input process to illustrate these processes in Figure 2.4. An align-synchronization process *sync* aligns the tokens of its input events, as shown in Figure 2.4a. It does not change the time structure of the input signals. A desynchronizer *deSync* removes the absent values, as shown in Figure 2.4b. All its input signals must have the same token pattern, resembling the output signals of the *sync* process. Removing absent values implies that the process is *stalled*. The desynchronizer changes the timing structure of the input signals, which must be recovered in order to prevent from causing

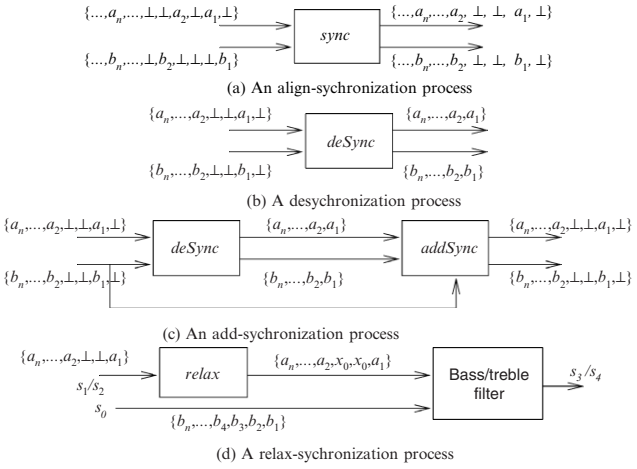


Fig. 2.4 Processes for synchronization

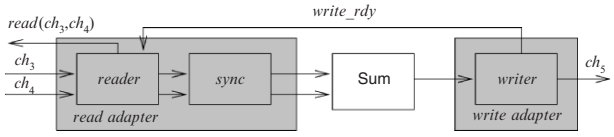


Fig. 2.5 Read/write adapters for a process with strong synchronization

unexpected behavior of other processes that use the timing information. An add-synchronizer *addSync* adds the absent values to recover the timing structure, as shown in Figure 2.4c. It must be used in relation to a *deSync* process. If the input events of the *deSync* is a token, the *addSync* reads one event from its internal buffers for each output signal; otherwise, it outputs a $\sqcup$ event. The two processes *deSync* and *addSync* are used as a pair to assist processes to fulfill strictness.

We can now use these synchronizers in connection with the *reader* and *writer* processes to wrap the original processes to interface with the service channels and maintain the synchronization consistency from the specification model to the refined model. For instance, as shown in Figure 2.5, we use a *sync* process and a pair of *reader*/*writer* processes to wrap the *Sum* process in the equalizer to maintain its strong synchronization. We use the three processes, *sync*, *deSync*, and *addSync*, and a pair of *reader*/*writer* processes to wrap the *bass/treble filter* process (Figure 2.2) to maintain their strict synchronization.

The refinement of processes with a nonstrict synchronization should be individually investigated according to their firing rules.

5.4 Feedback Loops

In the specification, feedback loops are resolved by using initial events. If the feedback signals pass through a service channel, the delays are nondeterministic. If following the initial-event approach in the refinement procedure, we encounter a problem since we are not certain how many initial events are required to resolve the deadlock. Consider the *bass/treble filter*, if the tokens of s_1/s_2 are not available, it cannot fire. This implies it may not be able to process enough audio samples in time, leading to violate the system's performance constraint. However, if the amplification level signals s_1 (*bass*) and s_2 (*treble*), are delayed and thus not available, the amplifiers should continue functioning by, for example, using the previous amplification level or simply using a constant level like 1. In this case, the effect of pressing buttons may be delayed several cycles. This is tolerable since the human sensing of the changes in the audio volume is not instantaneous.

By this observation, we can in fact *relax* the strict synchronization of the processes *bass/treble filter*, using a relax-synchronization process *relax* illustrated in Figure 2.4d. If the input event is a token, it outputs the token; otherwise, a token x_0 is emitted. The exact value of x_0 is application-dependent. Relaxing synchronization is a design decision leading to behavior discrepancy between the specification and the refined model. Care must be taken to validate the resulting system.

6. Communication Mapping

The inputs to this task are the refined model as well as a process-to-resource allocation scheme; the output is a communication implementation on Semla.

6.1 Channel Mapping

With a resource allocation scheme, all processes are allocated to resources in a one-to-one manner. Note that this is not a limitation but due to the assumption on the clustering and resources (refer to Section 1). With such a clustering, interprocess signals, which represent interresource communications, are mapped to service channels. Since the processes may be hierarchical, we need to flatten the hierarchy to the level that each signal mapped to a service channel can be uniquely identified with a pair of a producer and a consumer process with *finer* granularity. For simplicity, we do not consider mapping multiple service channels to one implementation channel. Mapping channels is thus straightforward. Each pair of processes communicating through a service channel in the refined model results in its dedicated unicast implementation channel, which is mapped to the open channel primitive `open()`. For example, with the producer–consumer case, a BE channel setup is fulfilled by a single line of code: `int ch[1]=open(P,Q,BE_SERVICE,NULL)`.

6.2 Communication Process Mapping

After the process refinement, a refined process consists of the original computational process, the writer and reader, and perhaps the synchronizer(s) to satisfy their synchronization properties. Our refinement keeps the original processes intact. Therefore, the tasks of communication process mapping are to implement the writer/reader and the synchronizers such as *sync*, *deSync*, *addSync*, and *relax*, and to coordinate the writing and reading operations.

In SystemC, processes are implemented as modules. The reader/writer may be implemented as separate modules or in the same modules as processes. We implement a process and its adapter(s) in a single module. For implementation, execution control in the module must be considered. Suppose the module has a single thread of control, we need to find a periodic admissible sequential schedule (PASS) for process executions (Lu et al., 2002). For the process

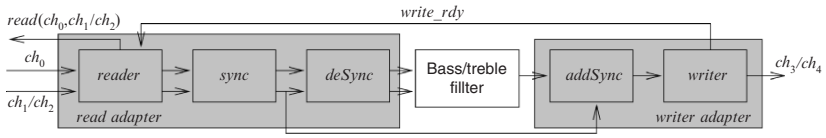
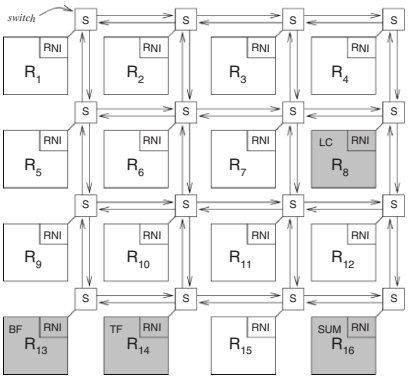


Fig. 2.6 Read/write adapters for a process with strict synchronization

in Figure 2.6, a PASS could be PASS={*reader*, *sync*, *deSync*, *Bass/Treble Filter*, *addSync*, *writer*}. Besides, a control signal *write_rdy* must be asserted by the *writer* to the *reader* to enable reading the channel(s) for the next-round PASS execution, as shown in Figure 2.6. This leads to a local feedback loop, and we adopt the initial-event approach to deal with it. In this case, *write_rdy* is initially asserted. Using the communication primitives defined in Section 3.2, the SystemC module for Figure 2.6 is sketched as follows, with each component explained briefly in commentary:

```
process_class::Process(){
  //initially write_rdy=1;
  //read_ch0_rdy=0; read_ch1_rdy=0
  //sync_rdy=0; compute_done=0;
  if (write_rdy==1){
  //(1) reader: nonblocking read ch1 and ch2
   if (read_ch0_rdy==0)
     if ((read(ch[0],&r_msg1))==true)
        read_ch0_rdy=1;
   if (read_ch1_rdy==0)
     if ((read(ch[1],&r_msg2))==true)
        read_ch1_rdy=1;
  //(2) sync: synchronize the two events
  if (read_ch0_rdy==1 && read_ch1_rdy==1)
     sync_rdy=1;
  else sync_rdy=0;
  //(3) deSync: desynchronization by guard
  if (sync_rdy==1 && compute_done==0){
     //process computation
     //return w_msg and set compute_done to 1
     w_msg=compute(r_msg1,r_msg2);
     write_rdy=0; compute_done=1;}
 }
 //(4) addSync: fill synchronization
  if (sync_rdy==1 && compute_done==1) {
  //(5) writer: nonblocking write ch3
   if (write_rdy==0)
     if (write(ch[3],w_msg)==true){
        write_rdy=1;
        sync_rdy=0; compute_done=0;
        read_ch0_rdy=0; read_ch1_rdy=0;}}
}
```



(a) Resource allocation

Load (%)	Average delay (cycles)	Throughput (samples/cycle)
9.06	15.00	0.0667
16.96	16.53	0.0605
25.16	18.52	0.0540
29.97	20.24	0.0494
35.36	22.59	0.0443
43.28	35.14	0.0285

(b) Performance

Fig. 2.7 The equalizer mapped on an NoC

In the implementation domain, whether to emit and pass $\sqcup$ through a service channel either as a special message or using one bit to indicate *presence* and *absence* can be a design decision. To preserve the semantics, $\sqcup$ must be transported. However, this incurs too much overhead on computation and communication, and may be meaningless since its value is useless. Therefore $\sqcup$ is usually neglected. Only in cases where the timing information carried by $\sqcup$ is used by other processes, it must be emitted and passed. In the equalizer case, $\sqcup$ is neglected since its timing information is not used by any of the four processes.

We have implemented the equalizer in Semla. The purpose is to validate the concepts of our refinement approach. Figure 2.7a illustrates the mapped equalizer in a 4×4 mesh NoC. All the five interresource signals $s_1, s_2, \ldots, s_5$ (Figure 2.2) use the BE service. The resources and the network run with the same speed. The switches operate synchronously with the switching per hop taking one cycle. The message streams on s_3 and s_4 are injected into the network conservatively so that a new audio sample will not be processed by the filters until the previous sample has been handled by the *Sum* process. This implies that the audio samples are not processed in a pipeline fashion in the network. In addition, we inject background traffic with uniformly distributed random destinations in the network. The motivation is to load the network with reasonable amount of traffic since the equalizer example can make use of only a small fraction of the network capacity. Figure 2.7b shows the equalizer performance, where the network load is the average percentage of active links per cycle. The process computations are function calls and complete instantly. We observe the average delay that is the time (in cycles) to process one sample. Since the audio processing is not pipelined, the throughput (samples/cycle) is simply the inverse of the average delay. In Figure 2.7b, the first row shows the

case where there is no background traffic. As expected, when the network is increasingly loaded, the average delay is increased and the throughput decreased. The average delay can be seen as the time to respond to a button press or to activate bass control. We noted that the audio output sequences are different from those observed from the specification due to relaxing the synchronization for the feedback loops. We conducted other experiments in which we removed the feedback loops, and could validate that the output sequences agree with each other in all traffic-setting cases.

7. Conclusions and Future Work

Communication refinement is a crucial step in an NoC design flow. We have presented a refinement approach that allows us to map a perfectly synchronous communication model onto the NoC BE service accessible through communication primitives. Particularly, we classify the synchronization properties of processes and describe methods to achieve synchronization consistency during the refinement upon the violation of the perfect synchrony hypothesis. For feedback loops, we relax the synchronization with the tolerance of system requirements. In this chapter we use Nostrum as target, but with few adjustments; this approach is also applicable for other NoC platforms.

In future work, we plan to develop formalism for synchronization consistency and realize automatically analyzing the synchronization properties of processes. During refinement, we take either automatic analysis that yields correct synchronization and system behavior or manual analysis with design decisions on the synchronization refinement combined with a systematic verification of the resulting implementation. For the refinement of feedback loops, we intend to use the Nostrum GB service to reach a systematic solution.

References

Benveniste, A., Caillaud, B., and Guernic, P. L. (2000) Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171.

Benveniste, A., Carloni, L., Caspi, P., and Sangiovanni-Vincentelli, A. (2003) Heterogeneous reactive systems modeling and correct-by-construction deployment. In: *Proceedings of the Third International Conference on Embedded Software* LCNS, Springer, Berlin.

Brunel, J.-Y., Kruijtzer, W., Kenter, H., Petrot, F., Pasquier, L., de Kock, E., and Smits, W. (2000) COSY communication IP's. In: *Proceedings of the 37th Design Automation Conference (DAC) 2000*. Los Angeles, CA.

Carloni, L. P., McMillan, K. L., and Sangiovanni-Vincentelli, A. L. (2001) Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076.

Coppola, M., Curaba, S., Grammatikakis, M., and Maruccia, G. (2003) IPSIM: SystemC 3.0 enhancements for communication refinement. In: *Proceedings of Design Automation and Test in Europe (DATE) 2003*. Munich, Germany.

Dömer, R., Gajski, D. D., and Gerstlauer, A. (2002) SpecC methodology for high-level modeling. In: *Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop*. Moneterey, CA.

Guernic, P. L., Talpin, J.-P., and Lann, J.-C. L. (2003) Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–303.

Lee, E. A. and Parks, T. M. (1995) Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

Lennard, C., Schaumont, P., de Jong, G., Haverinen, A., and Hardee, P. (2000) Standards for system-level design: practical reality or solution in search of a question? In: *Proceedings of Design Automation and Test in Europe (DATE) 2000*. Paris, France.

Lu, Z., Sander, I., and Jantsch, A. (2002) A case study of hardware and software synthesis in ForSyDe. In: *Proceedings of the 15th International Symposium on System Synthesis*. Kyoto, Japan.

Millberg, M., Nilsson, E., Thid, R., and Jantsch, A. (2004). Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In: *Proceedings of the Design Automation and Test Europe Conference (DATE) 2004*. Paris.

Nilsson, E., Millberg, M., Öberg, J., and Jantsch, A. (2003) Load distribution with the proximity congestion awareness in a network on chip. In: *Proceedings of the Design Automation and Test Europe (DATE) 2003*, pp. 1126–1127.

Thid, R., Millberg, M., and Jantsch, A. (2003) Evaluating NoC communication backbones with simulation. In: *Proceedings of the IEEE NorChip Conference*.

II

C/C++-Based System Design

Introduction

This part of the book contains the five most interesting contributions selected from the C/C++-Based System Design Thematic Area (CSD-TA) workshop, which was part of the FDL'05 conference. The CSD-TA addresses language-based hardware/software system methodologies and tools for modeling, simulating, evaluating the performance, and analyzing hardware/software systems. The focus of this thematic area is on research approaches applying C/C++-based languages like SystemC, but other languages are explicitly welcomed as well. Topics of interest also include Real-Time Operating System (RTOS) and embedded software aspects.

Through its open and extensible language architecture, SystemC provides an efficient framework for new research approaches addressing the design of complex electronic systems. On this foundation, Chapters 3–7 present methodologies and extensions for SystemC to enlarge its scope and expressiveness. Chapters 3–7 address very different design challenges such as behavioral separation in protocol-dominated systems, checking of timing properties, hardware/software integration, and mixing synchronous with untimed models of computation (MoC).

In the first chapter of this part, "Behaviour separation: a high-level methodology applicable in the SystemC environment" (Chapter 3), Giovanni et al. describe an approach to foster reuse when modeling design components communicating through protocols. Their technique enables an efficient reuse of parts of components sharing one protocol by separating the fixed protocol behavior from the device-specific behavior. The chapter illustrates the applicability and efficiency of the presented methodology using an Advanced Microcontroller Bus Architecture (AMBA) bus system master device design example.

In "Mixing synchronous reactive and untimed MoCs in SystemC" (Chapter 4), Fernando Herrara and Eugenio Villar present a methodology that allows to mix models of computations for describing components in a heterogeneous system. In particular, the work approaches the integration of the synchronous, reactive, and the untimed model of computation in the SystemC language framework. The chapter describes a concrete SystemC-based

mechanism applying so-called border channels and border processes to interface synchronous, reactive, and untimed models.

The third chapter by André et al., "Interface-centric abstraction level for rapid hardware/software integration" (Chapter 5), describes a portable set of abstract operative system (OS) primitives realized as an application programming interface (API) facilitating hardware/software codesign in SystemC. The API covers all aspects of typical operation systems, e.g., process management, communication, synchronization, and timing. For each OS primitive, they provide a hardware and a software implementation alternative. With these alternatives, the designer can easily evaluate different alternative system partitionings.

Roland et al. received the *FDL'05 Best Paper Award* for their contribution, "Efficient and customizable integration of temporal properties into SystemC" (Chapter 6). The work presents an extension of SystemC with customizable temporal properties. The two main aspects discussed in this chapter are the way to specify temporal properties in SystemC models and the mechanism to check these properties automatically. Both aspects are illustrated by modeling examples and performance-evaluation results.

The heterogeneity and complexity of electronic systems necessitates integration of different MoC into a semantically well-defined and efficient modeling environment. In "UMoC++: A C++-based multi-MoC modeling environment" (Chapter 7), Deepak et al. describe basic techniques to integrate generic MoCs taken from functional frameworks, such as ForSyDe and SML-Sys, into an efficient framework based on an imperative language. By doing so, the presented environment called UMoC++ promises to improve the efficiency while maintaining the well-defined semantic of the heterogeneous system model.

It is my hope that this short introduction will draw your attention to the very interesting work presented in the following chapters. As they outline the scope of the CSD-TA quite well, they might also encourage you to attend or even contribute to one of the following CSD workshops at the FDL conference.

Frank Oppenheimer
OFFIS e. V.
R&D division Embedded Hardware/Software-Systems
Oldenburg, Germany, February 2006

Chapter 3

Behaviour Separation: A High-Level Methodology Applicable in the SystemC Environment

Giovanni B. Vece, Massimo Conti, and Simone Orcioni
Dipartimento di Elettronica, Intelligenza artificiale e Telecomunicazioni
Università Politecnica delle Marche
via Brecce Bianche, 12
I-60131 Ancona
Italy

Abstract This chapter proposes a modelling technique called behaviour separation for the high-level design of devices bound to a working protocol. This technique is based on a device characterization through orthogonal behavioural components, favouring code reuse and making modelling effort easier. Its realization can be achieved in SystemC environment in a very suitable manner. This is possible by the advanced modelling capabilities provided by this environment, in particular the IMC communication strategy.

In this chapter, we show a concrete application of this approach based on the master devices bound to the AMBA AHB protocol.

Keywords: high-level modelling technique; code reuse; SystemC; AMBA AHB.

1. Introduction

SystemC (OSCI, 2006) is an emerging C++ environment for the modelling, the simulation, and the synthesis of electronic systems. Like other hardware description languages (HDLs), such as very high speed integrated circuit (VHSIC) hardware description language (VHDL) and Verilog, SystemC provides the means to express mandatory hardware semantics, such as modularity, concurrency, and synchronization (Panda, 2001; Grotker et al., 2002). In addition, it is possible to employ all the advanced descriptive functionalities offered by the C++ language: classes, objects, virtual function, polymorphism, templates, etc. (Stroustrup, 2000). In this way SystemC is able to provide powerful

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 43–59.

abstraction mechanisms, which are very suitable for a high-level modelling such as system level or transaction-level modelling (TLM) (Grotker et al., 2002; Pasricha, 2002; Jayadevappa and Shankar, 2004). In fact, at these levels a major importance is given to the functional aspects; implementation details are not yet taken into account and then cannot impose strict constraints on the modelling possibilities. This allows to treat the system to be modelled in a quite abstract and parameterizable way.

Nowadays, high-level modelling is assuming a more and more important role in the design of the modern electronic systems, because of the continuous increase in their complexity. One of the most important abstraction mechanism supported by SystemC is given by the uncoupling between module implementation and its external communication. This is carried out by developing communication through the actions involved in this task (for example, read or write operation), rather than through the accurate modelling of the input/output (I/O) physical features. At code level this is achieved through an interface class, which contains a set of virtual functions standing for the main actions associated with a certain communication modality. This modelling strategy is usually referred as interface method call (IMC) (OSCI, 2006) and represents a concrete realization of a powerful communication paradigm called interface-based design (Rowson and Sangiovanni-Vincentelli, 1997), which favours code reuse and modelling effort. The SystemC environment already provides interface classes for some well-known communication modalities (hardware signal, first-in first-out semaphore, etc.), together with default implementations. However, an user can define new interface classes and provide his own implementations for his interface classes and also for those already provided by SystemC.

In the past few years, several researches have been carried out in order to improve and optimize the communication capabilities provided by SystemC (Caldari et al., 2003; Coppola et al., 2003, 2004). However, up to now the IMC strategy has been typically applied to realize communication at the physical level, i.e., the effective communication between modules and channels physically separated. In this chapter, we present a modelling technique based on a logical extension of the IMC strategy and suitable for the high-level modelling of systems with a constant component in their behaviour. In particular, we will consider the case of devices bound to a working protocol. We have called the proposed technique *behaviour separation*, and we have applied it for modelling master devices related to advanced microcontroller bus architecture, advanced high-performance bus (AMBA AHB) communication protocol (ARM, 1999). It will be be shown how this approach offers the same benefits of the IMC strategy in its typical application for physical communication, favouring code reuse and easing the modelling effort.

In order to fix the ideas, we will show the features of this methodology through its application for the modelling of devices bound to a communication protocol, considering the concrete case of AMBA AHB master devices. However, we will also provide some guidelines for extending the application of behaviour separation for modelling devices bound to a generic working protocol.

2. Principles of the Behaviour Separation Methodology

First of all we will explain some theoretical principles on which behaviour separation is based. When modelling a system through a suitable language, we must describe all the rules that establish the system behaviour, according to the adopted abstraction level. From an abstract point of view, we can imagine to represent the overall behaviour of a system through a set that contains all the rules which establish the system behaviour.

The devices bound to a working protocol have to show a behaviour compatible with such a protocol. Moreover, more general devices may carry out some further tasks not related to the associated working protocol. Thus for this kind of devices, the set can be subdivided into two complementary subsets. One subset contains the working rules related to the protocol, whereas the other subset contains the working rules related to the other other possible tasks. Moreover, it is possible to further subdivide the former subset into two complementary subsets, as shown in Figure 3.1.

In fact, we can distinguish two categories as regards the protocol rules. In particular, there are protocol rules fixed by the the protocol in a strict and univocal manner. All the devices bound to a specific protocol have to reproduce these rules always in the same way. These rules are typically known a priori, depending only on the protocol specification and not on the intrinsic features of the devices to be modelled. We can refer to these rules as *fixed protocol rules* or *fixed protocol behaviours*.

However, there are other protocol rules that are not univocally fixed by the protocol and partially depend on intrinsic device features. In general, different devices bound to a same protocol can realize these rules in different ways. We can refer to these other rules as *free protocol rules* or *free protocol behaviours*.

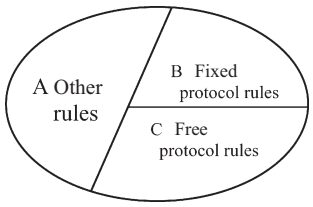


Fig. 3.1 Protocol rules classification in fixed and free protocol rules

In Figure 3.1, each of these subsets has been associated to an alphabetical letter. The subset *A* contains the working rules that are not related to protocol; the subset *B* contains the rules related to protocol and univocally fixed, that is fixed protocol rules; finally the subset *C* contains the rules related to protocol but not fixed, that is free protocol rules.

In order to explain this classification, let us consider the concrete case of a master device bound to the AMBA AHB communication protocol. In particular, let us consider some typical working situations. For example, when granted access to the AMBA bus, a master device always has to declare the features of a new transfer right from the start, without modifying these features during the transfer. Moreover, if the selected slave device replies with an ERROR response, a master device always has to set HTRANS signal to IDLE (ARM, 1999). All these behavioural rules are examples of the rules univocally fixed by the AMBA AHB protocol. So every master device has to reproduce these rules always in the same way. These rules are examples of fixed protocol rules.

Now let us consider other working situations. For example, there are no impositions on the moment when a master device can start a new transfer, when it is in IDLE state. Moreover, when a new transfer starts, the features of such transfer are exclusively set by the master device, with no imposition coming from the protocol in this case also. As we can see, these other rules are still related to AMBA AHB protocol, but are not fixed by the protocol. These other rules really depend on the intrinsic features of a master device, and different master devices could realize these rules in different ways. Hence these other rules are examples of free protocol rules.

Behaviour separation methodology describes the device behaviour taking into account this characterization, i.e., describing the device behaviour so as to separate these subsets of rules. In this way, it is possible to achieve some modelling benefits due to the properties of these subsets. In particular, for a specific working protocol, the fixed protocol rules (subset *B*) represent a constant behavioural component, since these rules are univocally established once and for all. So if it were possible to describe these rules in a separate and distinct way, this description would be valid for every device bound to the considered protocol. This description could be made once and for all, and the related code could be reused every time a new device has to be modelled. In this way, when modelling a new device, only the behavioural rules related to the subsets *A* and *C* should be described.

The code reuse for fixed protocol rules can be very significant, as it will be shown from the experimental results. Moreover, it is also possible to achieve some facilitations as regards the description of free protocol rules. All these aspects could provide an important contribution for making the overall modelling effort easier. In particular, this is true for high-level descriptions (system level

or transaction level), which represent the most suitable application context for the behaviour-separation approach.

At this point the question is how to realize a device description based on behaviour separation. The applicability of this approach depends on the language used for the modelling purpose. In other words, it is necessary that the used language provides modelling constructs suitable for realizing the behaviour-separation idea. Anyway these constructs are available in SystemC. Before examining the realization in SystemC, it is necessary to introduce some other features concerning architectural aspects.

3. Application for Communication Protocols

In this section, we will focus on the application of behaviour separation for devices bound to a communication protocol. However, in Section 8, we will provide some guidelines for extending this approach to devices bound to generic protocols.

Now let us consider a device bound to a communication protocol. Typically, the rules related to this kind of protocol are expressed by the relationships between the I/O signals referred to the protocol. In general, such device may be characterized by the I/O signals related to the protocol and further possible I/O signals. These further I/O signals may be present for implementing some free protocol rules and possible tasks not related to the protocol.

For applying behaviour separation, we could model the device through the architecture in Figure 3.2 where the device has been split into two complementary units called *bound unit* and *free unit*.

Bound unit should be a module that manages the whole behaviour related to the protocol. For this reason, only the bound unit should establish the run-time

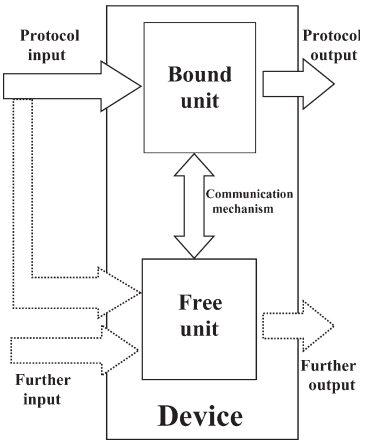


Fig. 3.2 Generic architecture for communication protocol–bound devices

progress for the output protocol signals, as reported in Figure 3.2. At the same time, bound unit should be characterized by a fundamental property in order to realize the behaviour-separation idea; it should explicitly implement only the fixed protocol rules, and should not depend on the real implementation of the free protocol rules. If this were true, the bound unit would really define only the protocol's constant component. All that could allow us to define the bound unit once and for all and to reuse its code for modelling different devices related to the same protocol.

Bound-unit processes should manage all the working phases related to the protocol. This means that such processes should realize both the fixed protocol rules and the free protocol rules. The fixed protocol rules must be explicitly described by bound-unit processes. For a communication protocol, the fixed protocol rules may depend only on the input signals related to the protocol. For this reason, according to the architecture in Figure 3.2, bound unit is connected to the protocol input signals.

In contrast, the free protocol rules must be handled by the bound unit in a different way. As we have explained, these rules are not known a priori and represent a variable component as regards the device behaviour related to the protocol. These rules are not explicitly implemented by the bound unit, but are handled in an implicit way. More precisely, the free protocol rules must be implemented inside the free unit but their execution can be required by bound unit. In particular, bound unit must be able to invoke the execution of free protocol rules from free unit during a simulation. This should happen when free protocol rules have to be executed in order to reproduce the correct run-time behaviour related to the protocol. For this purpose, as shown in 3.2, it is necessary to define a suitable communication mechanism between bound unit and free unit, which allows the invocation of free protocol rules.

With regards to the free unit, this component should realize the behavioural part that depends on the specific device features. In particular, the free unit must provide a suitable implementation for the free protocol rules and for all the possible tasks not related to the protocol. In practice, the free unit should implement the rules for the subsets A and C reported in Figure 3.1. For this purpose the free unit could need the access to all input signals, both protocol and further signals. Moreover, in order to implement the possible tasks not related to the protocol, the free unit has to establish the run-time progress for further output signals.

The architectural characterization we have now explained is abstract, with no references to a concrete realization through a particular language. In section 4 we will show how this architecture can be concretely realized in SystemC, emphasizing the communication mechanism between bound unit and free unit in particular.

4. Realization in SystemC

The architectural characterization through bound and free units can be realized in SystemC in a natural and flexible way (see Figure 3.3). In particular, this is possible through an extension of its IMC capabilities (OSCI, 2006). More precisely, we can define an interface abstract class for the communication between bound and free free unit (Grotker et al., 2002; OSCI, 2006). This class inherits from SystemC class `sc_interface` and contains a set of pure virtual methods. These interface methods realize the various free protocol rules and at the same time provide a mechanism for invoking these rules.

These interface methods must be defined by the free unit; so free unit has to be an implementation class inheriting from the interface class (Stroustrup, 2000). However, free unit could need the access to I/O signals and could be described through its own internal processes. As a consequence, in the most general case, free unit may be defined through an `sc_module` class (Grotker et al., 2002; OSCI, 2006) inheriting from the interface class.

On the other hand, bound unit must be an `sc_module`, which is connected to the protocol I/O signals and is described through suitable internal processes. Such processes should model all the working phases related to the protocol. At code level the fixed protocol rules are explicitly defined, whereas the free protocol rules are reported through the invocation of the interface methods. According to the typical SystemC approach, the invocation of these methods may be carried out through an `sc_port` object (OSCI, 2006) linked to the interface class. In this case, this `sc_port` does not match with a real physical port, but rather represents an object suitable for interacting with the interface methods.

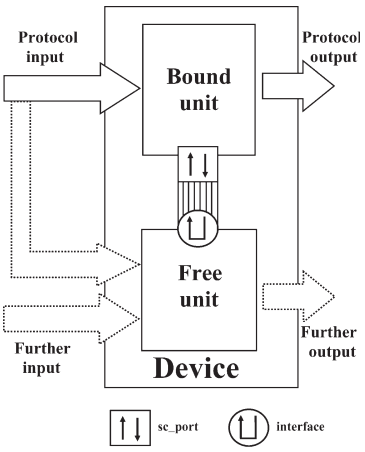


Fig. 3.3 Device architecture in the SystemC environment

An important aspect is that the bound unit does not depend on the implementation of the interface methods. In fact, this implementation is hidden inside the free unit, and the bound unit simply invokes the interface methods through their standard names, without knowing their real definition. All that leads to a separation between bound unit and free unit at code level. In this way, bound unit can be connected with different free units based on the same protocol interface. At the same time, it is possible to reuse the bound-unit code for modelling different devices related to the same protocol. In fact bound unit implements only the constant part of the protocol behavior, that is the fixed protocol rules. So, when modelling a new device, it could be necessary to define only a suitable free unit considering the specific device features.

The proposed SystemC realization is based on the same IMC technique typically used in SystemC for physical communication. In the typical application of IMC there are two main subjects: (1) an `sc_module`, which communicates by invoking the methods of a specific communication interface; (2) a channel, which is connected to this module and provides a specific definition for these methods. In behaviour separation, bound unit plays the role of the `sc_module` whereas free unit plays the role of the channel. In other words, this approach is based on an alternative application of the IMC capabilities in order to model the internal behaviour of devices bound to a working protocol. In these sections, we have considered the communication protocols; however, in 8, we will deal with the the extension for devices bound to a generic protocol.

5. Application Example Based on an AMBA AHB Master Device

In this section we consider a concrete application of the behaviour-separation approach, for the modelling of AMBA AHB master devices (ARM, 1999).

First of all we have executed an accurate analysis on the AMBA AHB protocol specification, in order to distinguish between fixed and free protocol rules. Through this examination we have found out all those aspects of the AMBA AHB protocol that are not univocally fixed, i.e., the free protocol rules. Then we have defined an interface class containing a set of pure methods (Stroustrup, 2000), that model such free protocol rules. This interface class is called `AMBA_Free_If`:

```
1 class AMBA_Free_If : public sc_interface
2 {
3 public:
4   virtual notransfer()=0;
5   virtual startransfer()=0;
6   virtual transferInfo()=0;
7   virtual getdata()=0;
8   virtual putdata()=0;
```

```
9     virtual trans_type transnow()=0;
10    virtual anothertransfer()=0;
11    virtual firstaddwrap()=0;
12    virtual endincr()=0;
13    virtual subtransfer()=0;
14    virtual actionposterror()=0;
15  };
```

For simplicity reasons, we have reported the virtual methods (lines 4–14) omitting their return values and their possible input parameters. These methods are involved in some typical working phases related to an AMBA AHB master device: bus request; generation and handling for reading and writing data; transfer features declaration; address handling for nonincremental transfers.

Then we have defined an `sc_module` for bound unit. We have realized this component through a high-level modelling, based on an algorithmic description. However, we have maintained a accurate (CA) behaviour (Grotker et al., 2002) through an exact timing triggering with respect to the input signals:

```
1   class BOUND_unit : public sc_module
2   {
3     // I/O ports and auxiliary objects, not shown
4     ...
5     // internal processes
6     void bus_request();
7     void bus_granting();
8     void first_transfer1();
9     void first_transfer2();
10    void loop_transfer();
11    void last_transfer();
12    void exception_manager();
13    void reset_manager();
14    // port for communication with free_unit
15    sc_port<AMBA_Free_If> Free;
16    // constructor and other components, not shown
17    ...
18  };
```

Inside the bound unit there is a set of processes that realize all the working phases related to AMBA AHB protocol (lines 6–13). In our implementation these processes are all `sc_thread`; their triggering is due to clock, asynchronous inputs, and some `sc_event` objects (OSCI, 2006). Besides, we can also notice an `sc_port` object called `Free`, and linked to `AMBA_Free_If` class (line 15). This `sc_port` is used by the bound unit to invoke the execution of the interface methods, in order to manage the free protocol rules.

At this point, in order to complete the modelling of a device, a final step is necessary: defining a suitable free unit class, which must provide an implementation for all the methods of `AMBA_Free_If` interface class.

Now we can consider the concrete application of behaviour separation at code level. For this purpose, we will examine the code of one of the bound-unit processes, i.e., the `bus_request` process:

```
1  void BOUND_unit::bus_request()
2  {
3    transfer_type trinf_tmp;  // auxiliary object
4    ...   //initialization instructions, not shown
5    while(true) {
6      if(HRESET) {
7        if(Free->startransfer()) {
8          HBUSREQ = true;
9          HTRANS  = NONSEQ;
10
11         trinf_tmp = Free->transferInfo();
12         HADDR     = trinf_tmp.firstaddr;
13         HWRITE    = trinf_tmp.write;
14         HSIZE     = trinf_tmp.size;
15         HLOCK     = trinf_tmp.lock;
16         HPROT     = trinf_tmp.prot;
17         HBURST    = trinf_tmp.burst;
18         ...     //setting of some auxiliary objects, not shown
19         notify(SC_ZERO_TIME,checkbusgranted);
20         wait(checkbusrequested);
21       } else {
22         wait(HCLK.posedge_event()|HRESET.negedge_event());
23         wait(SC_ZERO_TIME);
24         if(!HRESET.read()) {
25           ...   //reset condition handling, not shown
26         }
27       }
28     } else {
29       ...       //reset condition handling, not shown
30     }
31   }
32 }
```

The `bus_request` process manages the working phase before a new transfer starts. This process is triggered by the positive clock edges and by the negative edges of `HRESET` input signal (ARM, 1999). Let us consider the execution steps in normal conditions, neglecting the case of reset events.

At the beginning the interface method `startransfer` is invoked on the `sc_port Free` (line 7). This method returns a boolean value and its actual implementation is hidden inside free unit. `startransfer` is invoked in order to know if a new transfer has to be started. This protocol aspect is not fixed by

the AMBA AHB protocol and hence represents a free protocol rule. In fact, a master device can start a new transfer in any moment during this phase with no imposition from the protocol. The `startransfer` method is repeatedly invoked until its return value is **`true`**. When this happens, it means the master device wants to start a new transfer. Then the output signal `HBUSREQ` must be set to **`true`** and the output signal `HTRANS` to `NONSEQ` (lines 8–9). These two assignment instructions are univocally specified by the protocol in this working situation (ARM, 1999). Thus these instructions represent two fixed protocol rules and are explicitly defined in the code.

Then the setting of all these output signals, which communicates the features of the new transfer (lines 12–17), is carried out. The setting of these output signals is achieved by invoking another interface method, that is `transferInfo` (line 11). This method returns a `transfer_type` object; `transfer_type` is a C++ struct whose internal members can represent the values for the several protocol output signals. The `transferInfo` method returns a `transfer_type` object, which is assigned to an auxiliary object, called `trinf_tmp`; through such object the output signals are finally set. In this situation the values to be assigned to these output signals are not fixed by the AMBA AHB protocol. Thus this represents another free protocol rule, which is realized by the interface method `transferInfo`.

After this step the run-time execution is moved to another process, i.e., `bus_granting`. This process deals with the next working phase, and is triggered by the notification of the `sc_event checkbusgranted` (line 19). At this point the `bus_request` process stops. According to the final `wait` instruction (line 20) (OSCI, 2006), this process will be resumed when the `sc_event checkbusgranted` is invoked by another process of bound unit.

In the `bus_request` process we have seen a concrete application of behaviour separation at code level. The other processes of bound unit are implemented in the same way too. In particular, we have seen the description of a typical phase characterized by fixed and free protocol rules. The fixed and free protocol rules are reported in the code in an interleaved way, according to the execution scheduling dictated by the protocol. Fixed protocol rules are explicitly defined, whereas free protocol rules are reported through the invocation of the interface methods. In other words, this code implements only the fixed protocol rules, that is the protocol's constant component. In this way, when modelling a new master device, the code of the fixed protocol rules is already available through the bound-unit definition. At the same time there are also some facilitations for defining the free protocol rules, which can be described in a helped and driven way by considering the semantics of each interface method separately.

Now we can briefly summarize the necessary steps to apply behaviour separation for modelling a device bound to a certain working protocol:

1. Examining the protocol specifications and distinguishing between fixed and free protocol rules

2. Defining an interface class whose virtual methods model the found free protocol rules

3. Defining a suitable bound unit unit class that models the device behaviour related to the protocol; fixed protocol rules are explicitly described whereas free protocol rules are described by invoking the interface methods.

4. Examining the working features of the device to be modelled and understanding the specific realization of the free protocol rules

5. Defining a class for free unit, which inherits from the interface class and contains a suitable implementation for the interface methods

6. Defining the device behaviour not related to the protocol, that is the further possible tasks carried out; for a communication protocol this can be made inside the definition of free unit, even if, in general, it is possible to follow any solution valid in the SystemC environment.

The first three steps depend only on the protocol specification and hence can be made once and for all. On the contrary, the steps 4–6 depend on the device's intrinsic features and their realization is not univocally fixed. So, when modelling new devices bound to the considered protocol, only the steps 4–6 could be required.

6. I/O Adaptation; Limitations and Application Fields

For a specific communication protocol, bound unit should be defined once and for all and its I/O ports should agree with the I/O signals related to such protocol. This also means the types associated to these ports should be specified once and for all, considering that C++ is a strongly typified (Stroustrup, 2000). As far as possible it would be better to associate these ports to quite abstract and high-level types, without references to implementation details not imposed by the protocol.

However, when using a device description in different application contexts, in general based on different abstraction levels, there could be some incompatibilities with the types associated to the ports of the modules externally connected. In this case it is not necessary to define new bound-unit modules, which match with the required types for the I/O ports. In fact, it is possible to apply the typical SystemC solution for this kind of problem, that is adapter modules

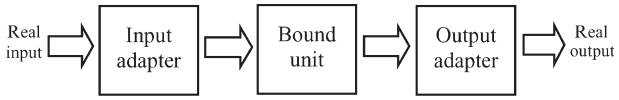


Fig. 3.4 Bound unit I/O adaptation

(Grotker et al., 2002; OSCI, 2006). By using two I/O adapter modules, bound unit should be instantiated with the structure in Figure 3.4.

In this way all the features and benefits previously described are still valid. If there were I/O type mismatching within a particular application context, it could be necessary to define only the suitable I/O adapter modules, reusing the same bound-unit core.

The proposed modelling technique could prove to be favourable in several applications, even if there are some limitations to be considered.

Probably, the main limitation is the fact that the separation of the protocol behaviour, and hence the separation in bound unit and free unit, is a purely logical characterization. So there is no matching with the physical structure of the real device, and this may make this approach unsuitable for direct synthesis. This means this modelling methodology may not be suitable in a context where synthesis is an important target. In the design flow, this approach could be applied in the early phases, for system level modelling and analysis. In fact, in such context physical details are usually not yet set and the main target is evaluating system behaviour. Moreover, this approach may also be applied for research applications. For example, we have applied behaviour separation for modelling particular AMBA AHB master devises, used to evaluate some performances concerning the AMBA AHB protocol (Conti et al., 2004; Vece et al., 2005).

So, in general, this modelling methodology can be taken into account in all those cases in which we want to reproduce the behaviour of a device, without being strictly interested in its physical implementation.

7. Modelling of Complete AMBA AHB Master Devices and Results

So far, we have seen the definition of a bound unit and an interface class for the AMBA AHB protocol. In order to achieve a complete device description, it is necessary to define a suitable class for free unit.

For our researches (Conti et al., 2004; Vece et al., 2005) we have realized several free-unit classes, which provide different implementations for the virtual methods of the `AMBA_Free_If` interface class. In this way we have modelled several master devices, characterized by a variable behavioural complexity.

Table 3.1 Code data for different master devices

Master behaviour features	*Free unit code (kB)*	*Bound unit code (kB)*	*Total code (kB)*	*Code reuse (%)*
Deterministic; not parameterizable	7	33	40	82
Pseudorandom; parameterizable	17	33	50	66
Pseudorandom; parameterizable; feedback for power management	23	33	56	58

The simplest devices have been based on a static and deterministic behaviour. In this case free protocol rules are static and do not depend on the device's input signals. On the contrary, the most complex devices have been developed through a parametric and pseudorandom behaviour, in which free protocol rules are not static but depend on input signals and also on some random variables. The statistics of these random variables can be set at the beginning of a SystemC simulation. Moreover, some of such devices are also based on a feedback mechanism for power optimization.

After realizing these devices we carried out several tests, in order to verify their compatibility with the SystemC environment and also their behaviour. This testing phase gave good results. The particular architecture of these devices did not cause rejections or limitations neither at compilation time nor at run-time. Also the expected behaviour was successfully confirmed by the simulations. In (Vece et al., 2005), it is possible to find a detailed description about this testing phase.

Now we can evaluate the modelling efforts spent to realize these device descriptions. For this purpose, we can consider the amounts of code required for realizing these devices. However, we should really consider the code for bound unit and free unit separately. In fact the real modelling effort was given only by the realization of free unit. For each device we have always used the same bound-unit code.

In Table 3.1 we have reported the amounts of source code for three different master devices. The values are expressed in kB units. The amount of code for bound unit is constant for each device and is equal to 33 kB.

As we can notice, the reuse of bound unit determines a significant saving in the code required to model a complete device. This data is especially interesting for the last two devices considered in Table 3.1, which are characterized by a quite complex behaviour and require a higher modelling effort.

8. Extension to Generic Protocols

Now, as final point, we will briefly explain how this modelling approach could be extended for devices bound to generic protocols.

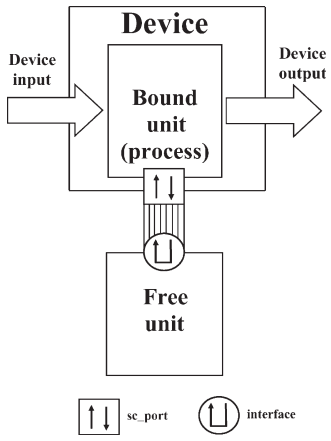


Fig. 3.5 Device architecture for protocols concerning internal behaviour

The characterization through fixed and free protocol rules is always valid. So we can always define an interface class for free protocol rules. Also bound unit and free unit should maintain their reciprocal roles. What may change is the architecture.

In the previous sections we have considered devices bound to a communication protocol, in particular the concrete case of AMBA AHB master devices. Now let us consider a device bound to another kind of working protocol. For example, we could consider a protocol that establishes the internal device behaviour, without input and output ports related to it; that could be the case of an intermediate protocol in the Bluetooth protocol stack. In this case, we should refer to a new architecture for applying separation. In particular, the architecture shown in Figure 3.5.

In this case bound unit is not an `sc_module` any more. Now bound unit is an internal process or also a set of internal processes. These processes describe all the protocol behaviour and execute free protocol rules by invoking them from a free unit class. Also in this case the invocation could be made through a suitable `sc_port` object. On the other hand, free unit is still defined as an implementation class for free protocol rules. In this case free unit needs not to be connected to I/O ports, and can be instantiated as an external object. Also in this case there are the same benefits seen for the devices bound to a communication protocol. In particular, it is still possible to reuse bound-unit code, that is the code of the processes related to bound unit.

This way to act can be intuitively generalized for other kinds of working protocols. More precisely, for modelling a device bound to a particular kind of protocol, the application of behaviour separation could require a customized architectural characterization. However, we can maintain all those features

concerning the classification in fixed and free protocol rules, the use of the IMC communication technique, and the benefits for code reuse.

9. Conclusions

In this chapter, we have explained the features of behaviour separation, a modelling methodology suitable for devices bound to a working protocol. We have shown how this approach can be realized in a suitable way in a SystemC/C++ environment. That is possible through the modelling capabilities of this environment, and in particular through an extension of the IMC technique. We have highlighted the benefits and limitations of this approach on the basis of the application contexts. Moreover, a concrete application has been considered, concerning the AMBA AHB master devices. This application has shown a quite significant code reuse and some facilitation for the modelling effort.

References

ARM (1999) *AMBA 2.0 Specification*. ARM Ltd. `http://www.arm.com/products/solutions/AMBA_Spec.html`.

Caldari, M., Conti, M., Coppola, M., Curaba, S., Pieralisi, L., and Turchetti, C. (2003) Transaction-level models for AMBA bus architecture using SystemC 2.0. In: *Proceedings of Design, Automation, and Test in Europe (DATE) 2003*. Munich, Germany, pp. 26–31.

Conti, M., Caldari, M., Vece, G. B., Orcioni, S., and Turchetti, C. (2004) Performance analysis of different arbitration algorithms of the AMBA AHB bus. In: *Proceedings of the 41th Design Automation Conference (DAC) 2004*. San Diego, CA, pp. 618–821.

Coppola, M., Curaba, S., Grammatikakis, M., and Maruccia, G. (2003) IPSIM: SystemC 3.0 enhancments for communication refinement. In: *Proceedings of Design, Automation, and Test in Europe (DATE) 2003*. Munich, Germany, pp. 106–111.

Coppola, M., Curaba, S., Grammatikakis, M., Maruccia, G., and Papariello, F. (2004) OCCN: a network-on-chip modeling and simulation framework. In: *Proceedings of Design, Automation, and Test in Europe (DATE) 2004*. Paris, pp. 174–179.

Grotker, T., Liao, S., Martin, G., and Swan, S. (2002) *System Design with SystemC*. Kluwer Academic Publishers, Boston, MA.

Jayadevappa, S. and Shankar, R. (2004) A comparative study of modelling at different levels of abstraction in system on chip designs: a case study.

In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI 2004*, IEEE, pp. 52–58.

OSCI (2006) *SystemC Community Homepage*. Open SystemC Iniative (OSCI). `http://www.systemc.org`.

Panda, P. R. (2001) SystemC: a modeling platform supporting multiple design abstractions. In: *Proceedings of the 14th International Symposium on System Synthesis 2001*, pp. 75–80.

Pasricha, S. (2002) Transaction level modeling of SoC with SystemC 2.0. In: *Synopsys User Group Conference (SNUG) 2002*, Bangalore, India.

Rowson, J. A. and Sangiovanni-Vincentelli, A. (1997) Interface-based design. In: *Proceedings of the 35th Design Automation Conference (DAC'97)*, pp. 178–183.

Stroustrup, B. (2000) *The C++ Programming Language*, 3rd edn. Addison-Wesley, Boston, MA.

Vece, G. B., Conti, M., and Orcioni, S. (2005) Devices modeling in systemc based on behaviour separation. In: *Proceedings of the Eighth International Forum on Specification and Design Languages (FDL) 2005*. ECSI, Lausanne, Switzerland, pp. 245–256.

Chapter 4

Mixing Synchronous Reactive and Untimed MoCs in SystemC

Fernando Herrera and Eugenio Villar
University of Cantabria
E. T. S. I. Industriales y de Telecomunicación
Avda. Los Castros s/n, 39005
Santander
Spain

Abstract The support of heterogeneity at the specification level, i.e., the ability to mix several models of computation (MoCs) in the system-level specification, is becoming increasingly important in system-level design methodologies of hardware/software (HW/SW) embedded systems. It presents several advantages. At the modeling level, it enables a more natural description that can be more efficiently simulated. In addition, it can ease the automation of the design flow over a heterogeneous target platform. This work is in the context of the development of a heterogeneous system-level specification methodology based on SystemC. The methodology is able to support untimed MoCs (such as process network (PN), Kahn process network (KPN), and communicating sequential process (CSP)) and MoCs with a more detailed handling of time, such as the synchronous reactive (SR) MoC. The problem of MoC interfaces has been addressed and specifically solved for untimed–untimed interfaces. In this chapter, this work is extended with the connection between untimed MoCs and the SR MoC. This connection involves the intersection of different MoC restrictions in the time domain. The way in which the untimed–SR interface specifies how the untimed events map onto the SR time domain is shown. These general concepts are reflected later in the SystemC untimed–SR interfaces, consisting of border processes and channels. The incompatibilities between time restrictions provoked or transmitted by the connection are also shown, as well as the way these are detected in SystemC. Previously, a study of the time semantics of the untimed and SR MoCs and how MoCs assumptions are abstracted from the time model of SystemC are presented.

Keywords: heterogeneity; system-level specification; models of computation; SystemC.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 61–81.

1. Introduction

The effort in the development of more productive methodologies for hardware/software (HW/SW) embedded systems is narrowing the design gap provoked by the increasing integration capability and thus complexity. This complexity enables the implementation of a whole system in a single chip (System-on-Chip or SoC; Chang et al., 1999). One of the main reasons for this productivity rise is the increase of abstraction level at the system specification. The design starts by constructing a system-level specification, which incorporates more abstract primitives, with a more powerful semantic content and less dependent on the implementation architecture. SystemC (OSCI, 2001; Grötker et al., 2002; Müller et al., 2003) has gained the confidence of many users (more than 36000 registered) as a system-level specification language. It provides important advantages, such as, being based on C/C++, support for SW modeling, HDL description, and system-level specification. This eases the adoption of the language by embedded SW developers and HW designers and the transition to a system-level specification style. In addition, it is an open-source project, which also provides extension features as part of the language itself. This facilitates the incorporation of an important feature in the specification methodology of HW/SW embedded systems: heterogeneity.

In this context, heterogeneity is the ability to mix (include and communicate) several models of computation or MoCs (such as synchronous data flow (SDF), finite state machine (FSM), kahn process network (KPN), register-transfer level (RTL); Lee and Sangiovanni-Vincentelli, 1998) in the same system specification. We name this heterogeneity at specification-level (marked in bold in Figure 4.1), in contrast to heterogeneity at the implementation level, which expresses the presence of many types of implementation architectures or technologies (application-specific integrated circuits (ASICs), digital signal processing (DSPs), application-specific instruction set processors (ASIPs) general-purpose processors, etc.) in the target platform.

Complexity and heterogeneity in the implementation platform are strongly related phenomena and usually occur together; thus a methodology for the design of such complex embedded systems has to provide abstraction and heterogeneity at the specification level. While abstraction enables confronting the specification with powerful primitives (in the semantic sense), heterogeneity at the specification level reflects the heterogeneity of the implementation platform.

There are several advantages in a system-level heterogeneous specification methodology:

- The specifier can build the specification more naturally. A description style suited to the functionality that must be specified in each part of the specification can be used.

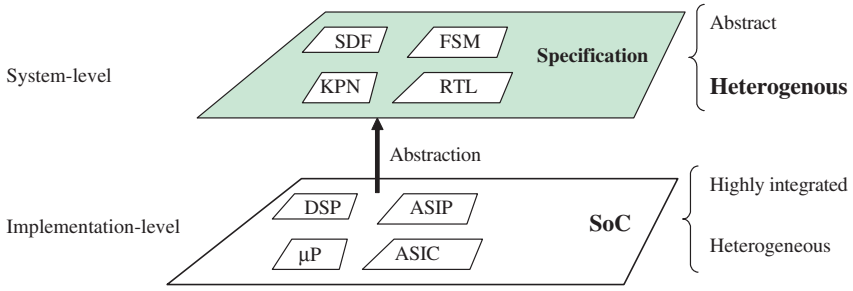


Fig. 4.1 Heterogeneity and abstraction in the specification

- The simulation speed is improved since the simulation engine adapts to the detail level that the semantics of MoC primitives requires.
- The benefits provided by the different MoCs are maintained. For example, properties such as determinism in SDF, KPN, SR, and CSP or static schedulability in SDF.
- The automation of synthesis and generation flows, thus the mapping to a heterogeneous target platform, is easier. For example, a cycle-based MoC, such as RTL, presents a description style more suited to the implementation flow of custom HW devices of the HW/SW platform.

As a consequence, heterogeneous system specification has gained an increasing interest from the research community. In Lee and Sangiovanni-Vincentelli (1998), a theoretical framework for the study and comparison of MoCs is proposed. It describes processes as a set of behaviors and a behavior as a set of signals. Process composition is the intersection of their behaviors. The interaction between processes is through signals, a collection of events. Each event is a value-tag pair, where the tag comes from a partially or totally ordered set. This permits useful and objective considerations in one of the most important features of an MoC: the handling of time. It defines a timed MoC as that where the tag set is totally ordered. It also defines synchronous events as those with the same time tag. Then, it defines synchronous signals as those having only synchronous events and synchronous processes as those having synchronous signals as behaviors. In Jantsch (2004), a useful tool called Rugby metamodel enables the study of MoCs through four separate domains: computation, communication, data, and time. Hierarchy is independently considered. Each domain represents a feature of the model that can be independently treated at a specific detail level and a classification of MoCs can be done just by considering the coordinates in the Rugby metamodel. In addition, a classification classification of MoCs considering one of the most

distinguishing domains, time, is done. Untimed MoCs, with less-restrictive time assumptions, are distinguished from synchronous and timed MoCS, the latter begin the most restrictive ones in time domain (see Figure 4.7).

In addition to theoretical frameworks, there are also frameworks for heterogeneous specification. One of the most known and influencing ones is Ptolemy (Ptolemy, 2006). It comprises a component-based graphical specification framework, which establishes a match MoC-component, clearly separating MoCs at the specification level (hierarchical heterogeneity). This work is mainly intended for modeling and each MoC is supported over its own execution framework (simulation engine), taking Java as the implementation language.

Efforts have been taken to incorporate heterogeneity in C-based specification languages and, specifically, in SystemC. Moreover, heterogeneous specification in SystemC requires the development of an appropriate specification methodology.

In Patel and Shukla (2004), a way to provide heterogeneity support for SystemC is shown. The extension of the SystemC discrete event (DE) strict-timed kernel in order to provide support to the MoC (Lee and Messerschmitt, 1987) is proposed. It is an approach similar to Ptolemy Ptolemy in that each supported MoC has its corresponding simulation kernel in charge of the execution of an efficient simulation of that part of the system. It can reach a simulation speedup around 75% in stand-alone SDF specifications and may admit the incorporation of any MoC, including analog MoCs. Currently, the support of more MoCs, such as CSPs (Hoare, 1978) or FSMs, is being explored. The main drawback of the approach is that it requires a nonstandard SystemC simulation kernel. In addition, as other heterogeneous specification methodologies, simulation speedups are affected by Amdahl's law (Michalove, 2006) when more demanding MoCs (in terms of simulation time) are also present in the specification.

In Herrera et al. (2004, 2005), a SystemC specification methodology was proposed with the goal of an efficient support of heterogeneity. It is supported by the theoretical frameworks presented, such as the time tag framework of Lee and Sangiovanni-Vincentelli (1998), extended in Villar (2002), or the Rugby metamodel and the MoCs taxonomy of Jantsch (2004), used in Herrera et al. (2005) for studying and providing MoC interfaces. The proposal has several distinguishing features. Firstly, it provides support for a single-source system-level design flow (Posadas et al., 2005). This design flow starts from a specification attending a general specification methodology. This is a first level of basic rules and methodological guidelines. Guidelines are general in the sense that they reflect the main restrictions and targets of the specification methodology, still letting some flexibility to the specifier. The most important ones are the focus of the design task on conceiving a specification of flat concurrent

functionality enclosed in a hierarchy of modules and a clear separation between communication and computation. The specifier basically fills the process computation by writing its functionality and channel accesses. The rules dictate which SystemC primitives (`SC_THREAD`, `sc_fifo`, etc.) to use and how (i.e., do not directly use wait on event in the computation code) to use them. In addition, a library, categorized in SystemC as methodology-specific library (OSCI, 2006, 2001), provides new elements and, therefore, features, which enable the extension of the language over the SystemC standard library in a core-decoupled way. For example, the library includes classes and macros for easing the debug of the concurrent specification. The implementation of these new elements (MoC channels, MoC interface channels, checking elements) is written using the public elements provided by SystemC, which explains why no kernel extension is needed.

Furthermore, heterogeneity support is obtained in this methodology by the application of the same philosophy. Heterogeneity is achieved through an extra level of rules, attending and eventually extending those of the general specification methodology, and through a set of new elements added to the specification library. Since these new elements become a majority in the specification library, the library may be assumed to be a heterogeneous specification library. Most of the new elements are nonstandard channels, such as the `uc_rv` `uc_rv` rendezvous channel; therefore it can also be understood as a new communication library. The methodology ensures the support of those MoCs that may be comprised through DE strict-timed model of the simulation kernel. Untimed MoCs (PN, KPN (Kahn, 1974)), CSPs, and the SR model of computation are the SR model of computation are supported. This methodology does not preclude heterogeneous specification based on multiple simulation kernels when it is required. This would be the case for simulations since the DE kernel might not be able to handle analog able to handle analog simulation efficiently. This would require clear temporal execution semantics. MoC interfaces are supported border processes and channels, provided as a general solution for solution for interfaces between different MoCs. In Herrera et al. (2005), untimed–untimed MoC interfaces are already dealt. Some questions, such as the support of untimed–SR, untimed–timed interfaces, and the temporal interpretation of events when these interfaces are present in the specification, are not covered yet.

In this chapter, the support of heterogeneity of the methodology is extended with the connection between the SR MoC and untimed MoCs KPN, untimed MoCs (KPN, PN, and CSP cases). The temporal semantics and effects of the MoC interface over the events of the specification have been analyzed. SR–untimed MoC interfaces are provided in the form of border processes and border channels. They include automatic checks for the fulfillment of MoC rules in a transparent of these interfaces requires the understanding of how each MoC is

supported by the standard kernel and the common points and incompatibilities between SR and untimed MoCs.

The structure of the chapter is as follows. In 2, the support of the SR MoC, showing and contrasting how SR and untimed MoCs are abstracted from the SystemC time axis, is explained. In Section 3, general concepts for the MoCs, which handles time at a different level of detail, are given are given and heterogeneous specification involving the usage of the involving the usage of the SR and untimed MoCs in the same specification is described. Section 4 conclusions of this work.

2. Mapping of SR and Untimed MoCs to SystemC

Synchronous MoCs include more restrictive assumptions on time information than untimed MoCs. In Jantsch (2004), two types of synchronous MoCs are considered: clocked synchronous and perfectly synchronous. The latter is also called SR (Benveniste and Berry, 1991). Specifically, a basic assumption of the SR MoC is known as the perfect synchrony hypothesis, which establishes that the system reacts, as a result of input data, producing output results in zero time (instantaneous reaction). That is, outputs are synchronous with the inputs. Perfect synchrony restrictions involve a more detailed level handling of time since it may be considered as a restriction added to the partial order between events (given also in untimed MoCs) provoked by the topology and causal relationships among the elements of the specification.

In an SR MoC, all the activity of the system concentrates on specific points of the time axis called slots, an event being synchronous with every event in the same slot. No further restrictions are imposed on event time tags, except for the set of slots to be an ordered set (such as natural numbers; Jantsch, 2004). Neither a physical interpretation nor a regular distance between event time tags is required. A further implication of perfect synchrony affects the role of constructive elements. A distinction must be made between generation entities plus their associated events (usually part of the environment) and reactive entities/events (usually part of the system). The former ones constitute a spontaneous functionality, which do not need to be triggered and which is able to feed the rest of the specification with new events. In many specifications, particularly, in those of reactive systems, spontaneous functionality is found only in the environment. The reactive entities form a reactive functionality, which needs the events to be present to run. Due to the perfect synchrony hypothesis, reactive functionality triggers and reactive functionality can be decomposed into decomposed into reactive computation quanta, that is, functionality portions without explicit delays or blocking computation quanta form a reactive chain, respecting the perfect link does not introduce a delay. introduce a delay.

The perfect synchrony hypothesis does not mean that the system processes of the specification need to be synchronous under the terminology of Lee and

Sangiovanni-Vincentelli (1998). Although some modeling approaches oblige every signal to always present events with the same tags (such is the case of Ptolemy, making explicit the bottom, $\perp$, to express the absence of value), others may not do so, making $\perp$ implicit (such is the case of our methodology, as will be seen).

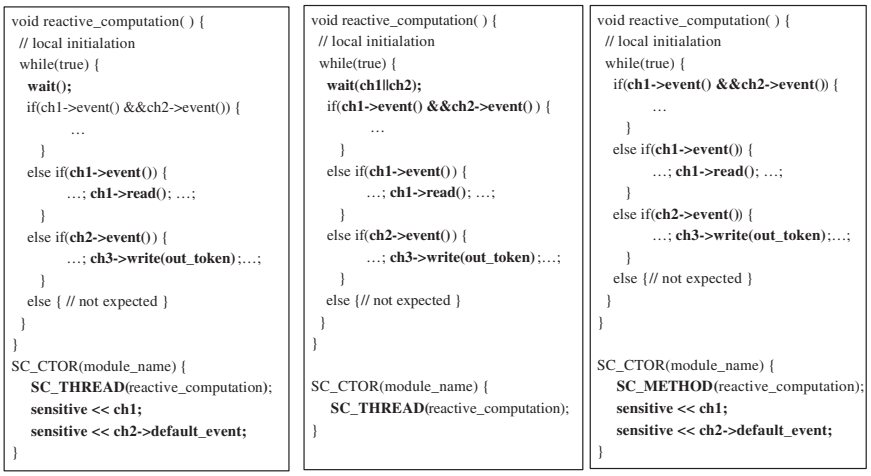
The fact of having the same time tags in input and output events, despite having a causality relationship among them, brings about some problems or paradoxes when cases of convergence and feedback are present in the specification. In order to explicitly solve this, new specification elements that explicitly introduce time information are often provided. Such is the case of delay elements to break feedback loops or empty events. This also affects the way in which the systems can be hierarchically composed.

Once the SR basic assumptions have been seen, the way in which this MoC is supported in the SystemC specification methodology is described. As mentioned, this is done in the form of new guidelines and elements, following the general specification methodology, which specification methodology, which in this case is extended in order additional checking features.

As the general specification methodology states, any functionality is enclosed in SystemC processes, while communication is done only through channels. The separation between spontaneous and reactive functionalities is supported through the separation between generator processes (GPs) and generator channels (GCs), usually in the context of the environment, and reactive processes (RPs) and reactive channels (RCs), usually part of the reactive system. A GP generates spontaneous events through write accesses to GCs. A GC triggers an RP that in time, can be linked to a sequence of more RPs, communicated by more RCs forming a reactive chain. The existence of concurrency in the GPs provokes the existence of several concurrent reactive chains. Another possibility for concurrency is a GP or an RP writing to two or more RCs connected to two or more RPs (i.e., two or more reactive chains).

Both GPs and RPs are enclosed in `SC_THREAD`s. RPs may also be written through `SC_METHOD`s. (data transfer and synchronization) among processes is supported is supported through the `uc_SR` channel (Herrera et al., 2004).

Several process specification styles are possible. A GP has a style similar to the one used in untimed MoCs, acting as a nonblocking process, running in a loop inside a `SC_THREAD` without a sensitivity list. A wait on time in the GP can separate the generation events to resume different slots. Nevertheless, a reactive process or RP RP (Figure 4.2) presents a statement, which stops the process activity until some event is some event is present. This event is a write access to at least one of the `uc_SR` (or `sc_buffer`) channels the process is reactive to. When the RP is enclosed by an `SC_THREAD`, this list can be statically (left-hand side of Figure 4.2) or dynamically (center of Figure 4.2) provided. Another static option is through an `SC_METHOD` (right-hand side of Figure 4.2),



```
void reactive_computation( ) {
 // local initialation
 while(true) {
  wait();
  if(ch1->event() &&ch2->event()) {
          …
    }
  else if(ch1->event()) {
          …; ch1->read(); …;
    }
  else if(ch2->event()) {
          …; ch3->write(out_token);…;
    }
  else { // not expected }
 }
}
SC_CTOR(module_name) {
   SC_THREAD(reactive_computation);
   sensitive << ch1;
   sensitive << ch2->default_event;
}
```

```
void reactive_computation( ) {
 // local initialation
 while(true) {
  wait(ch1||ch2);
  if(ch1->event() &&ch2->event()) {
          …
    }
  else if(ch1->event()) {
          …; ch1->read(); …;
    }
  else if(ch2->event()) {
          …; ch3->write(out_token);…;
    }
  else {// not expected }
 }
}

SC_CTOR(module_name) {
   SC_THREAD(reactive_computation);
}
```

```
void reactive_computation( ) {
 // local initialation
 while(true) {
  if(ch1->event() &&ch2->event()) {
          …
    }
  else if(ch1->event()) {
          …; ch1->read(); …;
    }
  else if(ch2->event()) {
          …; ch3->write(out_token);…;
    }
  else {// not expected }
 }
}

SC_CTOR(module_name) {
   SC_METHOD(reactive_computation);
   sensitive << ch1;
   sensitive << ch2->default_event;
}
```

Fig. 4.2 SR reactive process styles in SystemC

which prevents the use of any `wait` statement inside the computation. The RP is explicit and powerful, in the sense that computation to be performed is defined for each delta trigger. This is done by means of an if statement, which considers the possible combinations. The combinations considered by each if branch treat if branch treat or react to triggers with the same SystemC time coordinate (t_i, δ_k). There is no explicit use of the empty use of the empty event/ token, $\perp$, in the body of the process. there is an implicit use, since, among the trigger combinations of some may implicitly have the absence of a trigger on a specific `uc_SR` channel.

For process communication, the `uc_SR` channel provides additional features to those of `sc_buffer`, a standard SystemC channel, which could give a basic coverage of SR MoC in SystemC. The `uc_SR` channel provides a and a read interface with two access methods, written (or, equivalently, event) and read. It permits a unidirectional flow of triggering events and single data tokens of generic type from write to read interface. The write access method allows the GP to allows the GP to pass a generation event and, if desired, a updated at the end of each delta, in which a write access has been has been present. The read access is divided into a written access method, which permits checking if a specific `uc_SR` `uc_SR` instance was responsible for the current makes effective the read of the data without consumption. Therefore, without consumption. Therefore, any number of readers of the same `uc_SR` instance is allowed.

With respect to the execution semantics, the order of execution is fixed by the system topology and the semantics of the processes and the `uc_SR` channel. The topology is stated by the way in which processes are connected through channels. The semantics of processes is given by the standard and the structure

of GPs and RPs. `SC_THREAD`s trigger at the beginning of the execution and only stop execution if they find a blocking access wait or a delay statement. `SC_METHOD`s are reactive computation quanta, which admit no blocking statement. The `uc_SR` channel has its own abstract semantics transmitting an event and a data token in δ-time (without t-time advance) and offering nonblocking, nonconsuming read access.

In order to go into more detail about SR execution semantics in SystemC, the abstract concept of the time handled up to now for the general presentation of the SR temporal semantics is substituted. Till now, each event, e, was assumed to have an associated time tag represented as $T(e)$. From now, the simulation mechanism, which handles a time axis comprising a a double coordinate, is considered. For that double coordinate the notation $T(e) = (t_i, \delta_j) =$ (time-coordinate, delta-coordinate) is now assigned. The delta, δ_j, represents the SystemC evaluation-update cycle, while represents the SystemC time advance. The t_i coordinate is the the dominant one in order to establish an order relationship between events. That is, if $T(e_0) = (t_0, \delta_0)$ and $T(e_1) = (t_1, \delta_1)$ when $t_0 < t_1$, then $T(e_0) < T(e_1)$, the relationship between δ_0 and δ_1. In case $t_0 = t_1$, then $T(e_0) < T(e_1) \Leftrightarrow \delta_1 << \delta_2$. The flexible interpretation of deltas is necessary to support untimed MoCs and to give the designer enough flexibility to decide where to schedule each event depending on the mapping to the different processing elements (specific HW, application-specific processors, general-purpose processors, etc.).

Channels are the main elements to control the MoC temporal semantics. Their SystemC implementation is mainly done through the `sc_event` and `sc_primitive_channel` SystemC classes. The `sc_event` class permits time control at delta level through methods. The primitive channel primitive channel (`sc_primitive_channel` class) and its `update` methods enables a finer control. This is finer control. This is because the channel is able to perform actions at any of the two basic stages of the delta cycle (namely, the evaluation and the update phases).

The distinguishing time restriction in untimed models is the preservation of a partial order (P. O.) among events. Figure 4.3 represents in a simple CSP example how the restrictions, written in abstract terms within the box, are mapped over the SystemC time axis. The P. O. comes from the concurrency among processes and the synchronization restrictions. There is a total order (T.O.) within the same process computation. In SystemC, T. O. within the process computation is easily fulfilled since it comprises a sequential C/C++ algorithm. $T(e_{im}) < T(e_{in}) \cap T(e_{km}) < T(e_{kn})$ happens in Figure 4.3 only if there is at least a delay in delta between m-th and n-th events. Otherwise, $T(e_{im}) = T(e_{in}) \cap T(e_{km}) = T(e_{kn})$ fulfills. Synchronization restrictions come from time semantics of the channels $T(e_{im}) < T(e_{kn}) \cap T(e_{km}) < T(e_{in})$ for the rendezvous in commented, the channel semantics is achieved through a

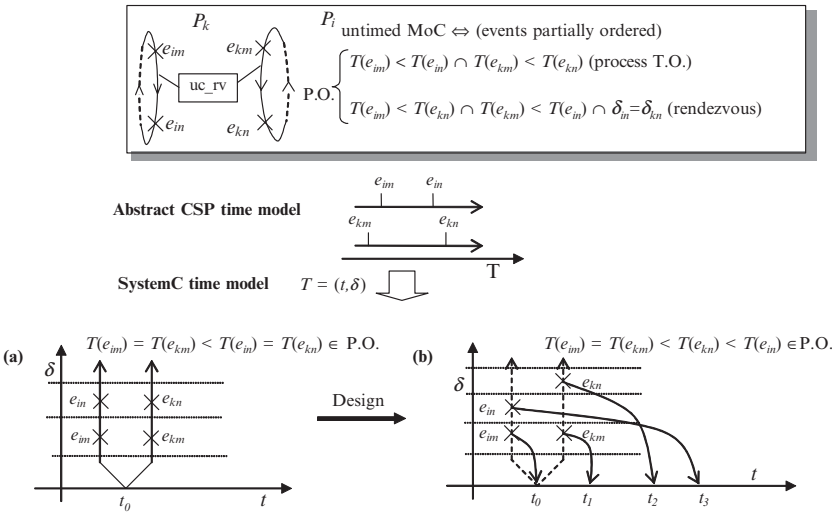


Fig. 4.3 Untimed MoCs P. O. over the SystemC time axis

SystemC implementation using the synchronization primitives using the synchronization primitives `sc_event` and `sc_primitive_channel`.

The flexible or relaxed interpretation of the untimed specification in the SystemC approach can also be appreciated. For example, a whole execution could run in the same "instant" ($t_{im} = t_{in} = t_{km} = t_{kn}$), over a δ-axis, varying only the δ_i coordinate, which is thus the only variable responsible for keeping the P.O. (Figure 4.3a). In the other extreme, each could have associated a different t_i (Figure 4.3b), for example, after an implementation or a time-estimation process. Anyhow, the P.O. has been preserved. An important remark is that, required conditions do not force a strict mapping of event mapping of event time tags over the SystemC time coordinate, but of the P.O. inherent to the specification plus the rest of rest of MoC time assumptions (perfect synchrony).

About the abstraction of the SR MoC from SystemC, as with MoCs, both spontaneous and reactive functionality are specified specified through C/C++ sequential algorithms within SystemC processes. Therefore, the T. O. within the process algorithm is kept. Perfect synchrony hypothesis is accomplished when the input and the provoked output events in the same time, interpreted in this methodology as having t_i t_i tag ($t_{i\ \text{inputs}} = t_{j\ \text{outputs}}$). Therefore, delta delays in computation do not affect the fulfillment of instantaneous reaction in the MoC. This means that while events of processes in the same reactive chain share the same SystemC t_i coordinate, a flexibility in the order of events with respect to the delta coordinate may occur. Specifically, this P. O. may happen among the events belonging to different processes of the same reactive chain. If

the reactive chain is composed by a single branch, there will be a T. O. among events of the same reactive chain, otherwise, there will be a P. O. A connection to an untimed MoC may provoke this P. O. on the reactive chain (i.e., an access to a first-in first-out (FIFO) in an RP), which does not involve a violation of the perfect synchrony. A more clear source of P. O. is the concurrency of reactive chains, which appears with several GCs. As can be seen, the common factor of P. O. is concurrency.

In Figure 4.4, it can be seen how e_{im} (m-th event of the i-th process) and e_{km} m-th e_{km} (m-th event of the k-th process) are related by the SR channel, which provokes a reaction in delta time but without t_i shift. That is, $\delta_{im} + 1 = \delta_{km}$ and $t_{im} = t_{km}$, which means $T(e_{im}) < T(e_{km})$. The RP reacts t_i advance and preserving its internal T. O. The sequential sequential algorithm of P_k computation determines that e_{km} occurs before e_{kn}. Formally, in the methodology, $T(e_{km}) \leq T(e_{kn})$ since a delta delay or even a time delay (d_2) may appear or not between these events. For instance, a delta delay may appear if the RP performs an access to a rendezvous channel. This is the case of MoC connections. In the case of Figure 4.4a, $T(e_{km}) = T(e_{kn})$, that is, happen in the same (δ, t) SystemC time, then fulfilling fulfilling $T(e_{im}) < T(e_{km}) = T(e_{kn})$. The GP and the GC may be formally defined. A GP is that making a write to a generator SR channel (GC). A GC is that receiving a write access first δ (δ_0) of the slot. The generator process GP required to separate write events in the t_i coordinate. Each t_i coordinate. Each single t_i coordinate represents a slot. No GP should generate more than one generation event to a single GC in the same t_i. Therefore, the generation events are separated by an arbitrary $d = (t_j - t_i) > 0$ time (with $j \neq i$). This separation is reflected in the d delay of Figure 4.4, which may be a data-arrival rate of the environment. However, since environment. However, since generation events are, in general, provided that $t_i \in \aleph$ in SystemC, then the t_i set or slot set in this SR approach fulfills being a totally ordered set. Therefore, the t_i coordinate defines the total ordering among slots over the

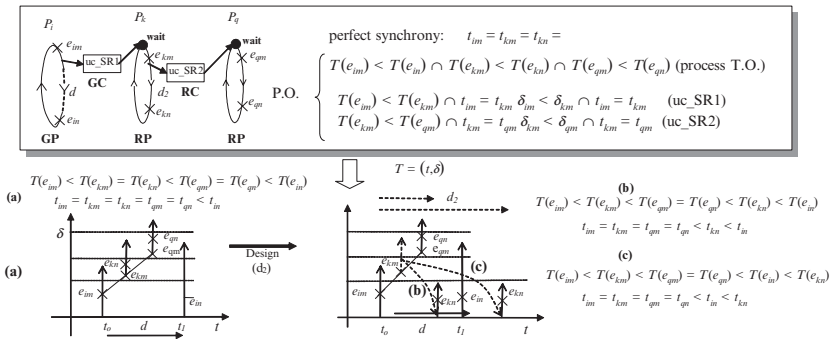


Fig. 4.4 SR MoC adds T. O. in the reactive chain and perfect synchrony

t-axis (if $t_i > t_j$, then $T(e_i) > T(e_j)$ the t-axis if $t_i > t_j$, then $T(e_i) > T(e_j)$ independently implementation enables the placement of the chained process in the the chained process in the same t-axis coordinate, with a the δ-axis, which strongly depends on the topology. The topology. The number of deltas between consecutive t tags, t_i not specified and depends on the activity of the system. In this depends on the activity of the system. In this sense, events of the reactive chain execute an SR evaluation and keeping t_i constant. Notice that a SystemC evaluation cycle is the first part of the evaluation-update cycle called δ SystemC simulation. This is different from the SR evaluation cycle, which involves one or several δ cycles. The longer δ cycles. The longer the reactive chain or the feedback loop, the larger the number of deltas, so the activity so the activity present between consecutive time slots. so the activity present between consecutive time slots.

Bearing in mind the assumptions shown, in an isolated SR MoC, there is always a static structure (the GPs of the reactive chains remain the same during all the execution). Despite these specification guidelines and new elements, the high flexibility of the language may still drive to violations of the SR MoC conditions. Most cases of possible violations of have to do with the break of the reactive chain through a through a delay (provoking a shift on t coordinate) in the reactive chain (Figures 4.4b and c). The the event order and determinism property, which do not necessarily have to be lost when perfect synchrony is missed, as a consequence consequence of a design process (d_2 delay in Figure 4.4).

As in untimed MoCs, some dynamic checks permit monitoring SR conditions. The ability to selectively apply them gives flexibility for constructing the reactive specification with different restriction degrees. A full accomplishment of SR conditions may be configured by default to obtain the benefits offered by this MoC. Summarizing, the checks are `SEVERAL_CALLERS`, `LOST_TRIGGER`, `OLD_VALUE_READ`, `SAME_DELTA_WRITE`, `SAME_TIME_WRITE`, `CHECK_DELTA_NUMBER`, and `CHECK_AUTOFEEDBACK`. Some of them are of special interest when the SR MoC is connected with untimed MoCs.

`SAME_DELTA_WRITE` (`BURST_WRITE` in Herrera et al. (2004)): It detects if the channel is written more than once in the same delta cycle. This eliminates the indeterminism source of loosing $N - 1$ data tokens from N write events, coming from either several or the same process.

`SAME_TIME_WRITE`: It checks one of the necessary conditions for avoiding nondeterminism when perfect synchrony is assumed. Specifically, it checks that a GC is not written more than once in the same time (t_i coordinate). In this way, it is guaranteed that a reactive chain is triggered by only one data token at each generator channel, written at the first delta of the reaction. This does not require a specific or regular

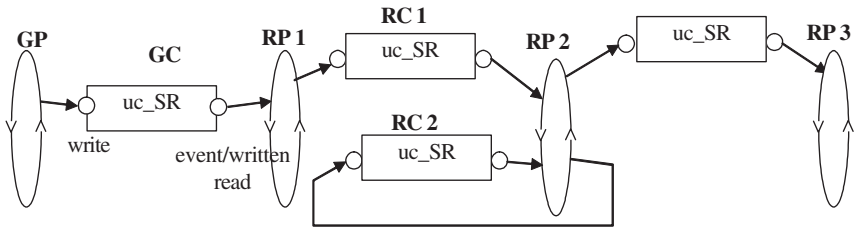


Fig. 4.5 SR reactive chain with a feedback loop

temporal spacing, but only its presence. This check is more restrictive than the `SAME_DELTA_WRITE` check, which allows a multiple write of the GP during the slot (since a slot may comprise more than one delta).

`SAME_TIME_WRITE`: This check is restricted to the GCs, instead of every `uc_SR` instances. The rest of SR channel instances, RCs, may be triggered more than once per t_i or slot (although no more than once per delta if the `SAME_DELTA_WRITE` is activated). This could happen in the reactive channel (RC 2) shown in Figure 4.5, since it is involved in a feedback loop inside the reactive chain. The loop in Figure 4.5 (which appears by a sequential composition in the reactive process RC 2; Jantsch, 2004) exposes the need for stabilization in order to reach a finite reaction in the slot and thus an advance at t_i coordinate. Finite reaction means that there will be no more triggering events (writes) after a finite number of deltas. In this approach, neither a formal nor a static check for stabilization is done. Instead, the `CHECK_DELTA_NUMBER` check has been provided.

`CHECK_DELTA_NUMBER`: It monitors the number of deltas given in an `uc_SR` instance at each slot. If it is defined as 0, it performs an activity analysis, reporting at the end of the simulation the number of deltas per `uc_SR` channel instance in each slot. In case it is defined as a number greater than 0, it becomes a limit, which raises an error if it is reached in some channel instance. Reaching this limit does not formally mean that the stabilization was not possible, but, at least, it gives a track for the sink-time of a never-ending simulation and the most active points of the system specification.

3. Untimed–SR MoC Interfaces

The study and development of the MoC interfaces is systematically done through an MoC interface taxonomy (Herrera et al., 2005) based on the MoC taxonomy in Jantsch (2004). In these taxonomies, a distinctive feature is the detail level handled in the time domain.

The MoC taxonomy establishes groups of MoCs, characterized by sharing basic assumptions in the time domain. Furthermore, the groups handling more detailed time information incorporate additional new restrictions. This enables the support of several MoCs by means of abstraction from more detailed MoCs and provides a basis to face the problem of MoC interfaces.

In general, whatever the MoC combination, the time axis that lets the whole set of events of the specification be coherently placed has to handle the most detailed level among those handled by the MoCs present in the specification. This is illustrated in Figure 4.6. An SR MoC and a CSP MoC are connected through an MoC interface, which in this case is a border process (BP). The most detailed level feasible in the methodology is the DE strict-timed MoC. However, MoC primitives MoC primitives are abstract (i.e., the `uc_SR` and the `uc_rv` channels) and no DE temporal semantics has to be taken into account. As has been seen, channel semantics and the T.O. of each process determines the execution semantics. In the case of Figure 4.6, the more restrictive semantics of `uc_SR` channel will force the hypothesis (implemented over the DE kernel) affecting the time the time tagging of the events of all the specification, including including those of the CSP part.

On the other hand, handling a different level of detail in the time domain in each MoC part involves that the MoC interface must perform an adaptation (something not necessary in untimed–untimed MoC interfaces; Herrera et al., 2005), extracting or time information, or at least the consideration of the possible possible incompatibilities. For instance, the BP must incorporate d_1 delay in Figure 4.6. It must also be considered that the violations of restrictions may cross MoC borders or be originated in other parts of the specification. Next, the untimed (PN/KPN/CSP)–SR MoC interfaces (highlighted in the right Figure 4.7), in the category of category of untimed–synchronous MoC interfaces (MI), will be presented. In this specification methodology, the SR MoC differs from the untimed MoCs in the communication domain (different channels)

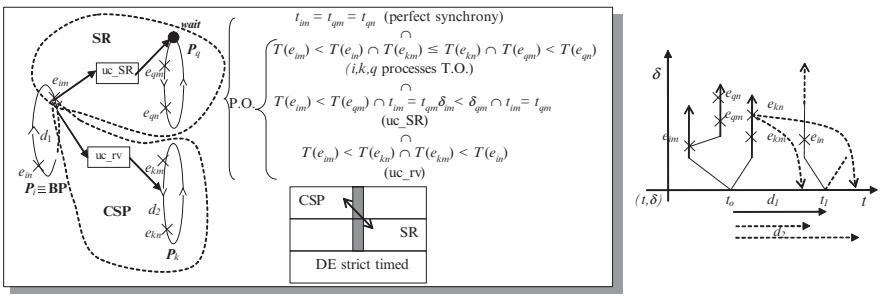


Fig. 4.6 Perfect synchrony and P. O fulfilling in CSP–SR connection

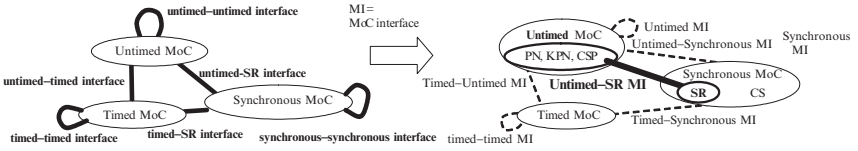


Fig. 4.7 Untimed–SR MoC interface in the MoC interface taxonomy

and, especially, in the time and computation domains. Thus, the untimed–SR MoC interfaces have to undergo adaptations in all these domains.

The adoption of the general specification methodology enables the consideration of a minimum difference in computation and communication domains. It is possible to use `SC_THREAD`s for specifying both untimed and SR MoCs. Moreover, both untimed and SR channels are similar in that they are basically templates allowing the unidirectional transfer of different data types. This permits making a general consideration of untimed–SR interfaces by analyzing only the differences in time assumptions and the distinguishing aspects in the communication domain, that is, in channels. Specifically, these aspects are the blocking or unblocking character of channel accesses, independent of which data is transferred and the sense of data transfer.

Two cases A and B are distinguished. In Figure 4.8, MoC interfaces are abstracted as lines delimiting MoCs. In case A, the SR MoC connects to the untimed MoC through an write access type, that is, entailing the untimed MoC as part of untimed MoC as part of the reactive chain. In Figure 4.8, $d = (t, \delta)$ represents the response delay accumulated in the untimed part, where t is the integer number of part, where t is the integer number of SystemC time units and δ is the integer number of deltas accumulated. Then, given that MoC assumptions are fulfilled in the SR part (slot total to different slots and instantaneous reaction of the reactive chain) and in the untimed part (partial ordering), three untimed part (partial ordering), three possibilities can be distinguished:

A.1 The untimed part provokes no blocking (immediate reaction, that is $d = (0, 0)$) or, in case it did, the reaction takes a limited number of deltas, $d = (0, \delta)$ with $\delta < \infty$. Then, perfect synchrony is kept in the SR part, since no advance in the t_i SystemC coordinate is given. The P. O. is also maintained in the untimed part, although events are overrestricted by the total slot ordering imposed by the SR part.

A.2 The untimed part is accessed through a blocking access, which provokes a reaction after $d = (t, \delta)$ time, with $0 < t < \infty$ (t is a bounded SystemC time) and $\delta < \infty$ (bounded number of deltas). While the untimed part keeps the P. O., the SR part sees a bounded reaction, but that introduces a shifting on t axis in the tail of the reactive chain. That is, the reactive

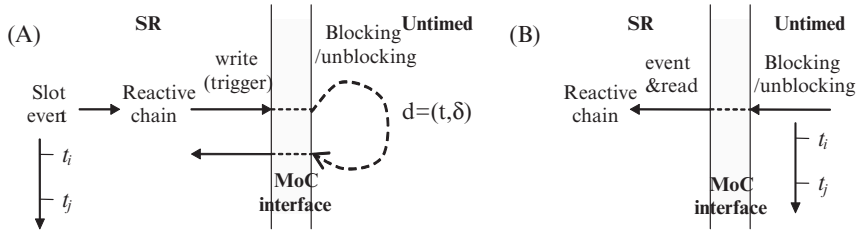


Fig. 4.8 Timing and blocking semantics in Untimed–SR interfaces

chain is broken. Although the break of the reactive chain violates the SR assumptions in SystemC, this kind of "timed" SR can still be useful for time analysis of possible implementations. Here two derived cases can be distinguished. The first one is when $t_i + d < t_j$ (case A.2.1). In that case, it can be considered that the reactive chain, prolonged by the untimed part, reacts fast enough and the system copes with the generation events. If $t_i + d \geq t_j$ (case A.2.2) reaction finishes too late, that is, after at least a new trigger has already arrived.

These cases are detected by a new check, represented in Figure 4.9, called `REAC_TIME`. This check can be configured at two assert levels. In a `RESTRICTIVE` mode, the violation of the perfect synchrony is asserted as an error, the simulation being stopped. In a `WEAK` mode, the reactive chain break will be allowed and reported as a warning if the reaction time was fast enough (case A.2.1). Otherwise (case A.2.2) an error is raised, because there would not be implementation to attend the triggers in time (before the new ones arrive), provided the explicit time information of the specification. In other words, any implementation would be unfeasible. The tricky question about the implementation of the `REAC_TIME` check is that the t_2 time must be tested in the reactive process, just before invoking the `wait`. This is implicitly done by an internal function called `check_reaction_time`. The association between GCs and reactive BP is implicitly done by means of the sensitivity list (or channels passed to the `wait`). If the style of the RP use a `wait` statement, a `wait_SR` macro lets implicitly call the `check_reaction_time` function.

A.3 The untimed part provokes a reaction that completes after $d = (t, \delta)$, with $t < \infty$ and $\delta = \infty$. Although in a limited t, it executes forever in the δ axis, which is equivalent to not to reach a stabilization within the slot. This situation is detected by the `CHECK_DELTA_NUMBER` check.

B A write access performed from the untimed part is considered. In this case, the untimed part behaves as a generator process of the reactive

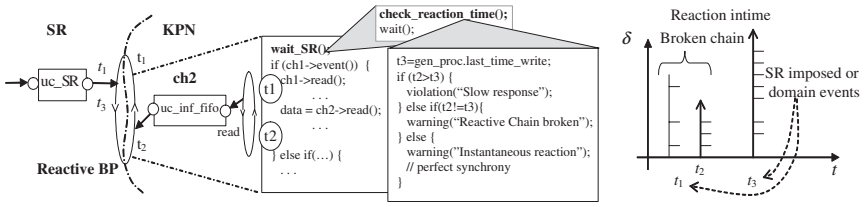


Fig. 4.9 Dynamic check of reaction time

chain in SR. Assuming that the SR part fulfills its assumptions, the untimed part of the MoC interface must present a separation at t_i coordinate in every pair of its accesses to the SR part. The `SAME_TIME_WRITE` check would detect a possible violation of this point.

After the generic analysis, the SystemC elements in charge of the support of heterogeneity can be shown. As commented, these MoC interfaces are BPs and BCs. The KPN–SR case is presented first. Later, this is extended to the PN–SR and CSP–SR cases. In Figure 4.10, four possible cases for BPs for KPN–SR MoC interfaces are shown. The cases a and b summarize the summarize the situation in which the BP performs as a reactive SR process, triggered by one or more SR channels. The BP accesses the `uc_SR` channel by means of the access methods corresponding to the read interface: the written (or event) method and the read method. Therefore, the reactive BP is involved in the middle of a reactive chain. Separately analyzing both cases, we first see in case a that the reactive BP also does a write to an infinite FIFO channel. Since this presents no blocking, the reaction of the BP lasts a finite number of deltas. That is, the BP without time delays plus an reactive chain in deltas without t_i advance, and the perfect synchrony hypothesis holds (case A.1 of untimed–SR interfaces). In case b, the reactive BP does a blocking read access. The question is whether this means a break of the reactive channel. If this means a break of the reactive channel. If the unblock inside the same time slot, case A.1 is repeated. However, in general, a blocking read access to infinite FIFO may drive to the possibility of A.2 or A.3 cases. Cases c and d of Figure 4.10 consider a BP where the access to the SR process is done by means of a write access which will provoke the trigger of a KPN-type access. The case B of untimed–SR interfaces is There is no incoherence with the KPN part and the point is to separate the writings to the SR reactive channel in different slots. In case c, at least an explicit delay is needed in the process. In the case d, that explicit delay in the BP is not case d, that explicit delay in the BP is not necessary whenever accesses from `uc_inf_fifo` occur separated in time.

Two types of border channels for KPN–SR are provided. They are represented in Figure 4.11 together with the time adaptation they perform.

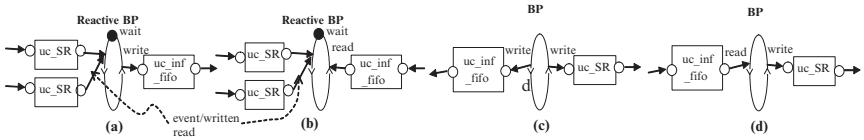


Fig. 4.10 Types of border processes in KPN–SR MoC interface

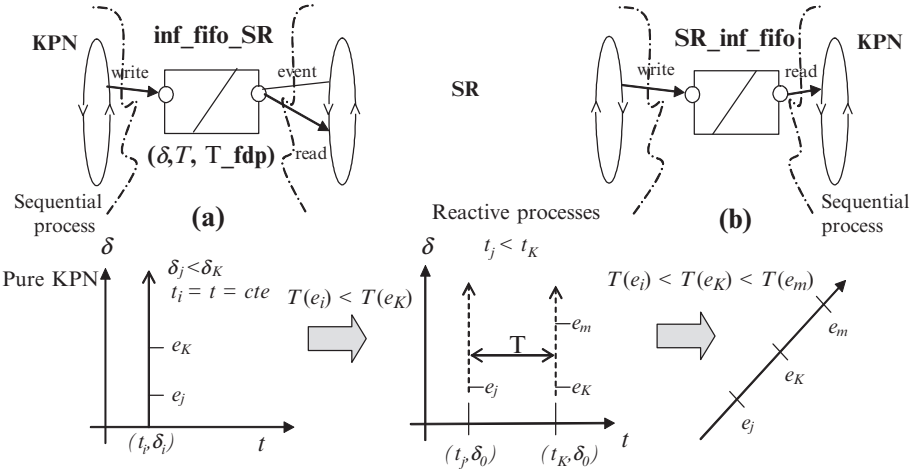


Fig. 4.11 Border channels in KPN–SR interface

Adaptation in the data domain is unnecessary since the channels are templates transferring generic data types. With respect to the computation domain, i.e., the style of processes they communicate and the semantics they have, in the case a, `inf_fifo_SR`, the KPN process is a typical sequential process, which can perform infinite writings into the channel. Not only are the data written accumulated but also their related events (write event queue). On the SR side, we have a reactive process whose sensitivity list can incorporate the `inf_fifo_SR` instance. The channel can receive a parameter at construction time, which would oblige the separation of successive writings in delta time (by default), in (represented in Figure 4.11), or in a random random amount of time. Another useful feature of this channel is that it can be configured to oblige time spacing as a sampling process. As can be seen, this is a BC with its own semantic and implementation. The case b, the `SR_inf_fifo` channel, has a write SR interface and a FIFO read interface. This channel is an infinite FIFO channel (`uc_fifo_inf`) extended with an SR write interface, thanks to its compatible nonblocking semantics with the infinite FIFO write interface.

Once the SR–KPN BPs have been shown, let us generalize to the case. The KPN MoC can be seen as a singular case ($N = \infty$) of of PN, which handles FIFO channels of limited size (N). For BPs, Figure 4.10 can be reused by substituting infinite FIFOs by limited FIFOs. With respect to the SR–KPN case, only the differences in cases a and c may appear. Case a generalizes now to the blocking possibility (A.2), thus the application of `REAC_TIME` check for this situation becomes necessary. The break of the reactive chain converts the reactive BP in a new generator process, since, although it only transfers events, it translates the remainder of the chain to a later slot (see right part of Figure 4.9). This slot is new with respect to the domain slots, that is, the set of slots forced by the SR part. This is a dynamism in generation, which contrasts with the static role of generator processes and reactive processes in a stand-alone SR specification. Case c does not present differences with respect to SR–KPN c case, since the finite FIFO does not ensure the write accesses to the `uc_SR` channels to be separated in the t_i axis. About BCs for the SR–PN, Figure 4.11 can be reused, but introducing `fifo_SR` and `SR_fifo` channels instead. The first case is similar to the `inf_fifo_SR` channel, but limiting the queue size and adding read blocking semantics. The second case is an extension of the limited FIFO channel supporting a write SR interface. The BPs and BCs for the CSP–SR case are considered now. A.1, A.2, A.3, and B cases can be identified. Recalling Figure 4.10 again (but substituting FIFO accesses by rendezvous channels), for cases a and b, the BP acts as a reactive process. Any rendezvous access will provoke a potential blocking access and `REAC_TIME` check is again applicable. Cases c and d are similar to those in KPN–SR and PN–SR. Whenever a rendezvous involves a t_i separation, the SR channel writings will not provoke the assertion of `SAME_TIME_WRITE`. Again, the BC cases are quite similar. An `rv_SR` channel accepts at its input the rendezvous write interface, but now there is no queue associated. Thus, it is basically an `uc_SR` channel with a rendezvous write interface, which admits the time spacing between write channel accesses. An `uc_SR_rv` is an asymmetric rendezvous channel (writing from SR side to CSP side), which provides an `uc_SR` write interface.

4. Conclusions

In this chapter, the untimed–SR MoC specification has been incorporated to a heterogeneous specification methodology based on SystemC. In this methodology methodology these MoCs can be easily mixed, thanks to the coherence in the mapping of the specification events over the SystemC (t, δ) time axis. Each MoC imposes its own mapping restrictions. Untimed MoCs impose a P. O., while SR MoCs add the perfect synchrony assumption. At the specification level, it is solved through interface BPs and BCs, which hide event handling and extend SystemC capabilities in a core-decoupled way. They also dynamically check the fulfillment of MoC conditions. Interface BPs and BCs also let

either a specification with clear delimited MoCs or an amorphous specification. These concepts have been exemplified with the connection between some specific untimed MoCs already supported in the methodology (PN, KPN, CSP) and the SR MoC itself.

The timing interpretation made in the chapter is one of many possibilities. A wide agreement should be reached in order to ensure a general methodology able to be standardized. In any case, a founded methodology fully understood by the user is required.

In future work, the incorporation of other untimed MoCs, such as SDF (Lee and Messerschmitt, 1987), will show the general character of the approach. Other works may explore the connection with MoCs with more level in the time domain, such as clocked synchronous, discrete discrete event and analogous models.

References

Benveniste, A. and Berry, G. (1991) The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9).

Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A., and Todd, L. (1999) *Surviving the SoC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, Norwell, MA.

Grötker, T., Liao, S., Martin, G., and Swan, S. (2002) *System Design with SystemC*. Kluwer Academic Publishers, Baston, MA.

Herrera, F., Sánchez, P., and Villar, E. (2004) Modeling of CSP, KPN and SR systems with SystemC. In: Grimm, C. (ed) *Languages for System Specification: Selected Contributions on UML, SystemC, SystemVerilog, Mixed-Signal Systems, and Property Specifications from FDL'03*, CHDL. Kluwer Academic Publishers, Boston, MA.

Herrera, F., Sánchez, P., and Villar, E. (2005) Heterogeneous system level specification in SystemC. In: Boulet, P. (ed) *Advances in Design and Specification Languages for SoCs: Selected Contributions from FDL'04*. Springer, Dordrecht, The Netherlands.

Hoare, C. A. R. (1978) Communicating sequential processes. *Communications of the ACM*, 21(8).

Jantsch, A. (2004) *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, San Francisco, CA.

Kahn, G. (1974) The semantics of a simple language for parallel programming. In: *Proceedings of the IFIP Congress 1974*, North-Holland.

Lee, E. and Sangiovanni-Vincentelli, A. (1998) A framework for comparing models of computation. *IEEE Transactions on CAD of ICs and Systems*, 17(12).

Lee, E. A. and Messerschmitt, D. G. (1987) Synchronous data flow. *Proceedings of the IEEE*, 75(9).

Michalove, A. (2006) Amdahl's law. `http://home.wlu.edu/ wahtleyt/classes/parallel/topics/amdahl.html`.

Müller, W., Rosenstiel, W., and Ruf, J. (2003) *SystemC: Methodologies and Applications*. Kluwer Academic Publishers, Boston, MA.

OSCI (2001) *Functional Specification for SystemC 2.0*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

OSCI (2006) *SystemC Library Overview*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/web/sitedocs/library_overview.html`.

Patel, H. D. and Shukla, S. K. (2004) *SystemC Kernel Extensions for Heterogeneous System Modelling: A Framework for Multi-MoC Modelling and Simulation*. Kluwer Academic Publishers, Boston, MA.

Posadas, H., Herrera, F., Fernández, V., Sánchez, P., and Villar, E. (2005) Single source design environment for embedded systems based on SystemC. *Journal on Design Automation for Embedded Systems*, 9(4): 293–312.

Ptolemy (2006) Ptolemy II: heterogenous modeling and design. Ptolemy Project. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`.

Villar, E. (2002) A framework for specification and verification of timing constraints. In: Mignotte, A., Villar, E., and Horobin, L., (eds) *System on Chip Design Languages: Best of FDL'01 and HDLCon'01*. Kluwer Academic Publishers, Boston, MA, pp. 267–274.

Chapter 5

Interface-Centric Abstraction Level for Rapid Hardware/Software Integration

André C. Nácul[1], Marcello Lajolo[2], and Tony Givargis[1]

[1] *Center for Embedded Computer Systems*
Department of Computer Science
University of California, Irvine
Irvine, CA 92697
USA

[2] *NEC Laboratories of America*
Princeton, NJ 08540
USA

Abstract With the continuous advances of high-level synthesis and hardware/software co-design, engineers now have the luxury and desire to explore multiple high-level architectures very quickly. System-level tools can enable trade-offs of architectures that rely on different combinations of memory access, resource sharing, and multiplexing. A good system-level design flow must enable fast and accurate viewing of multiple solutions based on different design choices. In this chapter, we present a system-level application programming interface (API) for text-based specifications that combine transaction-level modeling for the hardware interface, operating system (OS), and device drivers levels for the software interface into a unified semantics. We also present a refinement process that allows to generate a hardware/software integration very rapidly.

Keywords: interface synthesis; hardware/software codesign.

1. Introduction

The design of embedded systems is growing in complexity at a fast rate. Devices are more feature-rich than ever, incorporating new functionalities, newer protocols, and more modes of operation. At the same time, designers must keep pace to deliver a new generation of products in an even tighter

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 83–99.

time-to-market. Coupled with the growth in chip capacity, the task is daunting and calls for a simpler design flow.

Automation is one solution to cope with the growth in system design complexity. Automating the generation of code has long been used in different stages of embedded systems design, for instance, in hardware synthesis. More recently, software and operating systems (OS) have also been the focus of automation efforts (Besana and Borgatti, 2003; Herrera et al., 2003; Nacul and Givargis, 2004).

Aside from automation, the design flow also needs to maximize design reuse and portability. Ideally, functional blocks should be easily migrated from hardware to software, and vice versa. A higher level of abstraction in describing the system's functionality facilitates this process. In such high abstraction, system specification can be carried in a portable fashion, independent of associated models of computation, hardware availability, communication architecture, or specific OS support.

The system-level design community has tried to address these issues in different ways. We believe that a common programming interface is needed to allow designers to easily specify communication and iteration with an OS layer. SystemC (OSCI, 2006) has tried to support a common interface. Nevertheless, in the current version, the support for software and OS is still not complete.

In this work, we propose a generic framework for system specification. Our framework is composed of a portable application programming interface (API), its corresponding semantics, and alternatives for hardware and software implementation for each entry of the API. The objective is to provide designers with a minimal set of high-level primitives that can be used to abstract and specify the system behavior. Our API is not dependent on any system-level design language. Rather, we are presenting a methodology to synthesize hardware, software, and interface communication based on the proposed API. The API can be adapted or incorporated in the design language of choice. SystemC (OSCI, 2006), for instance, includes support for some of the primitives we present. Other primitives, however, are not present in SystemC, or have a behavior that is not the one we envision.

The API is partially based on the portable operating system interface (POSIX)/Pthreads standard (The Open Group, 2004), and encompasses primitives for processes instantiation, communication with shared variables and message passing, process synchronization, and timing behavior specification. The framework is integrated in our hardware/software codesign environment. When a functional block is mapped to a specific platform component, either hardware or software, we are able to automatically generate all the hardware descriptions, interconnections, software data structures, and even device drivers that will effectively implement the semantics of the API entries. Therefore, with the use of an abstract, high-level system description, it is

possible to automate hardware, software, and communication interface generation, enhancing portability and reuse.

A description provided with this API can easily be mapped to different processors, various OS, and many communication architectures, without the need to modify any part of the original specification. The use of a portable specification also enables a rapid exploration of design alternatives. Since the proposed API is platform- and implementation-independent, one can easily test different communication architectures or different OS much quicker than when using platform-specific code like the advanced microcontroller bus architecture (AMBA) bus API (ARM, 2003) or VxWorks OS API (Wind River, 2006).

This chapter is organized as follows. Section 2 discusses the previous works related to our proposal. We define the terminology used in this chapter in Section 3. Section 4 describes our API, presenting possible mapping alternatives. In Section 5, we present an example of hardware/software integration process starting from the proposed API. We present our conclusions in Section 6.

2. Related Work

System-level design has been the motivation for many publications in the literature. Most of the approaches address one of the components of synthesis, be it communication, hardware, or software synthesis. The synthesis of (real-time) OS support has been studied more recently. However, none of the approaches integrate all the parts into one framework, as we propose in this work.

Cadence's Virtual Component Codesign Environment (VCC) was an earlier tool in trying to provide a design space exploration environment for systems-on-chip (SoCs). It used library components to synthesize the design. VCC lacked a complete path to implementation, though. In this sense, Dziri et al. have combined VCC to other tools in order to provide a complete path to implementation (Dziri et al., 2003). The main difficulty was integrating the different design tools, each of them using a different specification model.

Concerning interface synthesis and analysis, Meyerowitz has presented a tool (Meyerowitz et al., 2003) that can evaluate different bus architectures and arbitration protocols. He shows that response time and bandwidth utilization can improve by combining different arbitration protocols. Passerone et al. generate interface adapters, allowing IP blocks to communicate even with incompatible protocols (Passerone et al., 1998). A transaction-level model for the AMBA bus in SystemC 2.0 is presented in Caldari et al. (2003). They propose a set of high-abstraction classes to model communication interfaces. This is similar to the work presented by Coppola et al. also on the synthesis of communication interfaces (Coppola et al., 2003).

There have also been proposals addressing automatic generation of real-time operating systems (RTOSs) for SoC architectures. However, most of them are concerned only with the software aspects of the RTOS generation, and do not integrate hardware synthesis or custom communication infrastructures. Some examples of these works are the proposals found in Le Moigne et al. (2004), Besana and Borgatti (2003), and Herrera et al. (2003). Gerstlauer et al. propose to model the RTOS functionality with system-level language primitives, refining the RTOS with the specification (Gerstlauer et al., 2003). His work is integrated in the SpecC environment.

In terms of system-level design languages, the two most developed approaches are clearly SystemC (OSCI, 2006) and SpecC (Gerstlauer et al., 2001). The transaction-level modeling (TLM) provided by SystemC supports modeling of hardware and software components, tied together with different communication interfaces. Software support and specially RTOS support is not yet fully integrated in the language, though, and is the subject of discussions for the next SystemC release. The same applies to SpecC, which still does not have a fully integrated RTOS interface.

3. Terminology

The terminology regarding OS, hardware and software design is way overloaded. Different terms are used to refer to the same concept, and the hardware and software communities do not have a common terminology. Since we are trying to define a system-level API suitable for hardware and software implementations, it is mandatory that we define the terms that will be used throughout this work. Two terms are specially of interest to us, be it processes, tasks, and threads; and concurrency and parallelism.

In the software and OS community, threads refer to units of execution, whereas processes also include resource allocation. A thread is generally viewed as a light process, because there is no need to allocate a separate memory space, file tables, page tables, and other structures that are common to processes. Usually, a process may contain many threads. In the RTOS community, the term task is used to represent an execution job, usually implemented by a thread. On the other hand, there is no thread concept in the hardware community. Instead, the term process is more common and often refers to some sort of processing unit, like a processor or an application-specific integrated circuit (ASIC). When describing two processes in hardware, we typically end up with two data paths and two controllers. In this chapter, we will use the term *process* to designate unit of execution, be it a hardware or software implementation.

Similarly, the terms parallel and concurrent are used interchangeably. In this work, *parallel* processes are those truly executing in parallel, i.e., at the same time, without multiplexing. Therefore, we need distinct hardware in

order to be able to run processes in parallel. On the contrary, *concurrent* refers to processes that are competing for the same hardware, usually in a time-multiplexed fashion. To the user, they are seen as if they were executing in parallel, but in reality only one of them will be executing in one processor at any given clock cycle.

4. System-Level API

The proposed generic API for design specification is presented in Table 5.1. It is partially based on the POSIX standard (The Open Group, 2004), a well-defined and accepted programming interface for OS, and includes extra primitives that are not part of POSIX. The API is divided into four parts: process management, communication, synchronization, and timing. Process management includes functions to control process creation and execution. The communication part encompasses shared memory and message-passing-based communication, both blocking and nonblocking style. Synchronization includes primitives for process synchronization, like mutexes, semaphores, and condition variables. Finally, the timing section allows some control over the

Table 5.1 The API functions

API parts	*API functions*

Process management	`process_create(id, param, func, arg)`
	`process_delete(id)`
	`process_suspend(id)`
	`process_resume(id)`

Communication	`port_send(port, data, size, mode)`
	`port_receive(port, size, mode)`
	`shared_mem_read(mem, offset, size, mode)`
	`shared_mem_write(mem, offset, data, size, mode)`

Synchronization	`mutex_lock(mutex)`
	`mutex_unlock(mutex)`
	`sema_wait(sem)`
	`sema_post(sem)`
	`cond_var_wait(var, mutex)`
	`cond_var_signal(var)`
	`cond_var_broadcast(var)`
	`sched_yield()`

Timing	`time_wait(time)`
	`process_join(id)`
	`mutex_lock_tmo(mutex, time)`
	`sema_wait_tmo(sem, time)`
	`cond_var_wait_tmo(var, mutex, time)`

timing behavior of the system, providing a timed-wait and controlling timeouts for blocking operations.

The API is thought to be integrable with any system-level specification language like SystemC. The API represents the abstract functionality that is believed to be needed to facilitate the design of hardware devices and the specification and synthesis of OS-based software. Some design languages might already include an equivalent form of part of the API. SystemC, for example, has its own classes for mutexes (`sc_mutex`) and semaphores (`sc_semaphore`), which work very similarly to those presented here. In that case, the native classes can be used. Other entries of the API are not available in SystemC or any other design language, and therefore must be included.

The API functions are inspired by POSIX and TLM. Process management and synchronization primitives are largely based on POSIX. There is a clear one-to-one mapping of the API entries to POSIX primitives. These are more likely to be used in software descriptions. Meanwhile, communication primitives are the highly abstract send and receive typical of a TLM description, along with shared memory access, useful for both hardware and software designs. The range of specification styles possible to target with the API is very broad. Hardware-oriented specifications might use bit manipulation and low-level constructs more intensively, whereas software-oriented specifications could use pointers, memory allocation, and stack manipulation more frequently. Nevertheless, the API we propose is neutral and can accommodate either style.

In the process management section of the API, four functions are defined. The function `process_create` is used to instantiate and start the execution of a new process. The function `func` is the entry point of the process. Note that the actual code of the process, be it hardware or software, is already available. The API function will create a new context for the new process and start executing the initial function. Also note that in case of hardware processes, if more than one process share the same hardware implementation, there is a need to synthesize a scheduler within the hardware implementation, so that time sharing of the hardware is possible. The `process_delete` stops and removes a process from the scheduler list forever, freeing all the resources that were held by that process. Finally, `process_suspend` and `process_resume` are used to stop and resume the execution of a process, respectively. A process is suspended by a `process_suspend` call, and stays suspended until some other process executes `process_resume` for that specific process.

Two different communication models are supported in the API, message passing and shared memory. Message passing is abstracted out by the concept of ports, and provides the primitives `port_send` and `port_receive` to implement the communication. Blocking and nonblocking styles are supported and are specified by the designer through the argument `mode`. A blocking send blocks the sender until the receiver reads the message. Similarly, a blocking

receive blocks the receiver until a message is available in the corresponding port. Shared memory communication is modeled with the `shared_mem_read` and `shared_mem_write` primitives. Here, two styles are also possible, synchronous and asynchronous, specified in the `mode` parameter. In synchronous mode, a lock is associated with each shared memory block, and only one process can access the memory at one specific time. Meanwhile, the asynchronous mode does not have a lock associated with the memory, and therefore concurrent accesses can happen. It is up to the programmer to ensure the correct behavior of the accesses. In all communication primitives, the size of the data block to be transmitted or received is specified in the `size` parameter. In case the data size is larger than the specified width of the communication interface, a protocol will have to be implemented to ensure that the data is correctly partitioned in the sender, and received and reassembled in the receiver.

In the synchronization section, three different synchronization mechanisms are defined by the API: mutexes, semaphores, and condition variables. A semaphore is a synchronization mechanism that controls access to shared devices or data structures. A semaphore is initialized to a specific count value C, representing the number of available devices or the number of concurrent accesses possible. A call to `sema_wait` will block the calling process if the semaphore value is zero, meaning that none of the shared resources are available, whereas a call to `sema_post` increments the value of the semaphore and unblocks a possibly waiting process. Mutexes are similar to binary semaphores, i.e., semaphores initialized with the value of one. The process calling `mutex_lock` will block in case the mutex value is zero and `mutex_unlock` will set the mutex value to one, allowing one of the possibly waiting processes to continue. Furthermore, condition variables allow processes to wait for some event or condition to happen. The process calling `cond_var_wait` will block until the condition is met and the corresponding `cond_var_signal` is invoked. Alternatively, `cond_var_broadcast` can be used to signal an event when multiple processes should resume execution as a result of one event. Finally, the last entry in the synchronization section is `sched_yield`, which is an explicit release of the processing unit. In software implementations, it will result in a context switch, whereas in a hardware implementation, it will be equivalent to forcing a clock boundary in the execution.

Lastly, the timing section allows the specification of the timing behavior of processes. Processes can wait for a fixed amount of time using the API called `time_wait`. The waiting time is provided in the parameter `time`. Additionally, it is also possible to specify time-outs for each of the blocking synchronization primitives, with `sema_wait_tmo`, `mutex_lock_tmo`, and `cond_var_wait_tmo`. These functions behave exactly like the corresponding non-timed-out versions, except that a maximum blocking time is provided as an additional

parameter of the function and an exception will occur in case the time-out is reached. In this case, the designer should handle the error appropriately.

4.1 Interface Synthesis

When the input design description contains communication primitives from the system-level API, there is a need to synthesize the communication interface between the processes. Depending on the design partitioning, the interface will need to connect two hardware modules, two software modules, or a hardware and a software module. In this section, we show examples of custom interface synthesis for different partitions. We refer to the process-sending data as the producer, and the process receiving data as the consumer.

Hardware-to-Hardware Communication. In the case where two processes that communicate through ports are mapped to a hardware implementation, there are different alternatives for interface synthesis. However, since this is a hardware-to-hardware communication, it is not necessary to generate RTOS code or software to handle this specific communication.

One possible architecture for a port-based hardware-to-hardware communication is shown in Figure 5.1. In this case, there is a direct data connection between producer and consumer. Additionally, control lines are synthesized according to the API usage. If the port is ever used for a blocking send, then an acknowledge line from the consumer to the producer is necessary. Therefore, the producer is suspended until it receives an acknowledge from the consumer in case of a blocking communication. For communications with multiple consumers, the producer waits for the acknowledge of all consumers. This behavior is implemented with a logic OR of the individual acknowledges of the consumers, as shown in Figure 5.1. Similarly, an event line is added from the producer to each consumer for the case when blocking receives are specified. Since the event and acknowledge control signals are synthesized only when needed, they are shown with dashed lines in Figure 5.1.

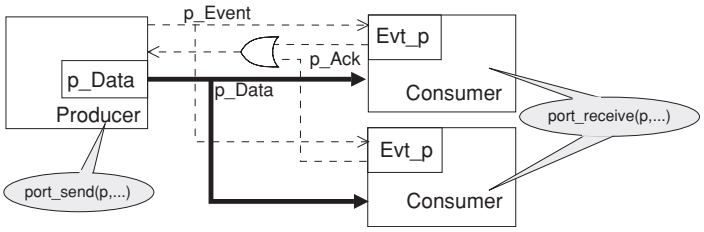


Fig. 5.1 Interface synthesis for hardware-to-hardware communication

Other architectures are also possible from the same system-level API. For instance, it is possible to generate a transaction-level model with AMBA-bus transactions for each port primitive. In this case, the `port_send` and `port_receive` primitives are replaced by a set of calls to the AMBA transaction-level API (ARM, 2003).

Software-to-Software Communication. When two software processes are mapped to the same processor, the interface synthesis is simpler. Our framework will generate a software data structure in memory, shared between the processes, that will keep the data along with event and acknowledge control signals. All the producer has to do is to update two memory locations, with data and event signaling (in case of blocking receives), whereas the consumer will read the data memory and update the acknowledge bit of the same port. Figure 5.2 shows the interaction between the processes.

Hardware-to-Software Communication. Hardware-to-software communications can be implemented by either interrupts or polling, using memory-mapped addresses in the latter case. In both cases, we will need some RTOS support in order to coordinate the processes. One possible solution is shown in Figure 5.3. Our framework will generate a bus adaptation layer for the hardware module, so that it can send and receive data from the bus. In the case of a memory-mapped communication, a device driver is also generated and runs inside the processor, monitoring the bus for activity in the memory-mapped region. The device driver is responsible for transferring data from the bus to the processor memory, to an equivalent port structure as the one shown in Figure 5.2. The software process will access the port data structure as it did in the software–software case, retrieving data and updating event flags. If instead an interrupt-based communication is specified, then an interrupt service routine (ISR) needs to be synthesized. The ISR will be responsible for receiving the event signaling from the producer. In the interrupt-based communication, the actual data is still transferred through a memory-mapped location to the port structure.

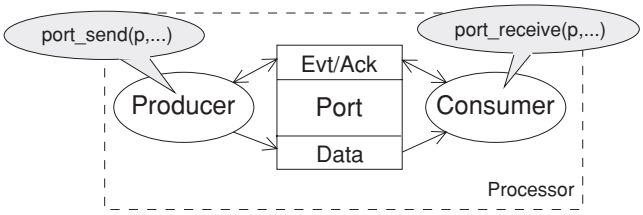


Fig. 5.2 Interface synthesis for software-to-software communication

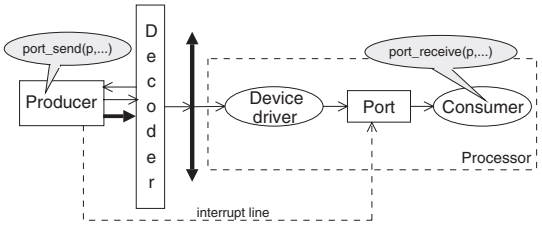


Fig. 5.3 Interface synthesis for hardware-to-software communication

Software-to-Hardware Communication. In software-to-hardware communications, the producer is running in a processor, communicating with a hardware module. In our model, this kind of communication is always memory-mapped. The producer will update a *port* data structure, and a device driver propagates data and events to and from the bus. Events and acknowledge signals are generated for the receiver whenever necessary.

Note that the device driver can be unique for all the software-to-hardware and hardware-to-software communications. It has to monitor a set of software ports, transferring data to the bus, as well as monitor the bus for memory-mapped communications.

Multiprocessor Communication. Finally, in case the processes are mapped to different processors, with different buses, a bridge will also be synthesized. Figure 5.4 shows the proposed architecture. In this scenario, the producer runs on processor 1, connected to system bus 1, whereas the consumer runs on processor 2, connected to system bus 2. The producer will see the bridge as the consumer, characterizing a software-to-hardware communication. Meanwhile, the consumer will see the bridge as the producer, therefore a hardware-to-software communication. The port will be accessed through a memory-mapped address. In addition to the bridge, device driver code is synthesized for both processors, linking the software process to the RTOS and to the bridge hardware.

For shared memory communication, two different architectures are possible, depending on synchronous or asynchronous communication. In the synchronous mode, a locking structure is generated for each shared memory, so that access is granted exclusively to each process. Every memory access has to obtain the lock first. In the asynchronous mode, only the memory is synthesized. The locking mechanism is implicit in the API call for shared memory access. Every shared memory will be directly connected to the system bus, accessible by the central processing unit (CPU). Additionally, a dedicated memory port will be available for each hardware module accessing the memory, so that using the bus is not necessary while accessing shared data. Therefore, there is less contention and higher parallelism in the implementation.

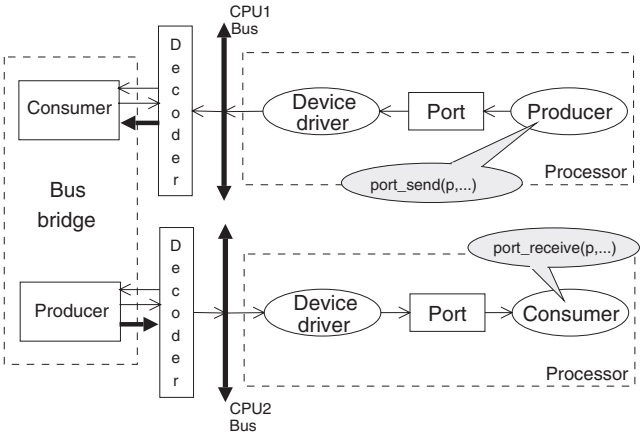


Fig. 5.4 Interface synthesis for multiprocessor communication

4.2 RTOS Synthesis

In addition to communication interface synthesis, the generation of RTOS support is required. In this case, our system-level API has to be mapped to OS-specific resources, adapting the generic API to the functionality available in the target RTOS. Since our API is based on POSIX, the mapping is trivial when targeting a POSIX-compliant OS, like Embedded Linux (ELC, 2006) or eCos (Massa, 2002). Alternatively, our tool is able to target non-POSIX RTOSs by mapping the API calls to the specific RTOS. Finally, the API-based description is used as input to tools that generate a customized OS infrastructure, like Polis (Balarin et al., 1997) and Phantom (Nacul and Givargis, 2004), in case a custom-generated RTOS was specified.

Figure 5.5 shows the code generation process for our system-level API. In Figure 5.5a, the design is specified with the API primitives in a C-like specification. Figure 5.5b shows the generation of C code for a POSIX-compatible OS. The API primitives are expanded to POSIX code. Some primitives have a direct transformation to a POSIX call. Others need to be expanded into more than one POSIX instruction. This is the case with send and receives and synchronous shared memory access.

Note that a data structure is generated for the port-based communication (lines 1 to 7), as discussed earlier. The `port_send` (line 8) and `port_receive` (line 5) are expanded to POSIX/Pthread calls for mutexes and condition variables (lines 20 to 24 and lines 12 to 17), and updates the port data structure, reading data (line 15), sending data (line 21), and setting event flags (lines 16 and 22). The generated code also includes the corresponding checks (line 13) and waits (line 14) in the case of the blocking port receive.

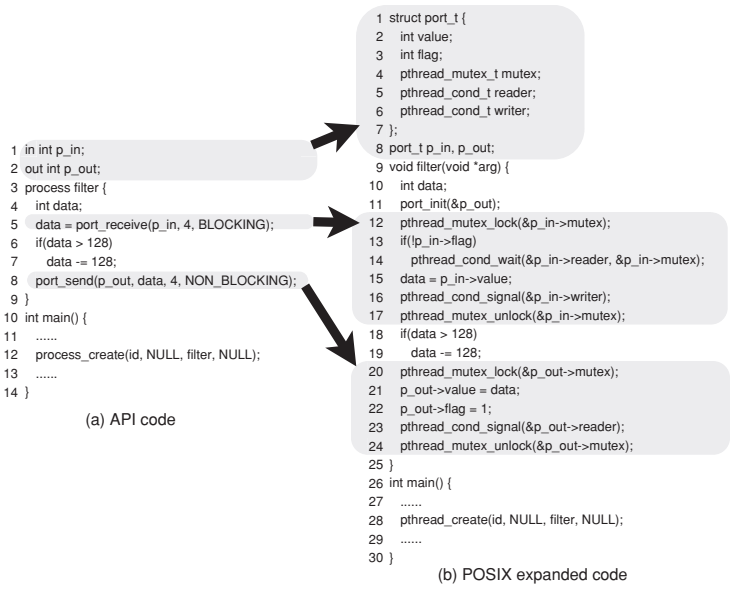


Fig. 5.5 Code example

4.3 Our Hardware/Software Codesign Environment

Our codesign framework provides an interface and a set of tools for synthesizing and simulating a design. Input to our codesign environment is a set of modules $M_1, M_2, \ldots, M_n$ that implement a design. Modules are described in a C-based system-level language, extended with the proposed API functions. Each module represents a process. Next, the mapping step partitions the design into hardware and software implementations. The partitioning granularity is at the process level, i.e., once a process is mapped to hardware or software, all its functionality is synthesized to execute as a hardware block or a software task inside an RTOS. Currently, the partitioning process is manual. Once the design is partitioned, the designer specifies the communication parameters. In case of hardware-to-software communications, for instance, it is possible to determine the use of interrupts. Finally, hardware, software, and interfaces are synthesized.

Hardware synthesis is handled by an in-house behavioral synthesizer that produces synthesizable register transfer level (RTL) for each module. Software modules are generated according to the OS support desired by the designer. For each target software environment, we provide a library that implements the specified API for the referred environment. At that time, our codesign framework can generate software modules based on the POLIS framework (Balarin et al., 1997), the Phantom Compiler (Nacul and Givargis, 2004), and

any POSIX-based OS, like Embedded Linux (ELC, 2006) or eCos (Massa, 2002) with the POSIX adaptation layer. In the latter case, there is a one-to-one mapping of some of the API functions to the POSIX library of the OS, while others require some expansions, as shown in Figure 5.1.

Software is compiled to a specific processor, which can be a NEC V850 or an ARM946. Finally, the interface is generated according to the partition and the communication style specified. We have simulators available that allow us to simulate the synthesized hardware, selected processor (cycle accurate in the case of V850 and instruction based in the case of ARM), software, and communication interfaces.

5. Hardware/Software Integration

The steps of our hardware–software integration process are depicted in Figure 5.6. The block diagram on top helps in visualizing the connections and processes. This example implements a matrix multiplication algorithm consisting of three processes: the index control, which controls the execution of the algorithm; the data retriever, which fetches data from the shared memory Mem, and passes them, two at a time, to the module MAC; MAC module, which multiplies the data, accumulates intermediate results and finally writes the result back into the shared memory. So data retrieve only reads data from the shared memory, whereas MAC only writes into it.

An excerpt of the specification code is shown in the specification section of Figure 5.6. The specification contains some of the API calls proposed in this work incorporated into a C-based specification. This specification is the input to our codesign environment.

The process mapping is specified next. In the example of Figure 5.6, the index control and the data retrieve processes are mapped to software running in a single processor, while the multiplier is mapped to a custom hardware module. During the mapping stage, it is also necessary to specify the options for the communication interfaces. For the ports between index control and data retrieve, the communication will be internal to the processor since it is a software-to-software communication. Therefore, a software port structure will be generated, like the one shown in Figure 5.5b. For the case of the ports between data retrieve and multiplier, we chose a memory-mapped communication to implement the hardware-to-software and software-to-hardware communications. Alternatively, interrupts could have been used for the communication that originates in the multiplier.

In the code generation stage, the environment synthesizes the communication interface, along with the software targeted at the OS specified by the user and the hardware modules. In this stage, the RTOS is adapted to the application. Since there is a memory-mapped communication, the appropriate device

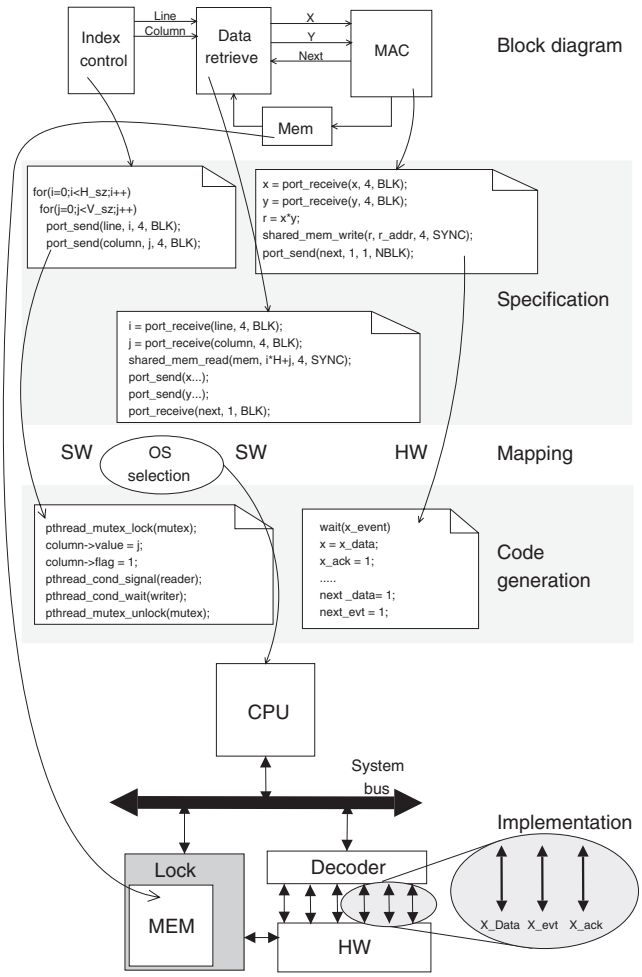


Fig. 5.6 Hardware/Software integration

driver for the communication will be incorporated into the software code. An address decoder is generated for the multiplier process, so that it can access the bus. A locking structure is synthesized around the shared memory block. The lock is needed to support the synchronous operations in the memory access. Software-mapped processes are expanded to a POSIX specification, which can be compiled against POSIX-compliant OSs. The same POSIX code can still be used to generate application-specific OS infrastructure, such as POLIS (Balarin et al., 1997) or Phantom (Nacul and Givargis, 2004).

Meanwhile, hardware-mapped processes result in the generation of a low-level SystemC description, to be synthesized with the appropriate tools. This

detailed description includes the necessary extra hardware and interconnections that implement the hand-shaking control discussed in Section 4.1.

The final architecture is shown in the implementation section of Figure 5.6. The CPU will be executing the two processes mapped to software, supported by the RTOS specified in the mapping stage. The CPU is connected by a system bus to the hardware and shared memory modules. An address decoder connects the hardware to the bus. Finally, the shared memory incorporates the lock to support the synchronous communication specified earlier in the design, providing a dedicated access port to the hardware module. Note the expansion of the connections from the hardware module, with the inclusion of the connections for event and acknowledge for each port from software to hardware and hardware to software. The figure shows the expansion for the X port, and the other ports are expanded similarly.

The final generated hardware and software architectures are simulated in an internally developed, cycle-accurate simulator. We are able to simulate the single and multiprocessor architectures, along with memory, buses, bridges, and reconfigurable logic to implement hardware-mapped modules. Currently, our simulator supports the NEC V850 processor and can provide cycle-accurate execution data. Additionally, we have an instruction-accurate model of the ARM946 processor.

6. Conclusions

Current complexity of embedded systems is driving a consensus toward the need for a higher abstraction level support for system specification. This will result in more opportunities for design reuse and better design space exploration capabilities. In this context, synthesis of OS and hardware/software interfaces is needed.

In this chapter, we have introduced a system-level API that provides a specification support for rapid hardware/software integration by combining into a unified semantics, both transaction-level modeling for hardware specifications and OS and device drivers layers for software specifications. We have shown how this API can be easily integrated in any current system-level design language and we have discussed its utilization into a hardware/software codesign flow.

References

ARM (2003) *AMBA AHB Cycle Level Interface Specification*. ARM Limited.

Balarin, F. et al. (1997) *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Besana, M. and Borgatti, M. (2003) Application mapping to a hardware platform through automated code generation targeting a RTOS. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2003*, Munich, Germany.

Caldari, M., Conti, M., Coppola, M., Curaba, S., Pieralisi, L., and Turchetti, C. (2003) Transaction-level models for AMBA bus architecture using SystemC 2.0. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2003*, Munich, Germany.

Coppola, M., Curaba, S., Grammatikakis, M., and Maruccia, G. (2003) IPSIM: SystemC 3.0 enhancements for communication refinement. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2003*, Munich, Germany.

Dziri, M., Samet, F., Wagner, F., Cesario, W., and Jerraya, A. (2003) Combining architecture exploration and a path to implementation to build a complete SoC design flow from system specification to RTL. In: *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC) 2003*, Kitakyushu, Japan.

ELC (2006) *Homepage of the Embedded Linux Consortium*. The Embedded Linux Consortium (ELC). `http://www.embedded-linux.org/`.

Gerstlauer, A., Doemer, R., Peng, J., and Gajski, D. (2001) *System Design: A Practical Guide With SpecC*. Kluwer Academic Publishers, Boston, MA.

Gerstlauer, A., Yu, H., and Gajski, D. (2003) RTOS modeling for system level design. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2003*, Munich, Germany.

Herrera, F., Posadas, H., Sanchez, P., and Villar, E. (2003) Systematic embedded software generation from SystemC. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2003*, Munich, Germany.

Le Moigne, R., Pasquier, O., and Calvez, J.-P. (2004) A generic RTOS model for real-time systems simulation with SystemC. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2004*, Paris.

Massa, A. (2002) *Embedded Software Development with eCos*. Prentice-Hall, Upper Saddle river, NJ.

Meyerowitz, T., Pinello, C., and Sangiovanni-Vincentelli, A. (2003) A tool for describing and evaluating hierarchical real-time bus scheduling policies. In: *Proceedings of the 40th Design Automation Conference (DAC) 2003*, Anaheim, CA.

Nacul, A. and Givargis, T. (2004) Code partitioning for synthesis of embedded applications with Phantom. In: *Proceedings of the International Conference on Computer Aided Design (ICCAD) 2004*.

OSCI (2006) *Homepage of the SystemC Community*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

Passerone, R., Rowson, J., and Sangiovanni-Vincentelli, A. (1998) Automatic synthesis of interfaces between incompatible protocols. In: *Proceedings of the 35th Design Automation Conference (DAC) 1998*, San Francisco, CA.

The Open Group (2004) *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*. The Open Group and IEEE. `http://www.opengroup.org/onlinepubs/009695399/toc.htm`.

Wind River (2006) *Wind River Homepage*. Wind River Inc. `http://www.windriver.com/`.

Chapter 6

Efficient and Customizable Integration of Temporal Properties into SystemC

Roland J. Weiss, Jürgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13
72076 Tübingen
Germany

Abstract In this chapter, we describe our enhanced extension of SystemC with temporal properties. The two main tasks involved are the synthesis of checker automata corresponding to given temporal logics formulae and the integration of property specification into the SystemC framework.

For synthesis of checker automata we rely on an intermediate language (IL) that is directly generated from property specification language (PSL) or finite linear temporal logic (FLTL) properties in a bottom-up fashion. IL can either be translated to native code in a preprocessing step or be executed directly in a simple virtual machine. Our temporal SystemC checker supports both alternatives.

Properties can be added dynamically any time during simulation. A dedicated observer process is responsible for checking all active properties continuously against the current system state during simulation. The actual executing property checkers can be customized according to the users needs. When special states are reached, i.e., finding a validation or violation of the property, a policy class dispatches to user-defined action handlers.

1. Introduction

The complexity of modern hardware/software systems requires more rigorous validation procedures in the development process. One promising approach is reusing formal specifications written in temporal logics during all phases of the design process. After requirements analysis, the informal specification is cast into temporal properties that capture the design intent. This formalization of the requirements improves the understanding of the new system. Then, the system model and implementation can be annotated with the same temporal properties. However, the executing system is now monitored for validation or violation of

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 101–114.

the properties. Special actions can be performed upon reaching these special system states, ranging from simple logging activities up to throwing exceptions or error recovery attempts.

The described methodology suffers from the typical deficit of simulation-based approaches to system validation which is incomplete coverage, i.e., only the paths executed during simulation can be monitored. No statements about the unexplored system states can be made. However, because the properties are already formalized, they can be reused if parts of the system are subjected to fully formal verification techniques like model checking (Clarke et al., 1999). Furthermore, the actual system coverage can be measured with respect to the specified properties.

This chapter describes in detail the integration of temporal properties into SystemC (Grötker et al., 2002), which meets the above characteristics. The framework is called SystemC temporal checker (SCTC). We focus on enabling an assertion-based design methodology (Coelho and Foster, 2004) in SystemC. To achieve this, several tasks have to be met. First, a mechanism has to be devised for adding properties to SystemC designs. These properties have to be synthesized into a form that is executable together with the SystemC model. Finally, the designer should be able to customize the actions taken when properties trigger, which we support by a policy-based design (Alexandrescu, 2001) of our checking engine. The advantages of assertion-based design are the following:

- A formalization of the design intent improves the understanding of the design.
- Temporal properties enhance the communication between involved parties by unambiguously capturing the system requirements.
- Assertions allow identifying problems close to the real error source and taking appropriate actions.

The rest of the chapter is organized as follows. Next, we detail the process of property synthesis to the intermediate language (IL). Then we describe the integration of temporal properties into SystemC and give some experimental results. Finally, we conclude and give a brief outlook on future work.

2. Property Synthesis

At the user level, properties are written in a property specification language (PSL). 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 behavior of the verification tools, e.g., if a property violation should stop the verification process or simply 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.

In this chapter, we concentrate on properties in temporal logics with a linear time model, which is well suited for simulation contexts. Our synthesis engine supports the PSL (Accellera, 2004) and finite linear temporal logic (FLTL; Ruf et al., 2001), an extension to linear temporal logic (LTL) with time bounds on the temporal operators.

The main task of the synthesis engine is to convert the plain text property specification into a format that can be executed during system monitoring. We use accept–reject automata (AR-automata; Ruf et al., 2001) for this purpose. AR-automata can detect validation or violation of properties on finite system traces, or they stay in pending state if no decision can be made yet. Figure 6.1 shows the AR-automaton that corresponds to the FLTL property $G(req \rightarrow F[2]\ ack)$.

Our first approach implemented in Java converted FLTL formulae into AR-automata using a straightforward automata representation. The SystemC checker reads the result from a text file describing the AR-automaton. Notice that this description contains a complete enumeration of all transitions for all combinations of input variables. Thus, the memory requirements are $n \times 2^{|I|}$, where $|I|$ denotes the number of input variables and n is the number of states of the AR-automaton. The size of the AR-automaton is minimized using a partitioning algorithm that merges bisimilar states (Milner, 1980; Ruf et al., 2001). Because of performance problems when synthesizing larger formulas, we developed a new synthesis engine (Krebs and Ruf, 2003). The new engine

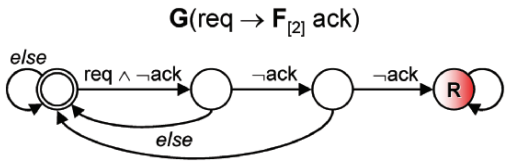


Fig. 6.1 Example of an AR-automaton for a simple FLTL property. The state labeled with R is the rejecting state.

translates a temporal formula to IL, an executable representation for AR-automata.

2.1 Intermediate Language

The motivation for IL is producing a space-efficient and executable representation of properties for the validation process. The commands available in IL can be grouped into four categories: time, compare, branch, and return statements. Table 6.1 shows the IL statements.

The translation of LTL formulae to IL converts temporal operators into sequences of IL statements. This algorithm works bottom-up and merges subformulae until the whole expression is translated. The main operation is merging two subformulae. More details are available in Krebs and Ruf (2003). The IL code for formula $G(a \rightarrow X\ b)$ is given in Listing 6.1.

This encoding does not represent every state transition of an AR-automaton explicitly. Moreover, the transitions are given implicitly by paths through the IL code ending in either a return or wait statement. Experiments in Krebs and Ruf (2003) show that the new translation scheme is usually orders of magnitude faster than the traditional one.

Once a property has been translated to IL code there are several options how to use the IL code. First, IL code can be further translated into code or data structures native to the target platform, e.g., SystemC. Second, the IL code can be executed directly in a virtual machine. We currently support both alternatives. Finally, another option is to synthesize hardware from IL code and use it for monitoring the final system. Figure 6.2 summarizes the IL approach.

3. Integrating Temporal Properties into SystemC

SystemC (Grötker et al., 2002) is a C++ library developed to support modeling not only at the system level but also at other levels of abstraction, such as register transfer level (RTL). The modeled systems may be composed of both 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 to transactional level. The SystemC library and reference implementation of the simulation kernel are available for free in source code (OSCI, 2006).

Table 6.1 The categorized IL statements

Category	*Statement*	*Semantics*
Time	**WAIT** *n*	Wait *n* steps
Compare	**CHK** *s*	Check signal *s*
Branch	**JMP** *n* **JEQ** *n* **JNE** *n*	Jump to address *n* (possibly depending on previous **CHK**)
Return	**RNE** T/F **RET** T/F **REQ** T/F	Terminate with true/false result (possibly depending on previous **CHK**)

Listing 6.1 The IL code for formula $G(a \rightarrow X\, b)$. The left column gives the code location and the statement's opcode, separated by a colon.

```
1  00000000:00000001 CHK 00000000
2  00000004:0000002a JEQ 0000002c
3  00000008:0000005f WAIT 1,5
4  0000000c:00000001 CHK 00000000
5  00000010:00000012 JEQ 00000020
6  00000014:00000011 CHK 00000010
7  00000018:fffffff0 JNE 00000008
8  0000001c:1fffffff RET 0
9  00000020:00000011 CHK 00000010
10 00000024:00000008 JNE 0000002c
11 00000028:1fffffff RET 0
12 0000002c:0000005f WAIT 1,5
13 00000030:fffffd0 JMP 00000000
```

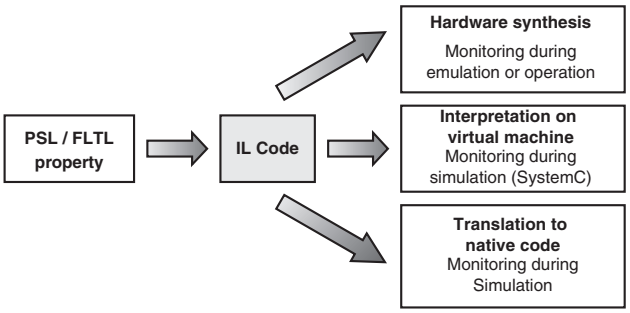


Fig. 6.2 Outline of the IL approach

However, SystemC currently does not contain a mechanism for specifying and checking temporal properties. In this section, we will look into extending SystemC with these abilities based on the IL synthesis engine described in Section 2.

3.1 Property Specification

C++, and therefore SystemC also, provides no language mechanism for temporal property specification. There are several ways to deal with this problem. One common approach is to hide the specification in comments (Chen and Roşu, 2005) or external files and instrument the system code in a preprocessing phase. Also, a library can provide functions to trigger property checking. The property specification itself is given as a string in the function call. The advantage of the first approach is that the entities in the specification can relate to source-level constructs like variable names. However, a separate tool is needed to preprocess the code. The latter technique is more easy to handle by the user, but entities in the specification have to refer to source code constructs by name. For example, in SystemC all signals are given a unique name and can therefore be used in such specifications. In addition, specifications are treated as first-class citizens of the code, which makes it much easier to control the addition of properties dynamically.

We support property specification in SystemC through a library extension. A base class `Proposition` allows wrapping arbitrary source code entities as named objects. A default implementation for SystemC signals exists.

3.2 Property Checking

After deciding how to specify the properties, the second major task is to determine how to check the property. A thorough taxonomy of design decisions is given in Chen et al. (2004). The main aspects are:

Running location: The validation process is either executed *in-line*, i.e., in the same process as the system under test (SUT), or *out-line*, i.e., the checker runs in a different process or on another machine.

Running time: The validation process is either executed *on-line*, i.e., together with the SUT, or *off-line*, i.e., a system trace is inspected by an external tool after system execution.

Scope: The scope attribute defines how the property relates to the system and thus fixes the checking semantics of the property. In Chen et al. (2004) the attributes *class*, *method*, *block*, and *checkpoint* are defined.

The taxonomy lacks in two regards. All scope attributes relate to single objects. We think that for system validation an attribute for checking interacting

Listing 6.2 The main loop of the checker process

```
1 checkerLoop(in: activationQueue, activeList)
2  for all properties p_i in activationQueue
3     if p_i.timestamp < now
4          activeList.append(p_i)
5          activationQueue.remove(p_i)
6  for all properties p_i in activeList
7     p_i.check()
```

components is essential. However, in order to allow intercomponent property checking, a notion of time has to be introduced when to sample the system state. We call this *system* scope. In SystemC, and other hardware-centric description languages, properties can be bound to clocks. For pure software systems, this is not as obvious.

Also, an attribute is missing for stating when a property should be activated. This is especially important for static specifications in comments or external files.

Our temporal SystemC checker is realized as a separate module with a thread process dedicated to executing the IL code corresponding to active properties. The main problem for nonintrusively integrating a checker process into the simulation kernel is SystemC's lack of scheduling features. The order in which processes are executed is undetermined. The main consequence of this behavior is that only the state of signals can be asserted across module boundaries. If the checker process would be called before or after all process execute during one simulation cycle, a stable snapshot of the system would be available for property checking. We deal with this problem by introducing an activation queue. Newly activated properties perform their initial check immediately and are then appended to the activation queue. The checker process in the current cycle then adds all properties from the previous cycle to its active list. Finally, the properties in the active list are executed. The checker main loop is shown in Listing 6.2.

The `check()` method depends on the representation of the AR-automata. For an exhaustive enumeration of the transitions, the valuation of all propositions present in the property provides an index into the current state's transition table. Then the next state is checked for being the automaton's accept (reject) state, thus indicating validation (violation) of the property. Otherwise, the automaton remains in pending state.

If we execute IL code directly in method `check()`, we have to follow the control flow of the code, possibly checking the current value of encountered input signals. This process ends either by reaching a wait statement, thus leaving

the property in pending state, or reaching a return statement, thus finding a validation or violation of the property.

Finally, notice that checking the specification can be reduced to calling the `check()` method on active properties and wrapping input variables by subclassing from class `Proposition`, thus making them named entities. The SystemC specific parts are captured by the checker module, which calls method `check()`, and subclasses of `Proposition` for handling SystemC data types like signals. The core checking engine can be adapted easily to other sampling sources, like event loops of a graphical user interface (GUI) framework.

3.3 Customizing Actions with Policies

For the user annotating code with temporal assertions, it is of primary importance to be able to take special actions depending on the state of the checked properties. In C++, policy-based design (Alexandrescu, 2001) has attracted considerable interest recently. The idea is to orthogonally decompose a class' behavior into *policies*. Policies are themselves classes that are passed as template parameters to the original class. The original class is therefore responsible for implementing its behavior of its policy classes. In our case an action policy allows customizing a property's checking process. Every legal action policy has to implement these functions:

OnInitial(Property& p): Call this function when the check method is executed the first time.

OnAccepting(Property& p): Call this function when the check method encounters an accepting state.

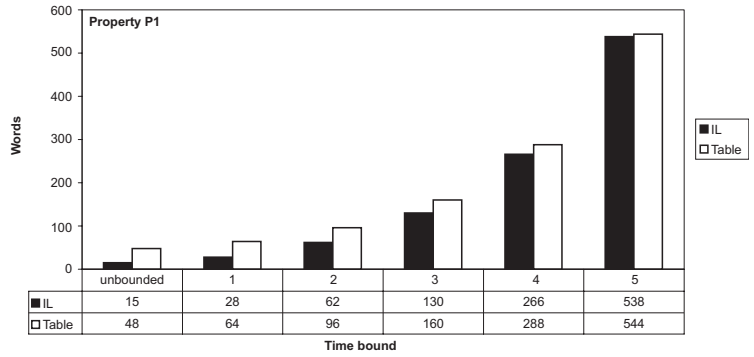
OnRejecting(Property& p): Call this function when the check method encounters a rejecting state.

OnPending(Property& p): Call this function when the check method leaves the property in pending state.

Default implementations are provided for action policies like throwing exceptions on encountering accept or reject states, or just logging the progress of the property-checking process.

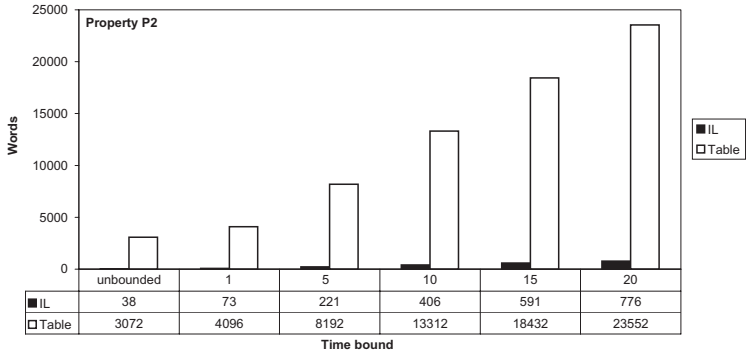
4. Experimental Results

The significant improvement of the IL property synthesis algorithm over the automata-based Java implementation has already been shown in Krebs and Ruf (2003). Therefore, we concentrate on comparing the run-time performance and memory requirements of an exhaustive transition table against IL code. In the first case, the data structures are created on-the-fly from IL code. In the second case, IL code is executed in its virtual machine.



	unbounded	1	2	3	4	5
IL	15	28	62	130	266	538
Table	48	64	96	160	288	544

(a) Property P1



	unbounded	1	5	10	15	20
IL	38	73	221	406	591	776
Table	3072	4096	8192	13312	18432	23552

(b) Property P2

Fig. 6.3 Memory consumption for properties P1 and P2

4.1 Memory Consumption

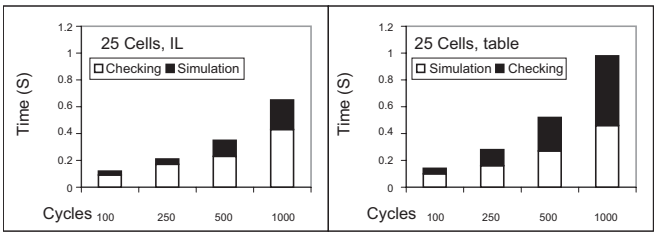
We have examined how much memory-representing properties actually requires.

Property P1 is $G(a \rightarrow F[n](b \vee c \wedge d))$. The results are shown in Figure 6.3a. Both IL and the table approach consume approximately the same amount of memory. The memory consumed grows exponentially with the time bound for this property.

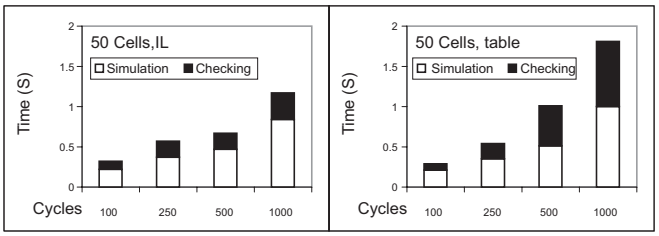
Property P2 checks in a holonic transportation system that two robots moving in the system do not collide. The position is encoded with 5 b for every holon, thus the property contains 10 propositions. The results for translating this property with different time bounds is given in Figure 6.3b. We see that because more propositions are part of the property, the memory consumption grows very large for the table approach.

4.2 Run-time Performance

In these experiments, we checked two properties against a scalable model of an arbiter (McMillan, 1992). The arbiter eventually acknowledges a request by a



(a) Arbiter example with 25 cells



(b) Arbiter example with 50 cells

Fig. 6.4 Run-time comparison for arbiter example

cell within $2n$ cycles, where n is the number of cells. The first property checks that at most one acknowledge is given each cycle. The second property checks that after a request signal, the acknowledge signal comes within the required time frame.

Figure 6.4a shows the time taken by the simulator and the time consumed by the the checker module for running a given number of cycles. On the left-hand side the times are shown for the checker that works directly on IL code, whereas on the right-hand side the table-driven approach is given. As expected, the simulator requires the same time in both cases. However, the IL module performs better than the table driven approach. The reason for this is that in the table-driven approach checking the termination condition takes longer.

Figure 6.4b shows the same test, however, with an arbiter consisting of 50 cells. The most noteworthy point is that the overhead of the checker slightly decreases. However, again the IL code checker performs better.

5. Related Work

Related work can be divided into two sections. On the one hand, we discuss algorithms for generating automata or isomorphic representations like IL from property formulae. On the other hand, we examine frameworks that allow adding property specifications, and temporal expressions in particular, to system models.

The specification framework most closely related to SCTC is monitoring-oriented programming (MOP; Chen et al., 2004; Chen and Roşu, 2005). Their monitors can be interpreted as AR-automata. It would be interesting to use IL

as monitor generator in MOP and use their monitor generators for SCTC. Both frameworks allow the exchange of the underlying property synthesis engine. MOP is tailored toward monitoring general Java code and uses a preprocessing phase to instrument Java source code. As mentioned in Section 3.2, they allow properties only at the object, method, block or code line level, whereas SCTC supports checking at the system level. We extended their taxonomy in this respect. No framework that supports temporal assertions in C++ is known to the authors. However, a few commercial offerings add temporal checking to SystemC.

Translating LTL formulae into Buechi automata has been covered in literature (Gerth et al., 1995; Daniele et al., 1999; Somenzi and Bloem, 2000) over the years. Buechi automata are an important representation for model checking LTL formulae.[1] The checking algorithm for AR-automata is simpler but sacrifices completeness as it cannot handle loops, which are intrinsic in algorithms on Buechi automata. One can make Buechi deterministic and minimize the resulting automata to get monitors; however, this approach is not very efficient because it requires multiple passes over the automata.

In Sen and Roşu (2003) and Sen et al. (2003), the authors describe a novel one-pass algorithm for generating optimal monitors using a rewriting method called circular coinduction for properties formulated with regular expressions and LTL, respectively. An algorithm for generating a dynamic programming algorithm corresponding to a past time temporal logic formula is presented in Havelund and Roşu (2002).

None of the discussed approaches handles PSL and time bounds on temporal operators. Also, none of the approaches uses an IL executable in a virtual machine for representing monitors.

6. Conclusions and Future Work

We presented a framework for annotating SystemC designs with temporal expressions. An efficient property synthesis engine is employed for turning the high-level formulae into executable intermediate code.

Future work aims at further enhancing the IL checker generator. This is done by transferring and applying automata algorithms directly on IL code. Also, various specific optimizations for IL are explored, like caching repetitive code sequences or selective compilation of subformulae.

We also explore ways to extend the scope of the SystemC checker to general C++ code. Here, the main task is to come up with an extensible scheme to define sampling points for creating a notion of time necessary for checking temporal expressions. Then, the checker can also be used to validate SystemC

[1] We also use AR-automata in a formal verification algorithm (Ruf et al., 2003).

models at the transactional level. Finally, an updated simulation kernel with special handling of the checker process is being developed.

Acknowledgments

This work has in part been funded by the German Research Council (DFG) within projects GRASP and KOMFORT and by the BMBF and edacentrum within project FEST.

References

Accellera (2004) *Property Specification Language (PSL), Version 1.1*. Accellera. `http://www.eda.org/vfv/`.

Alexandrescu, Andrei (2001) *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, MA.

Chen, Feng, D'Amorim, Marcelo, and Roşu, Grigore (2004) Monitoring-oriented programming: a tool-supported methodology for higher quality object-oriented software. Technical Report UIUCDCS-R-2004-2420, University of Illinois at Urbana-Champaign.

Chen, Feng and Roşu, Grigore (2005) Java-MOP: a monitoring oriented programming environment for Java. In: Halbwachs, Nicolas and Zuck, Lenore D. (eds) *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2005*, volume 3440 of *Lecture Notes in Computer Science*, Springer Edinburgh, UK, pp. 546–550.

Clarke, Edmund M., Grumberg, Orna, and Peled, Doron E. (1999) *Model Checking*. MIT Press, Cambridge, MA.

Coelho, Claudionor Nunes and Foster, Harry D. (2004) Asserstion-based verification. In: Drechsler, Rolf (ed) *Advanced Formal Verification*. Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 167–204.

Daniele, Marco, Giunchiglia, Fausto, and Vardi, Moshe Y. (1999) Improved automata generation for linear temporal logic. In: Halbwachs, Nicolas and Peled, Doron (eds) *Proceedings of the 11th International Conference on Computer Aided Verification (CAV) 1999*, volume 1633 of *Lecture Notes in Computer Science*. Springer, Trento, Italy, pp. 249–260.

Gerth, Rob, Peled, Doron, Vardi, Moshe Y., and Wolper, Pierre (1995) Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, Piotr and Sredniawa, Marek (eds) *Proceedings of the 15th IFIP WG6.1*

International Symposium on Protocol Specification, Testing and Verification (PSTV) 1995, volume 38 of *IFIP Conference Proceedings*. Chapman & Hall, Warsaw, Poland, pp. 3–18.

Grötker, Thorsten, Liao, Stan, Martin, Grant, and Swan, Stuart (2002) *System Design with SystemC*. Kluwer Academic Publishers, Boston, MA.

Havelund, Klaus and Roşu, Grigore (2002) Synthesizing monitors for safety properties. In: Katoen, Joost-Pieter and Stevens, Perdita, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2002*, volume 2280 of *Lecture Notes in Computer Science*. Springer, Grenoble, France, pp. 342–356.

ISO/IEC (2003) *Programming Languages—C++*. International Organization for Standardization (ISO), 2nd edn. Standard ISO/IEC 14882:2003 of Committee JTC 1/SC 22/WG 21.

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.

McMillan, Kenneth L. (1992) *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Milner, Robin (1980) *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, New York.

OSCI (2006) *Homepage of the SystemC Community*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

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 (eds) *Proceedings of Design, Automation and Test in Europe (DATE) 2001*. IEEE Press, Munich, Germany, pp. 742–748.

Ruf, Jürgen, Peranandam, Prakash M., Kropf, Thomas, and Rosenstiel, Wolfgang (2003) Bounded property checking with symbolic simulation. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2003*, Frankfurt, Germany.

Sen, Koushik and Roşu, Grigore (2003) Generating optimal monitors for extended regular expressions. In: *Proceedings of the Third Workshop on Runtime Verification (RV) 2003*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier B. V., Boulder, CO.

Sen, Koushik, Roşu, Grigore, and Agha, Gul (2003) Generating optimal linear temporal logic monitors by coinduction. In: Saraswat, Vijay A. (ed) *Proceedings of the 8th Asian Computing Science Conference on Programming Languages and Distributed Computation*, volume 2896 of *Lecture Notes in Computer Science*. Springer, Mumbai, India, pp. 260–275.

Somenzi, Fabio and Bloem, Roderick (2000) Efficient Büchi automata from LTL formulae. In: Emerson, E. Allen and Sistla, A. Prasad (eds) *Proceedings of the 12th International Conference on Computer Aided Verification (CAV) 2000*, volume 1855 of *Lecture Notes in Computer Science*. Springer, Chicago, IL, pp. 248–263.

Chapter 7

UMoC++: A C++-Based Multi-MoC Modeling Environment

Deepak A. Mathaikutty[1], Hiren D. Patel[1], Sandeep K. Shukla[1], and Axel Jantsch[2]
[1] *Center for Embedded Systems for Critical Applications*
Virginia Tech
Blacksburg, Virginia
USA
[2] *Department of Microelectronics and Information Technology*
Royal Institute of Technology
Stockholm
Sweden

Abstract System-on-chip (SoC) and other complex distributed hardware/software systems contain heterogeneous components that necessitate frameworks capable of expressing heterogeneous models of computation (MoCs) for modeling their functionalities. System-level design languages (SLDLs) that facilitate multi-MoC modeling should have well-defined semantics and should be readily subjected to formal analysis to handle the design complexity. As a result, we follow the multi-MoC paradigm based on timing abstraction and functional parameterizations that have rigorous denotational semantics, which are compliant to functional idioms as shown in functional frameworks such as ForSyDe and SML-Sys. However, functional frameworks are not widely used in the industry due to issues related to efficiency and interoperability with other widely used SLDLs. This imposes a requirement for an imperative language-based implementation of these generic MoCs that offers all the advantages of the underlying formal semantics. In this chapter, we formulate the basis for having generic MoCs in an imperative language and describe the implementation of an untimed modeling framework called UMoC++.

Keywords: model of computation; heterogeneity; untimed; function semantics; functional language.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 115–130.

1. Introduction

System-level modeling for system-on-chip (SoC) and hardware/(solidus) software codesign has been gaining importance due to the raising complexity of such systems, continual advances in semiconductor technology, and productivity gap in design. Efforts toward mitigating the productivity gap has led to the evolution of several design methodologies of which the one we address are system-level design languages (SLDLs). Examples of recently introduced system-level modeling languages are SpecC, SystemC (OSCI, 2006), and System Verilog (Accellera, 2006). In addition, most system models for SoCs are heterogeneous in nature and encompass multiple models of computation (MoCs) in their different components. As a result, we need a framework that provides a way to express heterogeneous MoCs for modeling SoCs that have well-defined formal semantics and is readily amenable to formal verification.

The multi-MoC paradigm discussed in Jantsch (2003) describes the function-based semantic definition of MoCs that provide a generic classification of computational models by abstracting time. These generic MoCs have well-defined denotational semantics that make them readily subjected to formal analysis and further amenable to functional paradigms. Hence, a functional language (FL) can easily be used to implement them and create a modeling environment. One such modeling framework is ForSyDe (Sander and Jantsch, 2004) built on the semantics of a synchronous computational model (Jantsch, 2003) that facilitates the application of formal methods for transformation and synthesis. However, ForSyDe is more compliant to applications that are synchronous in nature. On the other hand, we need a framework such as SML-Sys (FERMAT, 2006) that provides multiple MoCs for heterogeneous system design. We implemented SML-Sys in a functional language called Standard ML (SML; Milner et al., 1997). Some of the advantages of the SML-Sys framework are: (i) its generic design makes it highly expressible and easily extensible; (ii) precise semantics of the model are derivable because of the rigorous denotational semantics of functional programs; (iii) formal verification of the functionalities of the models can be achieved without having to parse through the C++/Java language specifics; (iv) design transformations are function applications, which provide clean and precise refinement semantics in the framework.

FLs are mainly used in academia for conceptualization and development of theoretical basis for research. It is not common practice to use FLs in the industry for hardware/software design, since the industry prefers imperative languages like C/C++ that have a lower learning curve and provides faster simulation results and requires less memory. The design of imperative languages is based on the von Neumann model, whereas that of FLs are based on mathematical functions. The formulation of generic MoCs in an FL like SML-Sys is inefficient (Okasaki, 1992) due to the implementation cost for single

assignment, call by value and recursion that bring in a lot of implicit garbage collection, potentially huge stack, and lots of copying. Furthermore, designers accustomed to the imperative notion of programming find it difficult to relate and work with FLs due to their abstract expressibility. Most industrial intellectual properties (IPs) are C or C++-based. With the current trend toward IP-based integration, FL-based model development brings in issues of interoperability and reusability. Hence, there is a necessity for an imperative language-based implementation of generic MoCs that takes advantage of features in FL to attain a well-defined formalism and further can be used for cosimulation with other SLDLs to facilitate interoperability and IP-based integration.

2. Related Work

The two most prominent works in classification of MoCs were done in the context of the Ptolemy II project and the ForSyDe project (Sander and Jantsch, 2004). Ptolemy II is built with multiple MoCs, which include various sequential MoCs such as finite state machine (FSM), discrete-time, continuous-time, and MoCs of interacting entities, such as communicating sequential process (CSP), Kahn process network (KPN), etc. The other distinguishing classification of MoCs was done by abstracting time and functional parameterizations. This work can be distinguished from Ptolemy's work as a distinction of the denotational view versus operational view of MoCs (Patel and Shukla, 2004).

ForSyDe is a library-based implementation that provides a computational model for the synchronous domain with interfaces implemented in Haskell. However, since ForSyDe has a single computational model based on the synchrony assumption, it is best suited for applications amenable to synchrony. SML-Sys, when compared with ForSyDe, has a higher modeling fidelity[1] (Patel and Shukla, 2004) since it is a multi-MoC modeling framework based on the generic definition in Jantsch (2003), which is an extension of the ForSyDe methodology.

3. Generic MoCs

We briefly introduce the generic MoCs defined in Jantsch (2003). These MoCs are built on processes, events, and signals. *Events* are the elementary units of information exchanged between processes. *Processes* receive or consume events and they send or emit events. *Signals* are finite or infinite sequence of events. The activity of processes is divided into *evaluation cycles*. A process partitions its input and output signals into subsequences corresponding to its evaluation cycles. During each evaluation cycle a process consumes exactly one subsequence of each of its input signals. To relate functions on events to

[1] how close it is to the conceptual MoC

processes we introduce process constructors, which are parameterizable structures that instantiate processes. Furthermore, we define process combinators to construct process networks (PNs) through process compositions.

An MoC is defined as a set of processes and PNs that are constructed from the given set of process constructors and combinators. We finally categorize the MoC based on "how the processes communicate and synchronize" with other processes and, in particular, with the "timing information" available to and used by the process.

Definition 3.1. In Jantsch (2003), an MoC = (C, O) is defined as a 2-tuple where C is a set of process constructors, each of which, when given constructor-specific parameters, instantiates a process. O is a set of process composition operators, which when given processes as arguments instantiates a new process.

MoCs are characterized by the duration of their evaluation cycles. The three generic MoCs defined in Jantsch (2003) are: untimed MoC (UMoC), synchronous MoC, and timed MoC.

Untimed MoC: Processes communicate and synchronize with other processes without the notion of time such that only the order of events are relevant.

Synchronous MoC: The Synchronous MoC divides the timeline into intervals. Every computation within an interval occurs at the same time, but the intervals are totally ordered along the timeline. In synchronous MoCs, the evaluation cycle of processes lasts exactly one time interval. We further categorize synchronous MoCs into:

- **Perfect Synchronous MoC:** This MoC is built on the basis of the *perfect synchrony hypothesis* (Jantsch, 2003), where the output events of a process occur in the same time interval as the corresponding input events.
- **Clocked Synchronous MoC:** This MoC is based on the *clocked synchronous hypothesis* (Jantsch, 2003). It differs from the perfectly synchronous MoC in that every process incurs a delay from an input event to an output event.

Timed MoC: This MoC is a generalization of the synchronous MoC. Timing information is conveyed on the signals by transmitting absent events at regular time intervals.

3.1 Preliminary Notations

We introduce few notations used in defining and distinguishing the generic MoCs. The set of values V represents the data communicated over a signal and the set E constitutes events containing values. A sequence of events constitutes

a signal. Processes are defined as functions on signals p: $S \rightarrow S$ that is a mapping between signal sets. Furthermore, they are allowed to have internal state such that for the same given input signal they react differently at different time instances.

3.2 Generic MoCs Formulation in SML-Sys

In this section, we briefly discuss the implementation of the UMoC for the SML-Sys framework (Mathaikutty et al., 2004). For these generic MoCs, finite signals are implemented as generic lists as shown below:

```
1 (* Definition of a Finite signal *)
2 datatype signal = nil | 'a :: 'a list
```

3.3 Untimed Model of Computation

UMoC adopts the simplest timing model, corresponding to the causality abstraction. Processes, modeled as state machines, are connected to each other through signals. Signals transport data values from a sending process to a receiving process. The data values do not carry time information, but the signals preserve the order of emission.

Process Constructors. In the UMoC, process constructors are higher order functions that take functions on events as argument and instantiate processes. We implement a set of process constructors that are used to define computational blocks which are either complex processes or process networks. We suffix the name with U to designate it to the UMoC. We discuss the implementation details of a Mealy-based process constructor with respect to finite signals.

Definition 3.2 (Mealy-based process constructor)**.** It resembles a Mealy-based state machine with the addition of a next-state function, an output encoding f that depends on both the input partition and the current state: $mealyU(\gamma, g, f, \omega_0) = p$, where $p(s) = \acute{s}$, with $\Psi(v, s) = \langle a_i \rangle$, where $v(i) = \gamma(\omega_i)$, $g(a_i, \omega_i) = \omega_{i+1}$, $f(\omega_i, a_i) = \acute{a}_i$, and $s, \acute{s}, a_i, \acute{a}_i \in S$, $\omega_i \in E$, $i \in N$.

The list of elementary process constructors implemented for the UMoC is shown in Mathaikutty et al. (2005). The Mealy-based process constructor shown in Listing 7.1 can be extended to handle multiple inputs making it more generic, which results in simplifying the above list.

Process Combinators. We define compositional operators to combine different processes to form complex processes and PNs. These are also implemented as higher-order functions (HoFs). The sequential, parallel, and

Listing 7.1 Mealy-based process constructor in SML-Sys

```
(* mealy-based process constructor for the UMoC *)
fun mealyU (h, g, f, w) = fn (s) => constructor (h, g, f, w, s)

fun constructor (_, _, _, _, []) = [] | constructor (h, g, f, ↵
    w, s) = f (w, head (partition([h w], s))) @ constructor ↵
    (h, g, f, g (w, partition ([h w], s)), drop (s, (h w)))
```

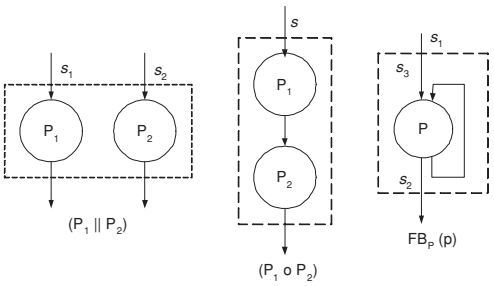


Fig. 7.1 Parallel, sequential, and feedback operators

feedback operators are shown in Figure 7.1. We discuss the implementation of the sequential and the feedback operators with respect to finite signals.

Definition 3.3 (Sequential composition operator)**.** Let p_1 and p_2 be two processes with one input and one output each, and let s ∈ S be a signal. Their sequential composition denoted by $p_1 \times p_2$, is defined as follows: $(p_1 \times p_2)(s) = p_2(p_1(s))$.

```
(* sequential composition operator *)
fun seqcomp (p1,p2) = fn (s) => p2 (p1 (s))
```

Definition 3.4 (Feedback composition operator)**.** Given a process $p : (S \times S) \rightarrow (S \times S)$ with two input signals and two output signals, we define the process $\mathrm{FB}_p(p) : S \rightarrow S$. The behavior of the process $\mathrm{FB}_p(p)$ is defined by the least fixed-point semantics: $\mathrm{FB}_p(p)(s_1) = s_2$, where $p(s_1, s_2) = (s_2, s_3)$.

```
(* Feedback Operator *)
fun fb (p) (s) = fixpt(p, s, [], length(s) + 1)
fun fixpt (q, s, sout, 0) = sout | fixpt (q, s, sout, n) = ↵
    fixpt (q, s, (q s sout), n - 1)
```

Formalized Definition of UMoC.

Definition 3.5 (UMoC)**.** UMoC is defined as MoC $= (C, O)$, where $C = \{mapU, scanU, scandU, mealyU, mooreU, zipU, zipUs, zipWithU, unzipU, sourceU, sinkU, initU\}$ and $O = \{||, \circ, FB_p\}$

4. Essential Concepts from FL mapped to C++ for our Implementation

The Standard Template Library (STL; Musser and Saini, 2001) is a C++-based generic library of container classes, algorithms, and iterators that are heavily parameterized through templates. Templates allow a generic component to take type `T`, where `T` can be replaced with the actual type. As a result the operations and element manipulations are identical regardless of the type of component, thereby facilitating a way to reuse source code. C++ provides two kinds of templates: class templates and function templates. Function templates are used to write generic functions that can be used with arbitrary types, whereas class templates are usually used as adaptive[2] storage containers. In order to mimic the advantages of FL languages offered by some key concepts like polymorphic types, HoF, and partial applications, we implement similar facilities in C++ using STL skeletons. In the following subsections, these functional concepts and their implementation using C++ are discussed.

4.1 Polymorphic Types

Polymorphism is a type discipline that allows one to write functions that can act on values of multiple types in a uniform way. C++ supports parametric polymorphism by means of templates. A template definition consists of a list of type variables, followed by the definition of a function, a class member function, or a class. Here is an example of a function template to compute the sum of two values:

```
1  template <typename T> T sum(T a, T b) { return a + b; }
```

where `T` is the placeholder for any built-in or user-defined C++ type. The substitution with concrete types is not transparent to the user. The different instances of `sum` are distinguished by the overloading mechanism of C++. For example, if the user passes two integer values to `sum`, the compiler automatically instantiates `int sum(int, int)`. For a user-defined type, the + operator needs to be defined.

[2] reusable and efficient

4.2 Higher-Order Functions

HOFs (Okasaki, 1992) are functions that take functions as arguments and/or return functions as result. Even though the term *higher-order function* is from the FL community, C++ STL also contains many examples of HOFs (e.g., `for_each`, `transform`; Musser and Saini, 2001). HOFs are implemented in C++ using function templates and the ability to overload the function call operator (**`operator()`**). An example of an HOF is shown below where the unary function `f` takes a value of type `x` as argument and applies `f` to `x` once:

```
1 template <typename OPR, typename ARG> ARG apply(OPR f, ARG x)
2 { return f(x); }
```

`OPR` is the placeholder for arbitrary C++ types that support the function call syntax such as pointers to functions and classes that overload the function call operator:

```
1 template <typename T>
2 struct Myfunc { T operator() (T c) { return c + c; } };
```

The overloaded **`operator()`** permits objects of type `Myfunc`[3] to be used as if they were ordinary C++ functions. Such objects are called functors or functional objects (Kuchën and Striegnitz, 2002).

4.3 Partial Application

Passing less than n arguments to an n-ary function is called partial application. Partial application (Okasaki, 1992) semantically means binding the first argument of an n-ary function to some fixed value. In C++, support for partial application is limited to binary functions using the `std::bind1st` and `std::bind2nd` constructs. To partially apply a function using these constructs, the user must create a wrapper class for the ordinary C++ function or use the `std::ptr_fun` adaptor that automatically generates an appropriate object. Consider a binary C++ function **`float`** `add(`**`float`** `a,` **`float`** `b)`, which returns `a + b`, using the `bind1st` and `ptr_fun`, we partially apply `add` to `2.0` as shown below:

```
1 std::bind1st( std::ptr_fun(add), 2.0 );
```

The result of `std::bind1st` is a functional object that behaves like a unary function which takes a single float as argument and returns the value of the argument incremented by `2.0`. For example, instantiating the following will return `12.2`:

```
1 std::bind1st( std::ptr_fun(add), 2.0 ) (10.2);
```

[3] **`class`** can be used as an equivalent for **`struct`**, except that all members are private by default

The `std::bind2nd` construct can be used similarly to bind the second argument of a binary function. The limitation of partial bind to binary function is overcome with the Boost library (Abrahams and Gurtovoy, 2004) that provides the `boost::bind` construct, which is a generalization of the standard functions `std::bind1st` and `std::bind2nd`. The `bind` construct can handle functions with more than two arguments, and its argument substitution mechanism is more general:

```
boost::bind(f, _1, _2, _3)(x, y, z);    // f(x,y,z);
```

C++ offers many features that can be tailored toward the implementation of functional idioms. One problem with the usage of templates to mimic the functional idioms is that of *code bloat* (Musser and Saini, 2001). An alternate approach to this implementation is the use of run-time type information (RTTI), but the problem is the run-time overhead involved.

Many library-based implementations supporting functional programming in C++ are available. The Boost library (Abrahams and Gurtovoy, 2004) provides enhancement to the function object adapters in C++, to support higher-order programming. FC++ (McNamara and Smaragdakis, 2000) is another library-based implementation that allows functional programming in C++ with the reusability benefits of higher-order polymorphic functions.

5. Generic MoCs Formulation in C++

Implementing the function-based formalism of MoCs in C++ brought up the following concerns: C++ allows passing of functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions at global or file scope, these function arguments cannot capture local environments.[4] Therefore, we had to model this type of function closure by enclosing the function inside an object such that the local environment or parts are captured as data members of the object. This is possible in C++ because objects in C++ are essentially higher-order records, that is, records that contain not only values but also functions. This sort of abstraction brings in type safety since it avoids the need for *type-casts* or *untyped pointers*.[5] This abstraction is facilitated through C++ classes or class templates. We illustrate how we model our process constructors and process combinators for the UMoC using this abstraction.

[4] the environment captured by a process

[5] an untyped pointer points to any data type

5.1 UMoC++ Framework

In this section, we briefly discuss the implementation of the UMoC in C++. Signals are defined as generic lists, which allows a signal to be of polymorphic type. To facilitate this in C++, we model signals as type `std::vector<T>`, where the placeholder `T` will be replaced by the actual type. Therefore, a signal is a vector of elements of type `T` as shown below:

```
/* Definition of a signal */
template <class T> class signalstruct
{
public:
  signalstruct();
   signalstruct();
  /* Define Accessory functions */
private:
  vector <T> signal;
};
```

We also model the different signal manipulators as function templates that take an input of type `signalstruct<T>` and the other input parameters and generate an output of type `T` or `signalstruct<T>`.

5.2 Process Constructors

We define a set of process constructors that have varied functionalities; some with internal state, some with a single input and output, and some with several inputs and several outputs. The parameters of a process constructor range from an initial state, a next-state function, output encoding function, to partitioning functions for different inputs and outputs. As a result of this varied type of parameters, we use the class template–based abstraction to built objects that hold the local environment for a specific constructor.

The Mealy-based process constructor has been implemented as shown in Listing 7.2. It is a function template that takes two arguments of type `MealyObj` and `signalstruct<SigType>` and returns a value of type `signalstruct<SigType>`. The first argument is an object that captures the environment of Mealy-based process. This class encapsulates an initial state and three member functions, which are as follows: (i) `pfn` determines the number of events handled during an evaluation cycle, from the current state of the process; (ii) `nfn`, when given the current state of the system and the input subsequence, calculates the next state of the system; and (iii) `ofn` is the output-encoding function that produces an output based on the current state and the input subsequence. Therefore, the `mealyU` function template, given an Mealy-based object and an input signal, produces an output and a transition of the system to the next state.

Listing 7.2 Mealy-based process constructor in C++

```
1  /* Mealy-based Process Constructor */
2  template <class MealyObj, class SigType>
3  signalstruct<SigType> mealyU (MealyObj obj, signalstruct ↵
       <SigType> isig)
4  {
5    signalstruct <SigType> osig;
6    if(isig.empty() == true)
7      return osig;
8    else {
9      MealyObj tobj;
10     osig   = obj.ofn(obj.w, take(isig, obj.pfn(obj.w)));
11     tobj.w = obj.nfn(obj.w, take(isig, obj.pfn(obj.w)));
12     osig   = append(osig, mealyU(tobj, drop(isig, ↵
           obj.pfn(obj.w))));
13     return osig;
14   }
15 }
```

The class template–based abstraction allows us to maintain a uniform structure[6] for the different process constructors, which is essential while defining the process composition operators as explained in the next subsection.

5.3 Process Combinators

Process combinators are operators that define the composition of different process constructors. We describe the implementation of the sequential and feedback combinator. Notice that the class template based–abstraction results in process constructors having similar structure, which facilitates the implementation of generic combinators. This abstraction allows the combinator to take any two processes and an input and define their composition, where the processes passed are objects that encapsulate their functionality.

The implementation of the sequential combinator is shown in Listing 7.3. It is a function template that takes four parameters. The first parameter of type `CombType` describes a type abstraction for the generic processes, whereas the second parameter of type `ProcObj1` and third parameter of type `ProcObj2` are process type objects and the fourth parameter of type `SigType` is an input signal. The first parameter is an encapsulation of two function templates `process1` and `process2`. The output of the instantiation of `process1`, with the object of type `ProcObj1` and the input signal, is given as input to `process2`

[6] w. r. t. number of arguments

Listing 7.3 Sequential composition

```
/* Sequential Composition of Processes */
template <class CombType, class ProcObj1, class ProcObj2, ↵
    class SigType>
SigType seqcomp (CombType comb, ProcObj1 obj1, ProcObj2 obj2, ↵
    SigType isig)
{
  SigType osig;
  osig = comb.process2 (obj2, comb.process1 (obj1, isig));
  return osig;
}
```

along with the object of type `ProcObj2` to return a signal that is the output of the sequential composition.

The implementation of the feedback combinator is shown in Mathaikutty et al. (2005). It is a function template that takes four parameters similar to the sequential combinator, except that the feedback composition is done on a single process and it takes two input signals. The second input signal is the fixed-point signal generated through the fixed-point operator. The fixed-point signal is computed on an event basis, until the fixed-pointing terminates. At each evaluation, the current fixed-point signal depends on the input signal and all the previously generated fixed-point values.

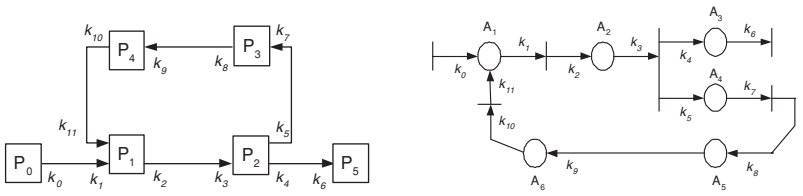
6. Example of Models in our Framework

We model an adaptive amplifier and a power state machine to demonstrate the expressiveness of the framework. The implementations for these models are provided in Mathaikutty et al. (2005). We illustrate the genericness of the UMoC++ framework by describing how to model Petri net, Synchronous data flow (SDF), and FSM.

6.1 Petri Net Style Modeling Using UMoC++

The Petri net style modeling is untimed by nature; therefore, we illustrate how analysis and design techniques can be applied to UMoC++ to allow Petri net modeling. A signature of a process is expressed as a pair of sets (I, O), where I contains the partitioning functions for the input and O contains the partitioning functions for the outputs. For a process network to be mapped to a Petri net, all the processes it is composed of should have constant signatures.

Consider the amplifier process composition shown in Mathaikutty et al. (2005). Let us assume that all processes have a constant signature as shown in Figure 7.2a, then this PN can be converted into a Petri net by representing each process by a transition and each signal by a place as shown in Figure



(a) The Amplifier process with constant partitions

(b) The Petri net for the Amplifier PN

Fig. 7.2 Mapping of the Amplifier PN to a Petri net

7.2b. This mapping is an abstraction of the original process because the data is abstracted into indistinguishable tokens. Consider Figure 7.2, the process signatures (I_i, O_i) for process P_i, $0 \leq i \leq 5$ are as follows:

$$I_0 = \{\} \quad I_1 = \{k_1, k_{11}\} \quad I_2 = \{k_3\} \quad I_3 = \{k_7\} \quad I_4 = \{k_9\} \quad I_5 = \{k_6\}$$
$$O_0 = \{k_0\} \quad O_1 = \{k_2\} \quad O_2 = \{k_4, k_5\} \quad O_3 = \{k_8\} \quad O_4 = \{k_{10}\} \quad O_5 = \{\}$$

with all k_j being constant natural numbers. Each transition in the Petri net represents an input or output of a process, and the weight of a transition is the corresponding constant partitioning function. The restriction on the process signatures can be relaxed to allow processes like state machines that have a rational match at each state[7] such that the ratio can be used to compute the weight of the transitions. Furthermore, these design techniques can be automated since there is a unique mapping from the UMoC++ framework to Petri net.

6.2 Synchronous Data Flow Style Modeling Using UMoC++

The SDF is also untimed and it is a specialization of our generic UMoC. In order to allow SDF style modeling, our UMoC is restricted to where all processes define only constant partitions for all their input and output signals. Therefore, all process signatures are constant. In Section 6.1, we had shown the mapping of PNs to Petri nets. A similar mapping can be used to derive the incidence matrix of the SDF graph, which in turn is used to compute schedules and maximum buffer sizes as explained in Jantsch (2003).

Modeling an FSM in UMoC requires using either the Mealy-based or the Moore-based process constructor. In order to illustrate the FSM capability of the UMoC, we model a fairly complex example of a power state machine as a Moore-based process shown in Mathaikutty et al. (2005).

[7] the ratio of I_q/O_q at state q is a constant

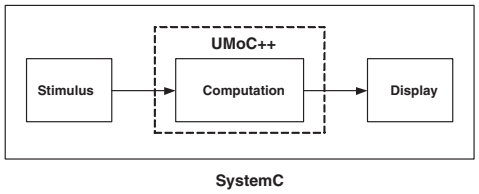


Fig. 7.3 An FIR Filter cosimulated using UMoC++ and SystemC

6.3 Cosimulation With SystemC

In SystemC, a process wraps the functionality that has to be scheduled to run through a discrete-event simulator; therefore, it results in slower simulation. The UMoC++ framework is independent of any simulator, and an immediate advantage is the ability to have an interoperable design approach with C/C++-based SLDLs such as SpecC and SystemC.

The relative ease to cosimulate with the UMoC++ framework is illustrated through an example of a finite impulse response (FIR) filter as shown in Figure 7.3, where the stimulus and display components are simulated using SystemC, and the computational component that convolutes the input and the FIR coefficient is modeled in UMoC++. Since the computation block has to communicate with the stimulus and display block, we built a wrapper for the computation block using SystemC. The computation block is modeled as a sequential composition of a zip-based process that groups the shifted input from the stimulus and the FIR coefficients and a map-based process that convolutes them. Furthermore, this interoperability positions UMoC++ as a framework that facilitates IP-based design integration.

7. Conclusion

We have presented a type-safe framework for modeling in the UMoC using C++, which supports a higher-order functional programming style. As a result, we get the advantages of FL in our imperative-based implementation and also remove the problems associated with efficiency and reusability. The framework is implemented entirely using C++ class templates. Limitation of this framework is that we provide minimal error-checking capability; therefore, the user is required to follow a strict discipline while modeling. Furthermore, during modeling the user is required to use only the generic process constructors and combinators and limit the usage of C++, since one immediate advantage of our framework is its extensibility. In this chapter, we primarily focus on how the foundation for the function-based semantics of generic MoC was built in an imperative language. Furthermore, we implemented the UMoC++ framework, which is a formulation of the UMoC. We illustrate through a set of

examples, the relative ease to model using our UMoC and its generic property that accounts for its expressibility and extensibility. We also briefly describe the mapping of models built in UMoC++ to many untimed variants, such as Petri net, SDF, and provide guidelines for an FSM style modeling. The other generic MoCs such as perfectly synchronous, clocked synchronous, and timed can be implemented using the formulation described in this chapter, which is the context of our future work.

References

Abrahams, D. and Gurtovoy, A. (2004) *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*. Addison-Wesley, Boston, MA.

Accellera (2006) *SystemVerilog*. Accellera. `http://www.systemverilog.org/`.

FERMAT (2006) *SML-Sys Framework*. FERMAT Group. `http://fermat.ece.vt.edu/SMLFramework`.

Jantsch, A. (2003) *Modeling Embedded Systems and SoC's Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers, San Francisco, CA.

Kuchën, H. and Striegnitz, J. (2002) Higher-order functions and partial applications for a C++ skeleton library. In: *Proceedings of the 2002 Joint ACMISCOPE Conference on Java Grande*, pp. 122–130.

Mathaikutty, D., Patel, H., and Shukla, S. (2004) A functional programming framework of heterogeneous model of computations for system design. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2004*, Lille, France. ECSI.

Mathaikutty, D., Patel, H., Shukla, S., and Jantsch, A. (2005) UMoC++: modeling environment for heterogeneous systems based on generic MoCs. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2005*. ECSI, Lausanne, Switzerland. pp. 291–302.

McNamara, B. and Smaragdakis, Y. (2000) Functional programming in C++. In: *Proceedings of the International Conference on Functional Programming (ICFP) 2000*.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997) *The Definition of Standard ML—Revised*. MIT Press, Cambridge, MA.

Musser, D. R. and Saini, A. (2001) *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, 2nd edn*. Addison-Wesley Professional Computing Series. Addison-Wesley, Boston, MA.

Okasaki, C. (1992) *Purely Functional Data Structures*. Cambridge University Press, Cambridge.

OSCI (2006) *Homepage of the SystemC Community*. The Open SystemC Initiative (OSCI). `http://www.systemc.org/`.

Patel, H. D. and Shukla, S. K. (2004) *SystemC Kernel Extensions for Heterogeneous System Modeling: —A Framework for Multi-MoC Modeling & Simulation*. Kluwer Academic Publishers, Boston, MA.

Sander, I. and Jantsch, A. (2004) System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32.

III

Analog, Mixed-Signal, and Heterogeneous System Design

Introduction

The design of analog and mixed-signal (AMS) systems has—unfortunately—never been done in a really systematic way. Until now, analog design is done rather bottom up and in an intuitive way. Therefore, the design of analog circuits has often been compared with "black magic". The lack of methodology was—and is—acceptable for small, stand-alone analog circuits that are functionally well separated from digital components.

Today's AMS systems no longer fulfil these conditions. Therefore, AMS designers face a number of challenges.

- The shrinking of analog circuits causes increasing process variations. This requires a more complete and more systematic verification, especially applying Monte Carlo Simulation, corner-case analysis, and regression tests. However, for reliable results many simulation runs (100–100,000) are needed. Considering the run time for numerical analog simulation, new methods like importance sampling, symbolic analysis, or even formal verification might become interesting complements.
- Analog circuits are more closely coupled and functionally linked with digital hardware or even software. Therefore, design and verification requires an overall system simulation. Despite attractive languages and simulators like very high speed integrated circuit (VHSIC) hardware description language (VHDL)-AMS or cosimulation environments, the mixed-domain and mixed-level modeling and simulation are still an issue and require especially appropriate modeling and verification methodologies.
- Many requirements (very low voltage, very low power, etc.) are hard to meet by the well-known analog circuit topologies. Available tools support the dimensioning and optimization of given topologies, but lack support for the more creative topology design. This task requires expert and application knowledge. Analog topology synthesis might solve the problem in the future. Today's designers must reuse the topologies once designed and adapt them to new requirements.

Using "black magic" from SPICE days for the design of AMS systems results in low design productivity and frequent redesigns. However, the application of new tools and languages can also be a challenge without the right knowledge, methodology, and design flow. The following part of the book contains some chapters that describe successful application of methodologies and tools. They give the reader valuable hints for solving the issues mentioned earlier.

Chapter 8 deals with the abstract modeling of micro mechanical components for system-level verification. Here, a behavioral model is created by reduced-order modeling methods and formulated in the language VHDL-AMS. This permits an overall system simulation with—despite the complexity—sufficient simulation speed.

Chapter 9 introduces a new kind of analysis that goes beyond simulation: semisymbolic simulation. Although not yet available in commercial simulators, the methods described seem to be an appropriate approach to deal with increasing process variations and to get better verification coverage.

Chapter 10 entitled "SystemC-WMS: mixed-signal simulation based on wave exchanges" introduces an extension to SystemC—originally intended for system-level analysis of hardware/software systems. This extension allows designers to include analog circuits into the system-level simulation, modeling the overall system in a single language—SystemC-WMS.

Besides the cosimulation based on specific languages, simulator coupling is an important issue. For simulator coupling, especially the interfaces between different languages and simulators require a lot of effort. Chapter 11, "Automatic generation of a coverification platform", gives an overview of an approach that supports the automatic generation of interfaces.

Finally, the application of Universal markup language (UML) for reuse of analog circuits is introduced in Chapter 12, "UML/XML-based approach to hierarchical AMS synthesis".

Christoph Grimm
University of Hannover
Hannover, Germany, February 2006

Chapter 8

Creating Virtual Prototypes of Complex MEMS Transducers Using Reduced-Order Modelling Methods and VHDL-AMS

Torsten Mähne[1,3], Kersten Kehr[1], Axel Franke[1], Jörg Hauer[1], and Bertram Schmidt[2]

[1]*Robert Bosch GmbH*
Department CR/ARY
P. O. Box 106050
D-70049 Stuttgart
Germany

[2]*Otto-von-Guericke-University Magdeburg*
Institute for Micro- and Sensor Systems (IMOS)
P. O. Box 4120
D-39016 Magdeburg
Germany

[3]*Ecole Polytechnique Fédérale de Lausanne (EPFL)*
Laboratoire de Systèmes Microélectroniques (LSM)
Bâtiment ELD, Station 11
CH-1015 Lausanne
Switzerland

Abstract In this chapter, the creation of "virtual prototypes" of complex micro-electro-mechanical transducers is presented. Creating these behavioural models can be partially automatised using a reduced-order modelling (ROM) method. It uses modal decomposition to represent the movement of flexible structures. Shape functions model the energy conservation and full coupling between the different physical domains. Both modal shapes and shape functions for strain energy and lumped capacitances of the structure can be derived in a highly automated way from a detailed 3D finite element (FE) model available from earlier design stages. Separating the generation of the ROM from the same FE model but for different operation directions circumvents the current limitations of the used ROM method. These submodels are integrated into a full model of the transducer. VHDL-AMS system is used to describe additional strong coupling effects

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 135–153.

between the different operation directions, which are not considered by the used ROM method itself. The application of this methodology on a commercially available yaw rate sensor as an example for a complex transducer demonstrates the practical suitability of this approach.

Keywords: micro-electro-mechanical systems (MEMS); surface micromachined (SMM) transducers; finite element method (FEM); model extraction; reduced-order modelling (ROM); modal decomposition; VHDL-AMS; geometry-, circuit-, and system-level simulation.

1. Introduction

Micro-electro-mechanical systems (MEMS) are characterised by a strong and non-linear interaction between various physical domains. The consideration of only one domain during the design process is therefore not sufficient (Mehner, 2000). The physics of microsystems can be described using partial differential equations. These can be solved numerically with boundary or finite elements methods (BEM, FEM). This approach leads to very detailed models, which are used to determine the mechanical properties of flexible microstructures and the electrostatic field distribution between their electrodes, i.e., to support the design process of the different MEMS components. However, these models are, in terms of memory consumption and computing time, too expensive to be used for the simulation of the entire microsystem. Thus feedback effects of driving and sensing circuits cannot be analysed at the detailed geometry level.

The whole system can only be described at higher levels of abstraction, like the circuit and system levels. On these levels, only the degrees of freedom (DOFs) at the interfaces (ports) of the different components are of interest. The derivation of simplified and verified behavioural models is therefore necessary. Their manual creation is time-consuming, error-prone, and often implies strong simplifications (only first and second DOFs). One better solution is the automatised generation of these models by extracting the necessary informations from detailed FE models that are already available from earlier design steps. This can be done using reduced-order modelling (ROM) methods that were in scope of several research efforts in the field of MEMS over the past few years (Bechtold et al., 2003; Chen et al., 2004; Gabbay et al., 2000; Mehner et al., 2000; Reitz et al., 2004; Rudnyi and Korvink, 2002, Rudnyi et al., 2004). These methods focus on various fields of application with different interdomain coupling effects.

One of these ROM methods, developed by Gabbay et al. was evaluated regarding its applicability to complex MEMS that use electrostatic fields to excite the mechanical structure and to detect its movements. It was successfully used to create a fully coupled behavioural model of a commercially available micromechanical yaw rate sensor. A new approach is presented to circumvent the limitation of the chosen ROM method to structures moving primarily in

one dominant direction. Separate models for the different operating directions are generated from the same FE model and afterwards coupled using VHDL-AMS system to manually model the missing effects. The modelling flow is presented as well as a methodology to organise the models for efficient future maintenance and extension.

2. Theory of the Reduced-Order Modelling Method

The chosen ROM method (Bennini et al., 2001b) uses a weighted sum of n mode shapes (modal amplitudes q_i, eigenvectors $\varphi_i(x, y, z)$) of the mechanical structure to represent its deflection u:

$$u(t, x, y, z) \approx u_{\text{eq}} + \sum_{i=1}^{n} q_i(t) \cdot \varphi_i(x, y, z) \tag{8.1}$$

where u_{eq} is the initial displacement caused by prestress in equilibrium state. Especially for MEMS, a few eigenmodes are usually enough to accurately describe the dynamic response of the structure (Bennini et al., 2001a). The strain energy W_{mech} that is stored within the structure due to deflection or prestress is expressed as a function of the modal amplitudes q_i. Geometrical non-linearities and stress-stiffening are considered by calculating the modal stiffness k_{ij} from the second derivatives of the strain energy with respect to the modal amplitudes:

$$k_{ij} = \frac{\partial^2 W_{\text{mech}}}{\partial q_i \, \partial q_j}. \tag{8.2}$$

The modal masses m_i and modal damping constants d_i are calculated from the eigenfrequencies ω_i of the modes i and the entries of the modal stiffness matrix k_{ij}:

$$m_i = \frac{k_{ii}}{\omega_i^2} \tag{8.3}$$

$$\mathrm{d}_i = 2\xi_i \omega_i \cdot m_i \tag{8.4}$$

where ξ_i is the modal damping ratio of mode i. The modal damping ratios represent the fluidic damping of the structure and can be obtained from analytical calculations (squeeze or slide film damping), numerical fluid dynamic simulations, or measurements. The deflection of the mechanical structure changes the capacitances between the electrodes in a non-linear manner. The capacitance C_{op} between the electrodes o and p is calculated as a function of the modal amplitudes and therefore provides the coupling between the mechanical and electrical quantities. The displacement current I_o through the electrode o can

be calculated from the stored charge:

$$I_o = \frac{\mathrm{d}Q_o}{\mathrm{d}t} = \sum_p \frac{\mathrm{d}\left[C_{op}(q_1, \ldots, q_n)(V_o - V_p)\right]}{\mathrm{d}t} \tag{8.5}$$

where V_o and V_p are the voltages at the electrodes. The governing equation describing the whole electrostatically actuated micromechanical structure in terms of modal coordinates is:

$$\begin{aligned} F_{\mathrm{M},i} &= m_i\ddot{q}_i + \mathrm{d}_i\dot{q}_i + \frac{\partial W_{\mathrm{mech}}(q_1, \ldots, q_n)}{\partial q_i} \\ &\quad -\frac{1}{2}\sum_r \frac{\partial C_{op}^{(r)}(q_1, \ldots, q_n)}{\partial q_i}(V_o - V_p)^2 + \sum_j \varphi_{ij}\lambda_j \end{aligned} \tag{8.6}$$

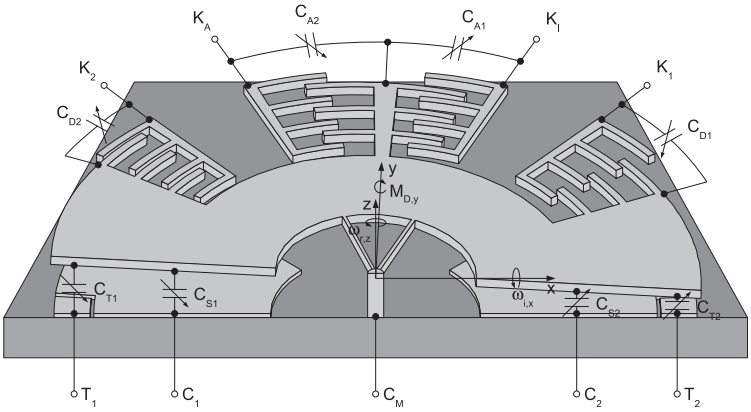
where $F_{\mathrm{M},i}$ is the modal force and r is the number of capacitances involved between the multiple electrodes. The λ_j are the reaction forces to the external forces $F_{\mathrm{N},j} = -\lambda_j$ at the master nodes j of the FE mesh that remain accessible in the behavioural model.

The ROM consists of Equations 8.1, 8.5, and 8.6, which fully describe the static and dynamic non-linear behaviour of the flexible structure and its non-linear coupling to the electrostatic domain. All missing parameters of the ROM can be derived from a detailed, fully coupled FE model of the MEMS component in a highly automated manner. The eigenvectors φ_i and eigenfrequencies ω_i of the considered modes i are taken from the modal analysis of the mechanical structure. The shape function of the strain energy $W_{\mathrm{mech}}(q_i)$ as well as the functions of the capacitances $C_{op}(q_i)$ are expressed in a polynomial form. They are fitted to a set of sample points of strain energy and capacitances extracted from a series of static analyses of the FE model, in which the structure is deflected to various linear combinations of its mode shapes. The ROM-Tool available in ANSYS/Multiphysics since Release 7 is one implementation of this method (ANSYS, 2002).

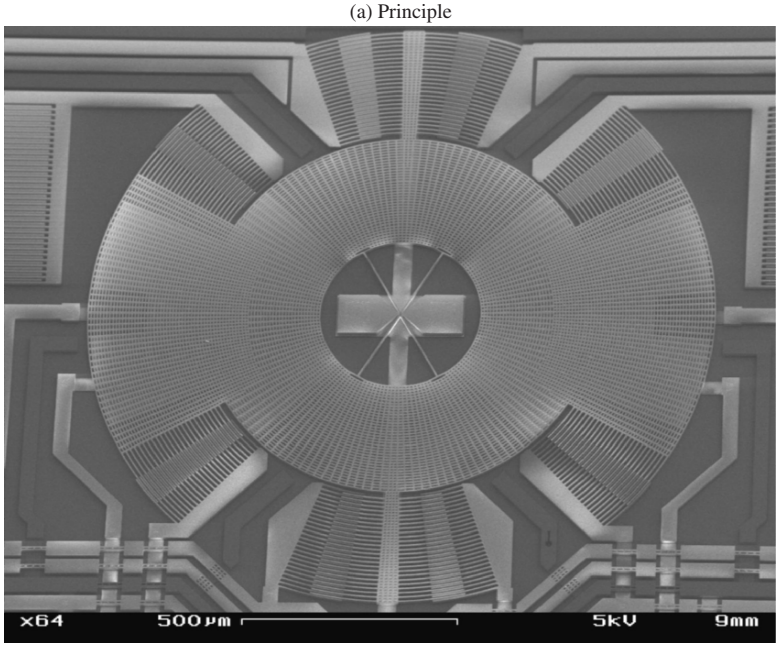
3. Micromechanical Yaw Rate Sensor

The yaw rate sensor (Figure 8.1) developed by Robert Bosch GmbH is manufactured in a surface micromachining (SMM) process (Funk, 1998). The mechanical part of the sensor consists of a flat polysilicon structure. This rotor is fixed to the centre by an X-shaped spring and thus movable around all three axes of the coordinate system. Comb-drive structures, which are placed in pairs at its perimeter, excite and detect the in-plane oscillation of the rotor around the vertical z-axis. If the whole chip with the oscillating rotor is rotated around the x-axis with the angular rate $\omega_{\mathrm{i},x}$, the law of the conservation of the angular momentum causes a torque $M_{\mathrm{D},y}$ around the y-axis:

$$M_{\mathrm{D},y} = \left(J_z + J_y - J_x\right)\omega_{\mathrm{i},x}\omega_{\mathrm{r},z} \tag{8.7}$$



(a) Principle



(b) SEM photography

Fig. 8.1 Yaw rate sensor manufactured by Robert Bosch GmbH

where J_x, J_y, and J_z are the moments of inertia of the rotor around the axes of the coordinate system and $\omega_{\mathrm{r},z}$ is the current angular speed of the rotor. The rotor starts a tilting oscillation around the y-axis with an amplitude proportional to the angular rate $\omega_{\mathrm{i},x}$. This out-of-plane movement of the rotor is detected electrostatically using the electrodes below the structure.

4. Preparation of the FE Models for the ROM Method

To apply the described ROM method, detailed FE models of the yaw rate sensor are needed, which model the structural as well as the electrostatic domains. To that end, the structural model available from earlier design steps of the device has to be extended to describe also the capacitances between the electrodes. To circumvent the limitation of the ROM method to systems moving primarily in one direction, two separate ROMs have to be generated for the yaw rate sensor: one representing the in-plane movement of the rotor and the other the out-of-plane movement. The two models will be coupled on the circuit level. To limit the model size and conserve simulation time during the generation pass, the FE models should contain only the details necessary for the particular ROM. Hence, two separate coupled FE models of the sensor are created from the same underlying structural model: one modelling the comb-drive capacitances to excite and detect the in-plane movement of the rotor and the other modelling the capacitances of the rotor to the electrodes below the structure used to detect its out-of-plane movement. This approach also overcomes some implementation-related limitations of the ANSYS ROM-Tool regarding the complexity of the models, especially the number of considered modes ($\leq$9), conductors ($\leq$5), and master nodes ($\leq$10), that can be transformed into ROMs.

To model the capacitances of the comb-drive structures and the capacitances between the movable structure and the subjacent electrodes, 1D electromechanical transducer elements of the type `TRANS126` were used (ANSYS, 2002). The nodes of these lumped elements have structural DOFs (displacement and force) as well as electrical DOFs (voltage and current) to fully describe the interaction between the structural and electrostatic domain in their proximity. The capacitance-displacement functions of the transducer elements are derived from an analytical approach using the parallel-plate capacitor assumption. The calculated characteristic curves can be corrected to account for the stray field using the results obtained from electrostatic field simulations with detailed cutaway FE models of one comb finger and a cross-shaped section of the moving SMM structure. In order to describe the total capacitance between two electrodes many transducer elements have to be connected in parallel, each describing one section of the space between the electrodes. In case of the comb-drive structures and the capacitances below the rotor, this results in one element at the end of each comb finger and at each cross section, respectively. Figure 8.2 illustrates all the steps necessary to prepare the coupled FE models for the ROM generation pass.

5. Generation of the Reduced-Order Behavioural Models

The prepared coupled FE models of the yaw rate sensor can now be simplified to the reduced-order behavioural models using the ANSYS ROM-Tool. This

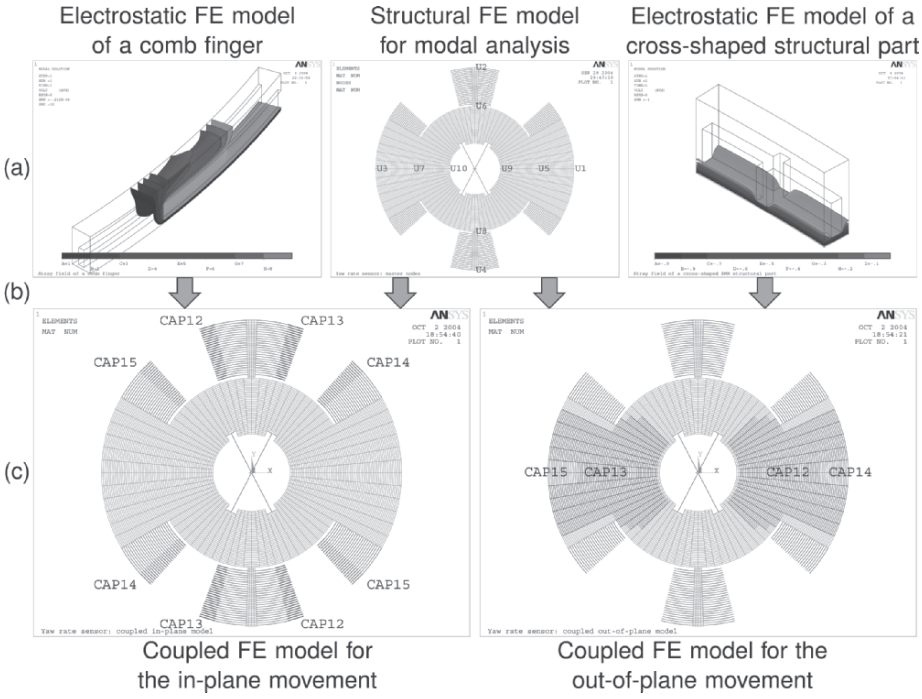
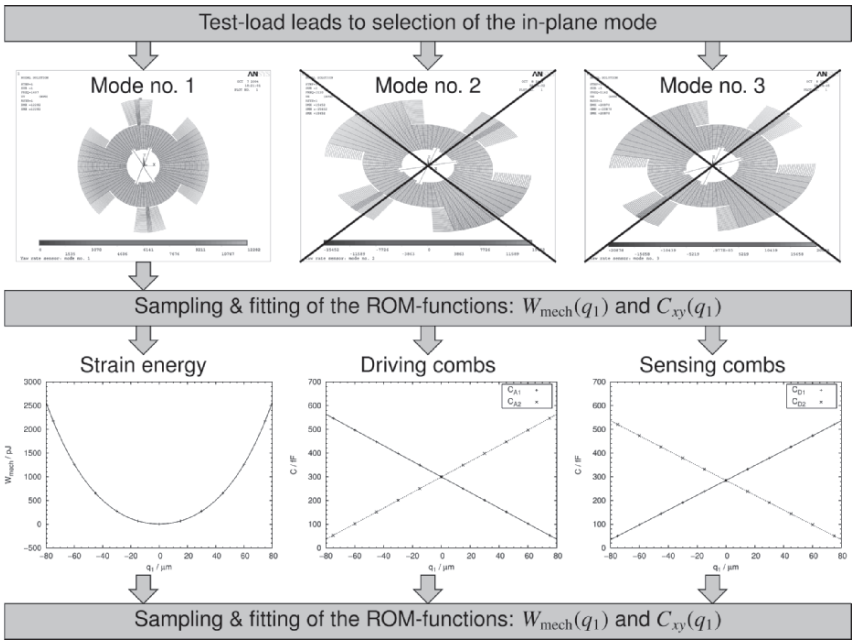


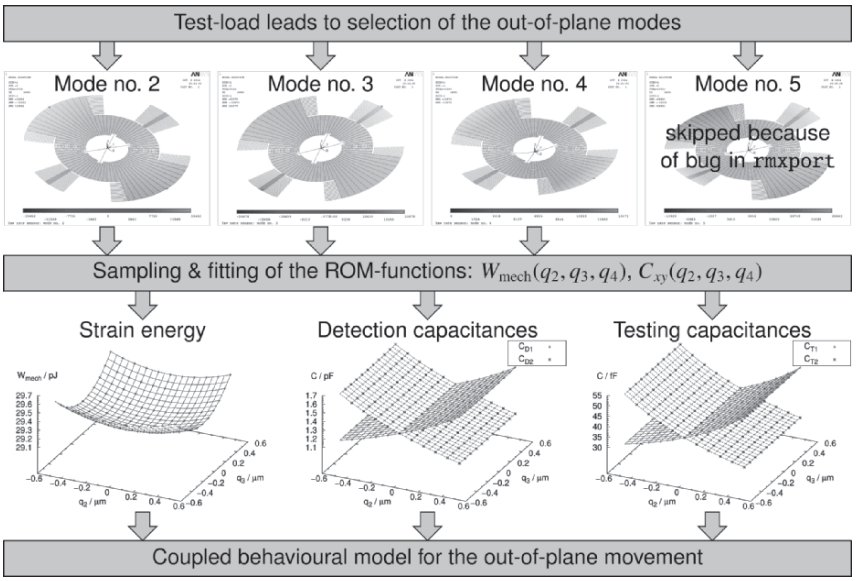
Fig. 8.2 Steps to prepare the coupled FE models of the yaw rate sensor for the generation of its in-plane and out-of-plane ROMs: (a) single-domain FE models, (b) preparation of the coupled FE models, and (c) electromechanically coupled FE models

is done during the generation pass, which consists of a series of steps that are described in detail in ANSYS (2002) and Mähne (2004). Figure 8.3 illustrates the most important steps for the generation of the in-plane and out-of-plane ROMs of the sensor.

First the tool is initialised by establishing the main properties of the ROM: name of the FE model and its dimensionality (2D or 3D), working direction of the structure, capacitances, and master nodes that should be retained in the ROM. Then a static test load is applied to the FE model to bring it in a typical deflection state that will be used to select the right modes for the ROM. The displacement of the structure through the test load is extracted from the FE model at the nodes that are selected to represent the neutral plane of the structure and stored in a file for later use. With a modal analysis of the structure, its first nine eigenfrequencies and mode shapes (represented by the eigenvectors of the nodes in the neutral plane and of the master nodes) are extracted. After these preparatory analyses, the modes for the ROM are selected automatically by calculating the contribution factor of each mode shape to resemble as close as possible the deflection state of the structure through the test load (Figure 8.3). The first eigenmode (oscillation around the z-axis) is identified as the only one contributing to the in-plane movement of the rotor and therefore selected



(a) In-plane model



(b) Out-of-plane model

Fig. 8.3 Steps to generate the reduced-order models of the yaw rate sensor

for the in-plane ROM. All extracted higher eigenmodes contribute to the out-of-plane movement of the rotor, but the calculated contribution factors show that the lower eigenmodes are dominating. These are the modes no. 2 and 3 (tilting oscillation around the *y*- and *x*-axes) as well as the modes no. 4 and 5 (butterfly-shaped oscillation of higher order around the *y*- and *x*-axes). Due to a bug in the VHDL-AMS export function of ANSYS Releases ≤ 8.1, only three modes can currently be considered in the ROMs. For this reason only the modes no. 2, 3, and 4 are selected for the out-of-plane ROM. This does not influence the modelling of the sensor effect (conservation of the angular momentum) itself, but influences the modelling of higher-order interfering effects like prestress and direct structural coupling between the different operation directions. The most time-consuming step is the following sampling pass where the strain energy and capacitances of the FE model are extracted for certain compositions of the scaled mode shapes. The sample routine had to be reimplemented to allow the extraction of the capacitances from the `TRANS126` elements. To this set of sample points the polynomials for the strain energy and capacitances are fitted. Type and order of the polynomials can be chosen individually for each shape function. The fitting step concludes the generation of the ROMs. They can be used afterwards within ANSYS (`ROM144` element) or exported to an equivalent VHDL-AMS behavioural model (Schlegel et al., 2005).

The complete generation pass is automatised using APDL scripts so that the generation of the in-plane and out-of-plane ROMs can be run overnight in batch-mode. This is necessary since each mode was captured using 11 samples giving just 11 static analyses for the in-plane ROM but $11^3 = 1331$ static analyses for the out-of-plane ROM. Since each static analysis takes roughly 1 min on recent PCs (P4 Xeon, 2.8 GHz, 4 GB RAM, Linux) both ROMs are generated within 23 h of CPU time.

6. Integration of the Reduced-Order Behavioural Models

The exported VHDL-AMS description of the ROMs are stored in a number of packages and entities that follow a fixed naming scheme (Mehner, 2004). Therefore, they have to be separated into different design libraries. Each ROM defines a package `initial` that contains its characteristic constants, namely, modal masses, modal damping ratios, and eigenvectors of the master nodes. The polynomials for the strain energy $W_{\mathrm{mech}}(q_i, q_j, q_k)$ and the capacitances $C_{\mathrm{op}}(q_i, q_j, q_k)$ are defined in separate packages called `s_ams_`ijk and `ca`op`_ams_`ijk, respectively, each defining the type of the particular polynomial, a flag if it should be inverted; the order of the polynomials with respect to the modal amplitudes q_i, q_j, and q_k; scaling factors for the functions to

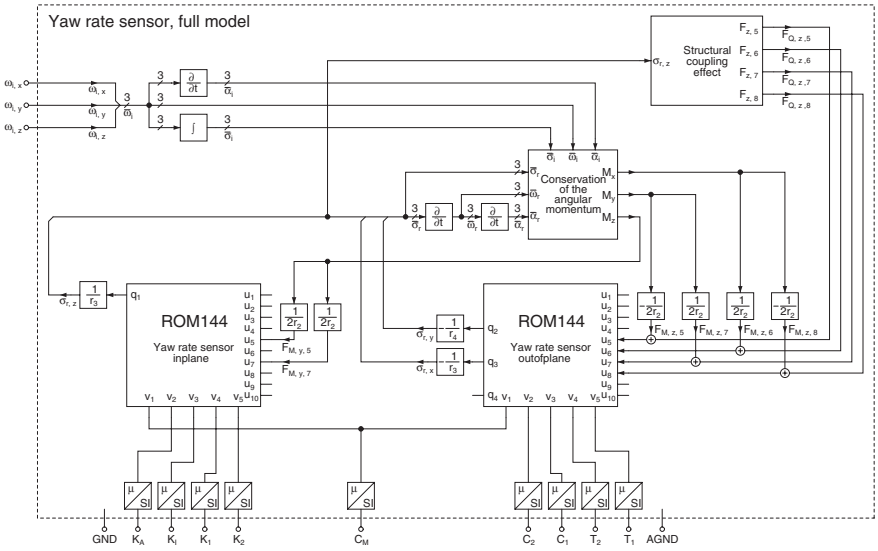


Fig. 8.4 Structure of the yaw rate sensor full model with the in-plane and out-of-plane ROMs

overcome numerical problems; the number of polynomial coefficients; and the coefficients themselves.

The entity `transducer` (the `ROM144` functional blocks in Figure 8.4) describes the interface of the ROM. Each DOF of the `ROM144` element is mapped to one of the across or through quantities of the terminals of the entity. At the modal terminals the modal amplitude q_i and modal force $F_{\mathrm{M},i}$ are available for the chosen modes. The master node terminals provide the displacement u_i and the inserted forces $F_{\mathrm{N},i}$ at these nodes. At the electrical terminals the voltages V_i and currents I_i are available for the electrodes of the system.

The architecture `behav` of this entity implements the complete behavioural model of the ROM. It declares all across and through quantities for the terminals of the entity. It also defines a function `spoly_calc()`, which calculates the strain energy and capacitances as well as their first derivatives with respect to the modal amplitudes q_i, q_j, and q_k using the information of the polynomials defined in the packages `s_ams_`ijk and `ca`op`_ams_`ijk, respectively. The ordinary differential equations (Equations 8.1, 8.5, and 8.6) that describe the ROM are directly included in the architecture body as simultaneous statements using the `'dot` attribute to describe the derivatives with respect to the time.

An interesting detail of the ROMs is the use of an own system of units called µMKSV based on µm, kg, s, and V. Its use is recommended for ANSYS to overcome numerical problems in MEMS FE models (ANSYS, 2002). Since the ROM method does not change the system of units of a model it

has to be declared for the VHDL-AMS models too. The use of the IEEE standard packages for electrical (`ieee.electrical_systems`) and mechanical systems (`ieee.mechanical_systems`) is not possible since they are based on the SI system of units that is used for most other models on the circuit and system level. Hence, a new package `µMKSV` has been created that declares all important mechanical and electrical quantities as subtypes of `real` using their own tolerance groups. The attributes `unit` and `symbol` are defined for each subtype naming their unit in a long and short form. These attributes document the declarations and are used for presentational purposes such as naming the axes in the signal plots of the simulator. The declared subtypes and natures can be named the same as their counterparts in the IEEE packages, allowing easy switching between the system of units, but only if the global name handling is implemented correctly within the VHDL-AMS simulation environment. Otherwise, additional prefixes have to be used to prevent name clashes. Converter entities were implemented as an interface between the µMKSV and SI systems of units for the electrical (voltage, current) and mechanical quantities (displacement, force). They convert the across and through quantities between the terminals for the different system of units so that Kirchhoff's law remains valid.

To describe the complete yaw rate sensor, the two generated ROMs of the structure have to be coupled to model the conservation of the angular momentum and a direct structural coupling between the operating directions of the sensor (Figure 8.4). The interface to this new entity representing the complete sensor consists of the electrical terminals that correspond to the pads on the sensor chip and three quantity input ports for the chip yaw rates $\omega_{\mathrm{i},x}$, $\omega_{\mathrm{i},y}$, and $\omega_{\mathrm{i},z}$ around the x-, y-, and z-axes, respectively. The coupling is done in the architecture `behavioural` using the VHDL-AMS support for combining the structural and behavioural descriptions in a single architecture. One instance of the in-plane and one instance of the out-of-plane ROMs are created and their electrical terminals are connected through the system of units converter entities to the corresponding external electrical terminals. Since the external terminals of the sensor model use the declarations from `ieee.electrical_systems`, it is fully compatible to other models of electrical components that use the SI system of units.

To model the conservation of the angular momentum, the state of motion of the rotor with respect to the sensor chip as well as the one of the sensor chip itself with respect to an inertial coordinate system have to be known. The state of motion of the rotor is calculated from the first three modal amplitudes. These are directly proportional to the deflection angels $\sigma_{\mathrm{r},z}$, $\sigma_{\mathrm{r},y}$, and $\sigma_{\mathrm{r},x}$ of the rotor around the z-, y-, and x-axes, respectively. The first and second derivatives of the deflection angles with respect to time give the current angular velocities $\omega_{\mathrm{r},z}$, $\omega_{\mathrm{r},y}$, $\omega_{\mathrm{r},x}$ and angular accelerations $\alpha_{\mathrm{r},z}$, $\alpha_{\mathrm{r},y}$, $\alpha_{\mathrm{r},x}$. The state of motion of

the sensor chip itself is calculated from the yaw rate input signals by integrating and deriving them with respect to time to get the current rotation angles $\sigma_{\mathrm{i},x}$, $\sigma_{\mathrm{i},y}$, $\sigma_{\mathrm{i},z}$ and angular accelerations $\alpha_{\mathrm{i},x}$, $\alpha_{\mathrm{i},y}$, $\alpha_{\mathrm{i},z}$. With the known state of motion of the rotor and the sensor chip, the torques M_x, M_y, and M_z can be calculated to conserve the angular momentum (Funk, 1998; Mähne, 2004):

$$\begin{aligned}
M_x &= -\left[J_x\left(\alpha_{\mathrm{i},x} + \omega_{\mathrm{i},y}\omega_{\mathrm{r},z} - \omega_{\mathrm{i},z}\omega_{\mathrm{r},y}\right) + \left(\omega_{\mathrm{i},y} + \omega_{\mathrm{r},y}\right)\left(\omega_{\mathrm{i},z} + \omega_{\mathrm{r},z}\right)\left(J_z - J_y\right)\right] \\
M_y &= -\left[J_y\left(\alpha_{\mathrm{i},y} + \omega_{\mathrm{i},z}\omega_{\mathrm{r},x} - \omega_{\mathrm{i},x}\omega_{\mathrm{r},z}\right) + \left(\omega_{\mathrm{i},z} + \omega_{\mathrm{r},z}\right)\left(\omega_{\mathrm{i},x} + \omega_{\mathrm{r},x}\right)\left(J_x - J_z\right)\right] \\
M_z &= -\left[J_z\left(\alpha_{\mathrm{i},z} + \omega_{\mathrm{i},x}\omega_{\mathrm{r},y} - \omega_{\mathrm{i},y}\omega_{\mathrm{r},x}\right) + \left(\omega_{\mathrm{i},x} + \omega_{\mathrm{r},x}\right)\left(\omega_{\mathrm{i},y} + \omega_{\mathrm{r},y}\right)\left(J_y - J_x\right)\right]
\end{aligned} \tag{8.8}$$

The torques are applied to the ROMs as force pairs at facing master nodes. The known deflection angles of the rotor can also be used to model the direct structural coupling between the in-plane and out-of-plane motion due to the non-rectangular sections of the beam springs supporting the rotor. To resemble the out-of-plane deflection of the rotor, forces $F_{\mathrm{Q},z,i}(\sigma_{\mathrm{r},z})$ are applied at the master nodes of the out-of-plane ROM. The relationships between forces and the in-plane displacement angle $\sigma_{\mathrm{r},z}$ are derived from the results of a series of static analyses of the structural FE model, to which polynomials are fitted. The implementation details of this effect are encapsulated in their own entity so that they can be changed easily to adapt the model, e.g., to changes in the manufacturing process. The created sensor model could easily be extended to include further effects such as cross talk between the electrical signal paths due to parasitic capacitances and resistors.

7. Simulation of the Complete Sensor System

The created full model of the yaw rate sensor can be integrated in a test bench of the complete sensor system for verification (Figure 8.5). The test bench connects simple structural models of the drive amplifier, detection circuit, and amplitude gain controller (AGC) to the sensor model. Although the models of the electrical components implement just their ideal behaviour, they could easily be extended to be more realistic by adding a new architecture.

The created models of the sensor systems set high demands to the used VHDL-AMS simulator regarding its IEEE standard conformity and the quality of its solvers. This is a general problem when simulating MEMS and other multidomain system that are common, e.g., in the automotive sector (Haase, 2003; Schwarz, 2002). Because of the coupling of different domains very different time constants appear in the system creating stiff differential equations, which couple quantities of very different orders of magnitude. An additional problem is the need for integration of models using different systems of units as discussed in Section 6 creating the necessity of support for tolerance

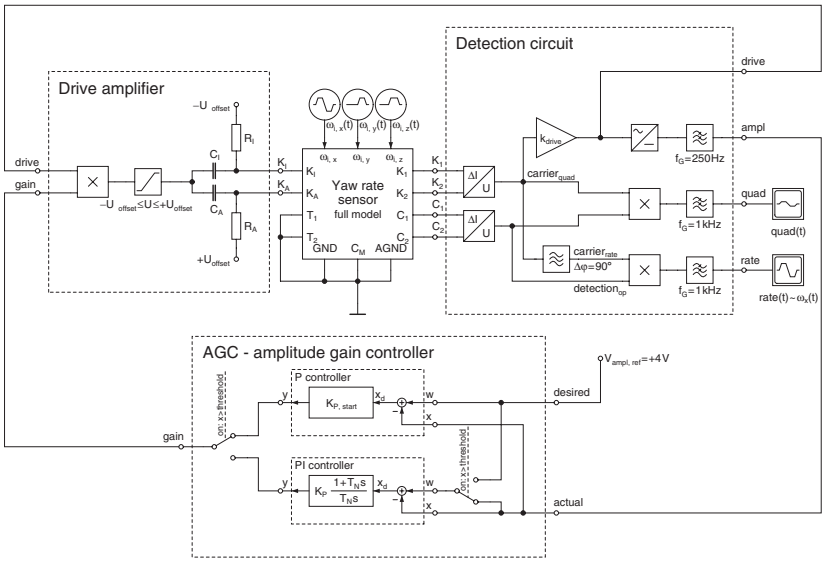


Fig. 8.5 Structure of the test bench for the yaw rate sensor full model

groups. For non-linear systems with discontinuities in their description, like the switch discussed later, the support for the **break** statement to reinitialise the operating point is very important. Only the simulator SMASH of Dolphin Integration met all these demands and its version 5.2.1 was used for the analysis of the created model (Dolphin, 2003). Operating point and small signal analysis of the ROMs and the complete yaw rate sensor model, which were exposed to certain static and small signal loads, showed that all important mechanical and electrical characteristic quantities (stiffness, masses, moments of inertia, eigenfrequencies, quality factors, capacitances) are very close (relative error ≤6%) to the values of the coupled FE model and correspond well with the specifications of the real sensor element.

The drive amplifier uses a variable *gain* to amplify the detected *drive* signal and couples it capacitively on the drive combs. This signal excites the in-plane motion of the rotor. The movement is detected by measuring the displacement current at the detection combs, which is converted to a voltage and amplified to give the *drive* signal that is fed back to the input of the drive amplifier. The closed feedback loop leads to a resonant self-excitation of the in-plane movement that is stabilised using the AGC. It consists of two controllers: a P controller with a high proportional gain used to speed up the start-up phase and a PI controller with a lower proportional gain but with a reset time. The control is switched when the amplitude reaches a threshold so that the in-plane amplitude can stabilise on the desired value. The switch models need support

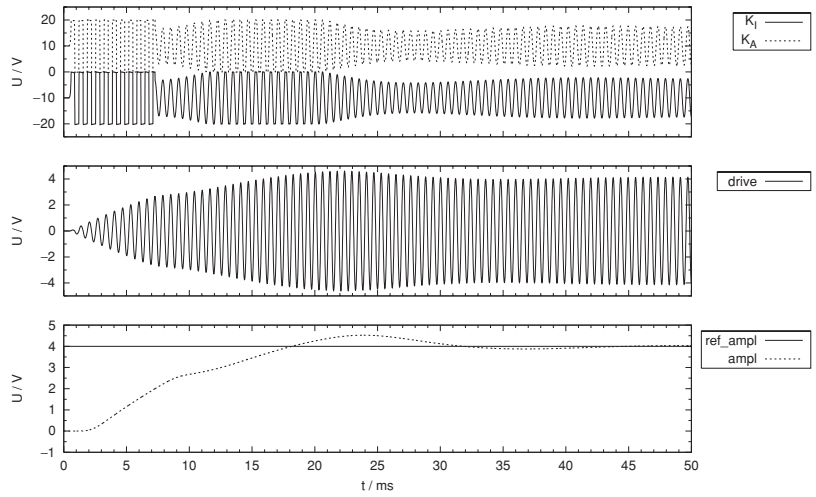
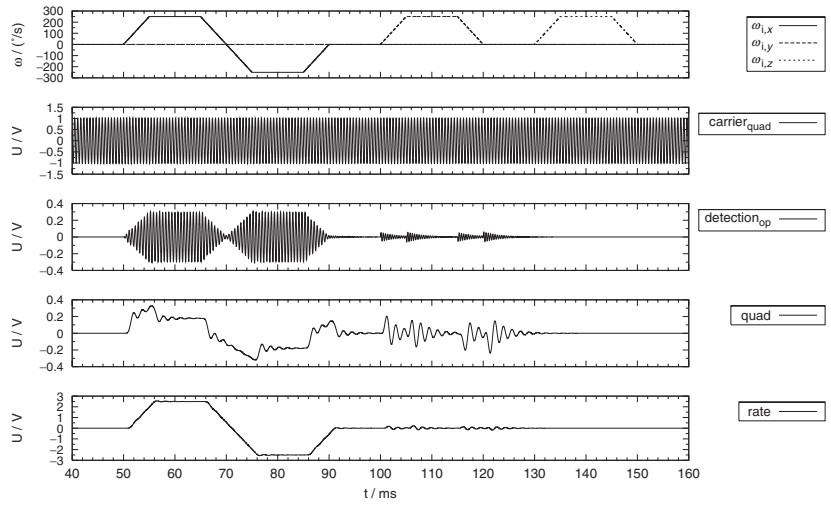


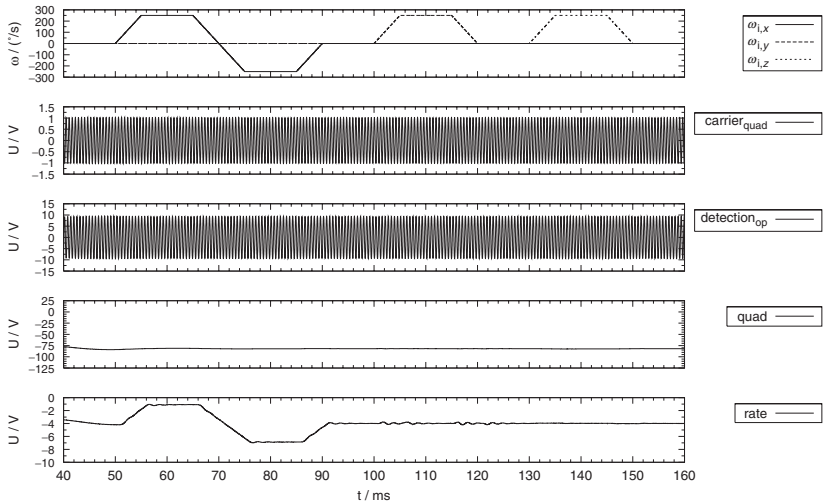
Fig. 8.6 Simulated self-excitation of the yaw rate sensor

for VHDL-AMS **break** statement within the used simulator to allow proper initialisation of the new operating point of the system. Figure 8.6 shows the results of the transient analysis of the self-excitation of the in-plane movement of the rotor. The first graph shows the feedback signals at the driving electrodes, the second one shows the detected in-plane signal, and the third one shows the amplitude and reference signals at the AGC. The in-plane oscillation stabilises on the desired amplitude within 50 ms. After this start-up phase the sensor is ready for detecting the yaw rate $\omega_{\mathrm{i},x}$. The results show also a frequency shift of the in-plane movement from the expected 1.487 kHz (first eigenfrequency of the rotor) to 1.680 kHz due to stress-stiffening of the X-shaped spring at large in-plane amplitudes and demonstrates the successful consideration of non-linear mechanical effects by the chosen ROM method.

The yaw rate $\omega_{\mathrm{i},x}$ at the input of the sensor causes an out-of-plane movement of the oscillating rotor, which is measured using the displacement currents at the subjacent detection electrodes. This out-of-plane signal contains the desired *rate* signal as well as the *quad* signal that is caused by the structural coupling between the in-plane and out-of-plane movement. The signals are 90° out of phase so that they can be clearly separated using synchronous amplitude demodulation with the *drive* signal as the demodulation carrier. It also allows a sign-sensitive detection of $\omega_{\mathrm{i},x}$. The results of the transient analysis of this *rate* detection is shown in Figure 8.7 for two cases—one neglecting the structural coupling between the in-plane and out-of-plane movement and one considering the structural coupling. The first graph shows the yaw rate test signals applied to the sensor, the second one shows the *drive* signal used as



(a) Structural coupling effect neglected



(b) Structural coupling effect considered

Fig. 8.7 Simulated rate detection of the yaw rate sensor

demodulation carrier of the out-of-plane signal in the third graph, the fourth graph shows the *quad* signal, which corresponds to the amount of structural coupling between the in-plane and out-of-plane movement, and the fifth graph shows the demodulated *rate* signal. Without structural coupling the *rate* signal is the envelope of the detected out-of-plane signal. It also shows the low cross sensitivity of the sensor against yaw rates other than $\omega_{i,x}$. The small detected *quad* signal results from a cross demodulation error. The simulation results for the second case show that the out-of-plane movement caused by the yaw rate is much smaller than the much stronger movement caused by the direct structural coupling between in-plane and out-of-plane movement. However, the *rate* signal can still be extracted because of the fixed phase relationship between the two movements. The remaining offset error could be easily suppressed.

The results (Figures 8.6 and 8.7) agree well with older results published in Lorenz (1999) and Reitz et al. (2004). The transient simulation of the whole sensor system including self-excitation and rate detection over a period of 120 ms with a maximum integration step width of 1 µs using SMASH took approximately 45 min on a recent PC (P4, 2.4 GHz, 512 MB RAM, Windows 2000). The newly created behavioural sensor model can be used to evaluate different circuit concepts (Funk, 1998; Rocznik, 2004) by creating new architectures and changing the configuration of the entities for the driving, sensing, and control circuits or by creating entire new testbenches that integrate the sensor model.

8. Conclusions

In this chapter, a new approach for creating accurate fully coupled behavioural models (virtual prototypes) of complex MEMS was presented. A commercially available ROM method was utilised to automatically extract the verified ROMs from available FE models of the component and VHDL-AMS to model missing effects. The whole modelling process was outlined using a micromechanical yaw rate sensor as an example. It was shown how structural FE models available from earlier design steps can be extended to also model the coupling to the electrostatic domain and how the ROMs are generated from these prepared FE models using the ANSYS ROM-Tool. While the ROM method itself is still under research, the implementation used here is already useful even though it has limitations. A careful partitioning of the problem and modelling of the missing coupling effects by hand can circumvent most of these limitations. This was demonstrated by generating separate ROMs for the different moving directions of the yaw rate sensor and their coupling on the circuit level. The modelling of the missing coupling effects can be done in a very natural way using VHDL-AMS powerful language constructs for behavioural and structural description. The realised partitioning of the full yaw rate sensor model offers an easy way to add/change the coupling effects

between the different ROMs. The created full model of the sensor was included in a testbench for the complete sensor system adding the circuits for driving, sensing, and controlling of the movement of the micromechanical element. Different analyses showed the successful modelling of all important mechanical and electrical properties of the sensor, the self-excitation of the in-plane movement, the yaw rate detection, and the low cross sensitivity of the sensor. It was shown that VHDL-AMS is a good choice to model non-linear, discontinuous, and multidomain systems even though their simulation imposes high demands on the used simulator, which were met only by SMASH for the presented project. In the future, MEMS will get even more complex so that further research on ROM methods is required to improve the automation of the model extraction and extend the coverage of considered effects within the ROM. More effort is also needed on the tool side to improve the implementation and integration of the different program systems that are used in the MEMS design process.

Acknowledgments

The work presented in this chapter was carried out within a diploma thesis project at the department CR/ARY of Robert Bosch GmbH in close collaboration with the IMOS of Otto-von-Guericke-University Magdeburg.

References

ANSYS (2002) *ANSYS 7.1—Coupled-Field Analysis Guide*. ANSYS, Inc.

Bechtold, Tamara, Rudnyi, Evgenii B., and Korvink, Jan G. (2003) Automatic generation of compact electro-thermal models for semiconductor devices. In: *IEICE Transactions on Electronics*, E86-C: 459–465.

Bennini, Fouad, Mehner, Jan, and Dötzel, Wolfram (2001a) Computational methods for reduced order modeling of coupled domain simulations. In: *11th International Conference on Solid-State Sensors and Actuators (Transducers 01)*. IEEE, Munich, Germany, pp. 260–263.

Bennini, Fouad, Mehner, Jan, and Dötzel, Wolfram (2001b) A modal decomposition technique for fast harmonic and transient simulations of MEMS. In: *International MEMS Workshop (IMEMS)*, volume 9, Singapore, pp. 477–484.

Chen, Jinghong, Kang, Sung-Mo (Steve), Zou, Jun, Liu, Chang, and Schutt-Ainé, José E. (2004) Reduced-order modeling of weakly nonlinear MEMS devices with taylor-series expansion and arnoldi approach. *Journal of Microelectromechanical Systems*, 13(3):441–451.

Dolphin (2003) *VHDL-AMS in SMASH Release 5.2*. Dolphin Integration, 39, avenue du Granier, B. P. 65 ZIRST, 38242 Meylan, France.

Funk, Karsten (1998) *Entwurf, Herstellung und Charakterisierung eines mikromechanischen Sensors zur Messung von Drehgeschwindigkeiten*. Dissertation, Technische Universität München, München.

Gabbay, Lynn D., Mehner, Jan E., and Senturia, Stephen D. (2000) Computer-aided generation of nonlinear reduced-order dynamic macromodels—i: Non-stress-stiffened case. *Journal of Microelectromechanical Systems*, 9(2):262–269.

Haase, Joachim (2003) Anforderungen an VHDL-AMS-Simulatoren (Entwurf vom 1. Juli 2003). Technical report, Fraunhofer-Institut für Integrierte Schaltungen, Außenstelle EAS Dresden, Zeunerstraße 38, 01069 Dresden.

Lorenz, Gunar (1999) *Netzwerksimulation mikromechanischer Systeme*. Number D46 (Diss. Universität Bremen) in Berichte aus der Mikromechanik. Shaker Verlag, Aachen.

Mähne, Torsten (2004) Ordnungsreduktionsverfahren zur automatischen Generierung von Systemmodellen bei mikroelektromechanischen Systemen. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Fakultät für Elektrotechnik und Informationstechnik, Postfach 4120, D-39016 Magdeburg.

Mehner, Jan (2000) *Entwurf in der Mikrosystemtechnik*, volume 9 of *Dresdner Beiträge zur Sensorik*. Dresden University Press, Dresden, München. Zugl.: Chemnitz, Techn. Univ., Habil., 1999.

Mehner, Jan (2004) *External System Simulation Based on VHDL-AMS*. CADFEM GmbH.

Mehner, Jan E., Gabbay, Lynn D., and Senturia, Stephen D. (2000) Computer-aided generation of nonlinear reduced-order dynamic macromodels—ii: Stress-stiffened case. *Journal of Microelectromechanical Systems*, 9(2):270–274.

Reitz, Sven, Döring, Christian, Bastian, Jens, Schneider, Peter, Schwarz, Peter, and Neul, Reinhard (2004) System level modeling of the relevant physical effects of inertial sensors using order reduction methods. In: *DTIP of MEMS & MOEMS*, Montreux, Switzerland.

Rocznik, Marko (2004) *Optimierung des Entwurfs mikroelektromechanischer Drehratensensorsysteme*. Dissertation, Fakultät Elektrotechnik und Informationstechnik der Technischen Universität Ilmenau, Ilmenau.

Rudnyi, Evgenii B. and Korvink, Jan G. (2002) Review: automatic model reduction for transient simulation of MEMS-based devices. In: *Sensors Update*, volume 11, pages 3–33. WILEY-VCH Verlagsgesellschaft, Weinheim.

Rudnyi, E. B., Lienemann, J., Greiner, A., and Korvink, J. G. (2004) mor4ansys: generating compact models directly from ANSYS models. In: *Technical Proceedings of the 2004 Nanotechnology Conference and Trade Show, Nanotech 2004*, volume 2. Boston, MA, pp. 279–282.

Schlegel, Michael, Bennini, Fouad, Mehner, Jan E., Herrmann, Göran, Müller, Dietmar, and Dötzel, Wolfram (2005) Analyzing and simulation of MEMS in VHDL-AMS based on reduced-order FE models. *IEEE Sensors Journal*, 5(5):1019–1026.

Schwarz, Peter (2002) Modellierung und Simulation heterogener technischer Systeme. Technical report, Fraunhofer Institut für Integrierte Schaltungen Erlangen, Außenstelle Entwurfsautomatisierung Dresden, Zeunerstraße 38, D-01069 Dresden.

Chapter 9

Modeling Uncertainty in Nonlinear Analog Systems with Affine Arithmetic

Wilhelm Heupke[1], Christoph Grimm[2], and Klaus Waldschmidt[1]

[1] *Department of Computer Engineering*
University of Frankfurt
Germany

[2] *Institute of Microelectronic Systems*
University of Hannover
Germany

Abstract This chapter describes a semisymbolic method for the analysis of mixed signal systems. Aimed at control and signal-processing applications, it delivers a superset of all reachable values. The method that relies on affine arithmetic is precise for linear systems, but in the case of nonlinear systems approximations are needed. As a new term is added for each approximation, the number of approximation terms increases during simulation and therefore slows down the simulation. This leads to a quadratic time complexity in the number of time steps. A method to avoid this and an example implementation based on SystemC analog and mixed signal (AMS) are presented. Efficiency and time complexity of the improved semisymbolic simulation are analyzed and discussed.

Keywords: affine arithmetic; intervals; SystemC-AMS; simulation; uncertainty; tolerance.

1. Introduction

Today's automotive, telecommunication, and ambient intelligence applications consist of sensors, actuators, analog and digital circuits, and a large portion of software. At the system level, designers usually specify and model such applications by continuous-time block diagrams with directed signal flow. For the verification and analysis of such systems, most notably a transient simulation is used: input stimuli are specified and the simulator computes the output signals. The transient simulation allows designers to get important insight into

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 155–169.

the behavior of the modeled system and provides a basic functional verification. However, within the design process of many of the above-mentioned systems, much time is spent for analyzing the impact of uncertainties:

- Static variations of operating conditions (e.g., production tolerances)
- Dynamic variations of operating conditions (e.g., temperature drift)
- Quantization and rounding errors in digital signal processing (DSP) and analog/digital conversion
- Physical effects in analog circuits (e.g., noise)

One big problem is that some deviations are compensated and do not have a large impact on some output of interest, whereas another deviation of same magnitude will be amplified and thus violates the specification.

The established analog or mixed-signal simulators at the electrical level provide different methods that help designers to evaluate the impact of deviations: noise analysis, sensitivity analysis, worst-case analysis, Monte Carlo analysis, AC analysis, and sometimes combinations thereof. These analyses are either based on the fact that analog circuits can be linearized around a working point (AC analysis, noise analysis), require monotonicity (sensitivity analysis), or use a huge number of simulation runs to find corner cases (worst-case simulation) and to compute statistical properties at the outputs (Monte Carlo analysis).

Although these analyses are very useful, they have several drawbacks. Methods based on linearization are usually restricted to analog circuits and are not applicable to mixed-signal systems or even DSP software. In order to overcome these problems designers have to provide discrete models and additional models that are used for AC analysis. Furthermore, time domain simulations are used in combination with fast fourier transform (FFT) methods to get information about the spectral distribution of noise, for example. Unfortunately, transient simulations and Monte Carlo methods do not provide information about the contribution of single sources of uncertainty to the total uncertainty at, for example, outputs in a direct and easy way; usually the interpretation is rather difficult.

The above-mentioned classical analyses are aimed toward the electrical level and are based on linearization and linear equation solvers. The method proposed in this chapter is intended for a system-level simulation with a discrete time static data flow model of computation, which is implied by the use of SystemC analog and mixed signal (AMS). One should be aware that there is no automatic way to use the electrical-level models at the system level or vice versa, yet.

Compared with purely numerical simulation, the symbolic or formal techniques provide designers with more information, e.g., about the origin of a

deviation (Henzinger, 1996; Zhang and Mackworth, 1996; Hartong et al., 2002). Unfortunately, symbolic or formal techniques are far away from being applicable to the design of complex and heterogeneous systems (Stauner et al., 1997). In this situation, semisymbolic techniques are very attractive if they combine advantages of symbolic techniques with the general applicability of simulation. A promising approach is the use of affine arithmetic (Andrade et al., 1994) for semisymbolic analysis (Fang et al., 2003; Lemke et al., 2002) or even a semisymbolic simulation (Heupke et al., 2003). A direct and easy interpretation is of particular interest for the case of design automation at system level.

Fang and Rutenbar are doing a static analysis of rounding errors in DSP algorithms with affine arithmetic (Fang et al., 2003). In Heupke et al. (2003) and Grimm et al. (2004a), we use affine arithmetic for semisymbolic transient simulations of complex signal processing systems. The simulation result is a numerical output together with a symbolic, affine approximation of the contributions of different (symbolic) sources of uncertainty. An important advantage of the proposed method is the safe inclusion of all reachable values by the affine expression, there by delivering reliable results. On the other hand, the increasing number of terms and the resulting overapproximation caused by each nonlinear operation are the disadvantages.

In this chapter, we introduce a method for semisymbolic simulation with affine arithmetic that efficiently handles these approximation terms. Section 2 gives a brief introduction to affine arithmetic and semisymbolic simulation with affine arithmetic as described in Heupke et al. (2003). Section 3 introduces a method to handle the overapproximation terms in semisymbolic simulation based on affine arithmetic. This enables affine arithmetic to reach the same asymptotic time complexity as that of conventional numerical simulation. Section 4 demonstrates the applicability of the method by a simple example.

2. Semisymbolic Simulation with Affine Arithmetic

2.1 Basic Concepts of Affine Arithmetic

Affine arithmetic (Andrade et al., 1994), introduced by Comba et al., is a kind of improved interval arithmetic and therefore allows us to compute with uncertain values. In each affine expression, the influence of independent sources of uncertainty i to a variable $\hat{x}$ with the central value x_0 is represented by a symbolic sum of terms $x_i\epsilon_i$. Noise symbols ϵ_i represent arbitrary, but for one simulation run fixed values from the interval $[-1, 1]$. The partial deviations x_i then scale these intervals. The ϵ_i is a symbolic representation and a certain value is never assigned to them.

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\epsilon_i, \qquad \epsilon_i \in [-1, 1]$$

Table 9.1 Affine expressions and their interval counterparts

Affine arithmetic		*Interval arithmetic*	
Affine form	*Diameter*	*Interval*	*Diameter*
$\hat{x} = 17.3 + 2.5\epsilon_1$	5.0	$X = [14.8, 19.8]$	5.0
$\hat{y} = 15.4 + 2.5\epsilon_1$	5.0	$Y = [12.9, 17.9]$	5.0
$\hat{z} = 15.4 + 2.5\epsilon_2$	5.0	$Z = [12.9, 17.9]$	5.0
$\hat{x} - \hat{y} = 1.9 + 0.0\epsilon_1$	0.0	$X - Y = [-3.1, 6.9]$	10.0
$\hat{x} - \hat{z} = 1.9 + 2.5\epsilon_1 - 2.5\epsilon_2$	10.0	$X - Z = [-3.1, 6.9]$	10.0

Basic mathematical operations are defined by

$$\hat{x} \pm \hat{y} := (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\epsilon_\mathrm{i}$$

and

$$c\hat{x} := cx_0 + \sum_{i=1}^{n} cx_i\epsilon_\mathrm{i}.$$

The operation $\mathrm{rad}(\hat{x})$ is the radius of the affine expression $\hat{x}$.

$$\mathrm{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i|$$

The results of linear operations give accurate limits and have no over-approximation (no unnecessary expansion of the error interval).

2.2 Interval Arithmetic versus Affine Arithmetic

The subtraction of two affine expressions, which include the same noise symbols ϵ_m, may reduce the partial deviation of the result, in contrast to the same values with different noise symbols. This allows us to model error cancellation, for example in feedback loops. In Table 9.1 the variables with a hat denote affine arithmetic variables whereas the ones written with a capital letter are corresponding interval variables. The diameter is obviously twice the radius for affine forms. In the case of intervals, the diameter is the difference between supremum and infimum of the interval.

Table 9.1 shows the difference between affine arithmetic and interval arithmetic in the case of different or same source of uncertainty. The variables x and y share an uncertainty caused by the same source of uncertainty and therefore both have a term ϵ_1. For demonstration purposes also the influence of this

uncertainty is of same magnitude and direction/sign. In contrast to that the variable z has a term ϵ_2 that shows that the uncertainty of z has a different source of uncertainty, although both have the same magnitude in example given in Table 9.1. The effect shows up in the subtraction of $x - y$. Interval arithmetic increases the interval diameter instead of bringing it to zero, whereas affine arithmetic keeps the correlation and delivers the precise result. This is because the information contained in interval arithmetic is too limited, as the range of values is not the only important information that is needed to describe the influence of uncertainty.

This effect of interval arithmetic may be tolerated sometimes, but a simulation of a control loop, where a too pessimistic result is fed back in each time step, results in a diameter that is increasing with simulated time and depending on the system will increase exponentially in the worst case. This will deliver with interval arithmetic that the result is $[-\infty, +\infty]$ within a small number of simulation time steps (Heupke et al., 2003). For sure this is a safe inclusion, but would be useless pessimistic.

The concept described in this chapter can be extended to dynamic uncertainties and therefore to analyze effects like colored noise as described in Grimm et al. (2004b).

An important aspect is the guarantee that, after each operation, the result is a superset of all reachable values. This is a problem especially for nonlinear operations. For nonlinear operations, different methods for overapproximation are defined in Andrade et al. (1994). These methods add a new noise symbol ϵ_{m+1} that describes the over approximation. For example, multiplication of two affine expressions is defined by:

$$\hat{x} \cdot \hat{y} := x_0 \cdot y_0 + \sum_{i=1}^{m} (x_0 \cdot y_i + x_i \cdot y_0)\epsilon_\mathrm{i} + \mathrm{rad}(\hat{x}) \cdot \mathrm{rad}(\hat{y}) \cdot \epsilon_{m+1}.$$

In general, the error introduced by some nonlinear operation is overapproximated by a new noise symbol ϵ_{m+1}.

All nonlinear operations introduce new noise symbols, and therefore some systems may present a problem because of the permanently increasing number of terms. But some systems include strategies to reduce the influence of deviations (e.g., feedback). Caused by these strategies, the influence of these noise symbols converges to zero and for a stable system they are absolutely summable. Section 3 describes how this property delivers a solution for the problem of the increasing number of terms.

2.3 SystemC-AMS-Based Implementation

For the implementation we chose SystemC-AMS (Vachoux et al., 2004), but the concept mentioned later can be implemented in every language that

supports operator overloading, e.g., very high speed integrated circuit (VHSIC) hardware description language (VHDL)-AMS.

SystemC-AMS is an extension of the class library SystemC, aimed at supporting the modeling of mixed-signal (analog and digital) systems. It provides means to simulate analog, mixed-signal, and signal processing systems as a block diagram. In contrast, SystemC allows us to immediately reuse the code portion for these blocks, which have to be implemented in software later on. Additionally, the code of the models that will be implemented in digital hardware can be automatically synthesized to create, for example, an application specific integrated circuit (ASIC) or field programmable gate array (FPGA) implementation. Only for the blocks, which model analog behavior, there is no automatic way for implementation. These blocks are specified by transfer functions or static nonlinear functions implemented in C++. Static data flow is used as the model of computation.

The implementation of the affine arithmetic is based on the `libaffa` library (Gay, 2003), which defines linear and nonlinear operations on affine arithmetic variables in a class called `AAF` (affine arithmetic forms). It allows us to model computations with uncertainties in general.

Using the `AAF` class with SystemC-AMS is very simple. In SystemC-AMS, as in SystemC, signals are instantiated with a template parameter `T` that specifies their value type. For example by `sca_sdf_signal<double> my_signal`, a signal with a value range of a variable with double precision is instantiated in SystemC-AMS. Of course, one can as well specify the template parameter `AAF` instead of `double`. This small change is all that is needed with SystemC-AMS to turn the numerical simulation into a semisymbolic simulation based on affine arithmetic. Instead of using operators defined for `double` values, the compiler will use the operators defined in the `AAF` class, which overload the standard operators. The results of the simulation are affine expressions that semisymbolically represent possible deviations.

For example, one can write the following code:

```
1 AAF a(2.0), b(3.0), c(2.0), y;
2
3 // constructor which adds a noise symbol
4 // x_i with partial deviation 0.1:
5 AAF uncertainty(Interval(-0.1, +0.1));
6 a = a + uncertainty;
7 y = (a + b) * (c + uncertainty);
8 cout << "y␣=␣" << y << endl;
```

This simple program produces the following output:

```
y = 10 + (e1*0.7) + (e2*0.01)
```

So after the uncertainty is introduced one can use a variable of type `AAF` like any other variable.

The advantage of semisymbolic simulation compared with a pure numerical simulation becomes obvious if the uncertainties at the output of the simulated system exceed some specified range. In this case, the symbolic representation provides designers with the contribution of all sources of uncertainty to the deviation at the output. It also models the effects that are created by the combination of uncertainties. This together allows the designer to identify the sources of uncertainty where improvements are most fruitful. As a long-term perspective, one day a mixed-signal synthesis system can be directed this way where further optimization is needed.

3. Efficient Handling of Additional Terms in Feedback Control Systems

Each nonlinear operation approximation creates an additional term, as can be seen in the code example. These approximations are a problem for the affine arithmetic, as potentially a high number of very small and thereby insignificant terms in the symbolic expression is created.

This problem shows up especially if the system that is modeled contains a loop and has at least one component which creates an approximation in the path of this loop (e.g., by multiplication of two affine expressions).

Then any kind of memory (e.g., some modeled energy storage) in a block within the modeled system will contain most of the approximation terms that are created in each simulation cycle of this loop. If there is a constant number of approximations, this means that in each simulation cycle the number of terms increases by this constant number.

To cope with this, we introduce a method that "cleans up" the affine arithmetic expression variables. It somewhat resembles the garbage collection concept used to free unneeded memory of variables, which is used in some programming languages, e.g., Java.

If the number of noise symbols in the affine expressions increases above a certain level, the `simplification()` method is called. For all variables in the system, all terms smaller than a cut level l, set by the user, are summed up separately by the ones with a positive and the ones with a negative sign to two special noise terms.

Deleting the terms with an absolute value below this cut level could potentially lead to inaccurate results in the case of a high number of simulation time steps and certain functional blocks, e.g., integrators, because in this example they may grow to a big one over time. Therefore it is better to sum them. This way it delivers a safe inclusion, but it means that the correlation of the individual terms is lost. But it does not exhibit the same problem like interval arithmetic does, as described earlier, because the correlation of this sum is still valid for all `AAF` variables in the future time steps and the terms with

different signs are kept separate. Furthermore, these uncertainties are usually far smaller than the nominal values; and if l again is far smaller than the other uncertainties, any kind of over-approximation would not create a problem. So the influence of approximations decreases below the level l after several time steps.

Please note that if the simplification method would be called too often, the unneeded overapproximation could potentially show up significantly, and in the above-mentioned example the concept of feedback that makes these terms converge to 0 would not work. On the other hand, if called not often enough, the computation time will increase significantly. Our experience is that the choice of the time point, when to call the simplification method, was not very critical for the example system.

The method resembles the typical strategy of leaving away smaller terms. But with affine arithmetic we do not have to really leave the smaller terms away, instead we can handle their sum as a new uncertainty. So not only the modeled uncertainties of the real system but obviously also the uncertainties caused by the modeling process, like these simplifications, are analyzed.

3.1 Implementation of the Simplification Method

In the present implementation the simplification method is invoked every thousandth simulation cycle, but later on it might be automatically invoked by some heuristic. For example the change in the highest index of the noise terms since the last simplification could be used as a criterion, when to call this method. The cut level l is set to a constant small fraction of the smallest explicitly introduced uncertainty by the user.

The change in an affine expression can be seen by the following example of a simplification with $l = 1.0 \cdot 10^{-4}$. First a variable was printed immediately before the simplification:

```
28.9796 + (e1*2.9925) + (e5*0.000856951)
+ (e6*1.14971e-006) + (e7*1.11085e-006)
+ (e8*-1.34821e-007) + (e9*1.07968e-006)
+ (e10*-1.12145e-007)
```

After the simplification the printed variable changed to

```
28.9796 + (e1*2.9925) + (e5*0.000856951)
 + (e34*3.34024e-006) + (e35*-2.46966e-007)
```

Usually this happens with far more terms, but for demonstration purposes it would be difficult to show. In this case ϵ_{34} sums up the positive insignificant terms and ϵ_{35} sums up the negative insignificant terms.

By handling a list with pointers to all affine variables in the system, it is possible to access all `AAF` variables. This list is added as a static member of the `AAF` class so that all `AAF` variables share it.

The `AAF` class saves the affine expression in one variable for the central value x_0 and two pointers to dynamically allocated arrays called `coefficients` and `indices`. In a first run across all `AAF` variables and across all coefficients in these variables, the significant terms are collected based on the cut level l. A term x_i of an affine variable $\hat{x}$ is significant if it fulfills the condition:

$$|x_i| > l.$$

The second run goes again across all variables. For each of the variables it is determined how many significant terms are contained, based on the result of the first run. Then two new arrays for the coefficients and the indices are allocated and the significant terms are copied to the new arrays. After that the memory of the old arrays is freed.

3.2 Comparison of Efficiency

The following text analyses the effort to handle one variable. So the total effort also scales with the number of variables for all similar simulation methods.

Figure 9.1 shows a system we implemented as a test in order to validate the behavior by transient simulations with affine expressions as data type. It contains two elements that can be troublesome:

- The first is feedback. Another range arithmetic, the interval arithmetic (Moore, 1966), will not deliver a meaningful result for the simulation of systems with feedback, whereas affine arithmetic works fine in this respect (Heupke et al., 2003).

- The second is the emphasized nonlinear block in the system, which is interesting, as it creates additional approximation terms in each iteration through the loop. This results in a high number of terms that slows down the simulation more and more if nothing is done about that.

Let us assume such a system with a loop, n be the number of total simulation time steps, and c be a constant that describes the maximum number of nonlinear operations along the path of the loop. Remember that these nonlinear operations add terms. Further let k be the number of explicitly introduced uncertainties.

With conventional simulation based on the static data flow model of computation and with variables of type, e.g., **`double`** the space complexity is $O(1)$ and the simulation time is $O(n)$.

In contrast to that in the naive implementation, the maximum memory needed for each affine variable is

$$cn + k \subseteq O(n)$$

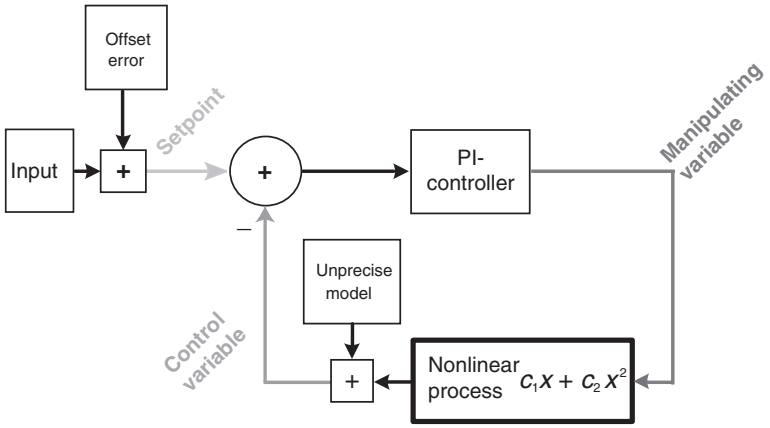


Fig. 9.1 Simulated system with nonlinear block

because in each of the n steps c uncertainties are added, caused by overapproximations, and a maximum of k has been added intentionally at the elaboration phase. This means that the space complexity is $O(n)$.

Even worse is the resulting time complexity. This is because in each simulation time step each term of an affine variable needs to be handled, e.g., an arithmetic operation has to be performed for it by the central processing unit (CPU):

$$\sum_{i=1}^{n}(ci+k) = \frac{cn(n+1)}{2} + kn = \frac{c}{2}\cdot n^2 + \frac{c+k}{2}\cdot n \subseteq O(n^2)$$

For every stable system system theory require that every bounded input delivers a bounded output. Obviously, every technically meaningful system to be implemented is stable. Furthermore, a discrete system is stable if and only if the impulse response is absolutely summable:

$$\sum_{i=-\infty}^{\infty} |h(i)| < \infty$$

This implies an important aspect: the impulse response of the opened control loop converges to zero. So every introduced overapproximation term will converge to zero with the number of iterations through the control loop in the given example. This means that we can apply a trick that copes with the terms caused by the overapproximation. From time to time we sum up all approximation terms that got extremely small (smaller than l) by a simplification method, thereby keeping the safe inclusion but reducing the number of terms. On the other hand, this means, if the number of terms cannot be reduced, we get a strong indication that the system might be unstable.

Table 9.2 Measured computation time

Number of time steps	*Computation time without simplification (s)*	*Computation time with simplification (s)*
1,000	1	1
2,000	4	2
4,000	16	5
8,000	61	10
16,000	244	20
32,000	999	40
64,000	4,083	79
128,000	–	159
256,000	–	319
512,000	–	640
1,024,000	–	1,275

This simplification method, if called every m simulation time steps, is a substantial step forward regarding efficiency; because in the m time steps between two simplification operations, a maximum of c terms adds in each time step. To this adds the number of k explicitly introduced terms. As c, m, and k are all constants, we get asymptotically the same space complexity like pure numerical simulation:

$$cm + k \subseteq O(1)$$

The time complexity of the simulation with the simplification method needs $cm + k$ computations in one simulation time step in the worst-case of the time step before the next simplification method call. This happens n times in the worst-case. To this adds the effort of the simplification method, called n/m times. The simplification method itself needs in the first and the second pass to touch every term. This gives a total time complexity of $O(n)$; the same complexity numerical simulation has

$$(cm + k)n + \frac{2n}{m}(cm + k) = \left(cm + k + \frac{2}{m}(cm + k)\right) \cdot n \subseteq O(n)$$

4. Experimental Results

The system shown in Figure 9.1 was implemented in SystemC-AMS and the `AAF` class. With this setup transient simulation runs were performed.

Table 9.2 shows the time needed for the simulation with and without the simplification method used. The time interval that was simulated was the same for all values in the table and was scaled to deliver time results that are easy to

interpret. Only the step width in time was changed for each row. The simplification method was called every thousandth time step, respectively, never in the case of no simplification.

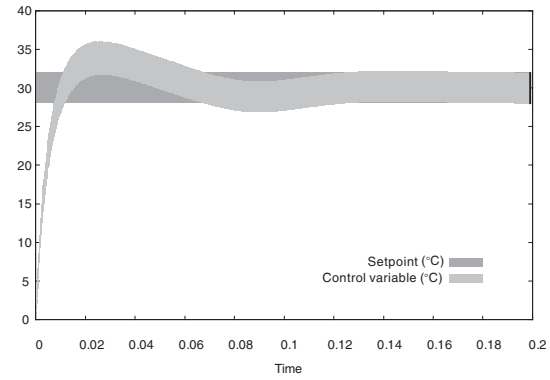
The table shows the clear advantage of the simplification method, as the computation time increases linearly with the number of simulated time steps if the simplification method is used. It is very clear to see a quadratic increase of the needed computation time for the simulation without using the simplification method that shows up as a fourfold increase of the required computation time for a twofold increase in the number of time steps. So it gets obvious that affine arithmetic would be much harder to use without this simplification method for long simulation runs in the presence of feedback and nonlinearity.

For a visual representation, we convert affine expressions to intervals, by use of the `rad` operator. These intervals can be plotted as shown in Figure 9.2 as a range. In the case of an uncertainty that is substantially smaller than the central value, two separate traces with different scalings are plotted. For both types of plots we use the program gnuplot, as usual waveform viewers do not support interval type signals. Figure 9.2a shows the step response of a feedback loop that contains a nonlinear control path, which is shown in Figure 9.1. Figure 9.2b shows the step response close to the stability border and Figure 9.2c the same system, but beyond the stability border. It is interesting to note the typical chaotic effects of nonlinear systems near the stability border that show up very clearly in the uncertainty, and which are not linear with the central value in Figure 9.2b.

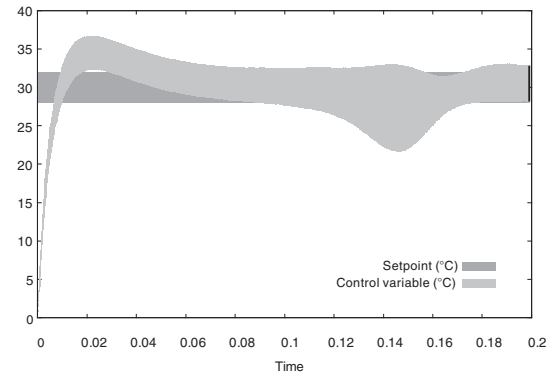
5. Conclusion

Without the described method, semisymbolic simulation with affine arithmetic has quadratic time complexity. On the other hand, with the presented method, simulation with affine arithmetic has linear time complexity even in the presence of nonlinearities and feedback. This means that affine arithmetic is feasible for simulation even with a large number of time steps in nonlinear feedback systems.

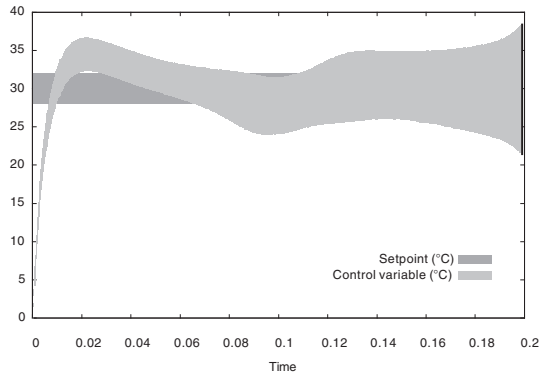
Compared with the analyses in “analog” simulators, the described method is applicable to DSP methods and to discrete-time approximations of analog circuits. This allows designers an analysis of heterogeneous systems that include large fractions of DSP software. The symbolic representation of the contributions to the deviations at the outputs can be interpreted easily and it delivers a safe inclusion, an important aspect for the design of security critical systems, which could create otherwise dangerous situations if their deviation is too large.



(a) System within the stable area



(b) System near the stability border



(c) System beyond the stability border

Fig. 9.2 Step responses of a feedback loop containing a nonlinear block

The improved efficiency of semisymbolic simulation is a basic foundation for semisymbolic simulation of more complex systems. For example, in Grabowski et al. (2006), semisymbolic simulation is extended toward analog circuit simulation. Without the method described in this chapter, this would not have been possible.

References

Andrade, M. V. A., Comba, J. L. D., and Stolfi, J. (1994) Affine arithmetic (extended abstract). In: *Proceedings of INTERVAL'94*, St. Petersburg, Russia.

Fang, C.F., Rutenbar, R.A., Püschel, M., and Chen, T. (2003) Towards efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In: *Design Automation Conference (DAC 2003)*, Anaheim, CA.

Gay, O. (2003) Libaffa—C++ affine arithmetic library for GNU/Linux. `http://savannah.nongnu.org/projects/libaffa/`.

Grabowski, D., Grimm, C., and Barke, E. (2006) Semi-symbolic modeling and simulation of circuits and systems. In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS) 2006)*, Kos, Greece.

Grimm, Christoph, Heupke, W., and Waldschmidt, K. (2004a) Refinement of mixed-signal systems with affine arithmetic. In: *Design, Automation and Test in Europe 2004 (DATE'04)*, Paris.

Grimm, C., Heupke, W., and Waldschmidt, K. (2004b) Semisymbolic modeling and analysis of noise in heterogeneous systems. In: *Forum on Specification and Design Languages (FDL'04)*, Lille, France.

Hartong, W., Hedrich, L., and Barke, E. (2002). Model checking algorithms for analog verification. In: *Design Automation Conference (DAC 2002)*, New Orleans, LA.

Henzinger, T. A. (1996) The theory of hybrid automata. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pp. 278–292.

Heupke, W., Grimm, C., and Waldschmidt, K. (2003) A new method for modeling and analysis of accuracy and tolerances in mixed-signal systems. In: *Proceedings of the Forum on Design Languages (FDL'03)*, Frankfurt, Germany.

Lemke, A., Hedrich, L., and Barke, E. (2002) Analog circuit sizing based on formal methods using affine arithmetic. In: *ICCAD 2002*.

Moore, R. E. (1966) *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

Stauner, T., Müller, O., and Fuchs, M. (1997) Using HyTech to verify an automotive control system. In: Maler, O. (ed) *Hybrid and Real-Time Systems—International Workshop, HART'97*, volume 1201 of *Lecture Notes on Computer Science*. Springer, Berlin, pp. 139–153.

Vachoux, A., Grimm, C., and Einwich, K. (2004). Towards analog and mixed-signal SoC design with SystemC-AMS. In: *IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)*, Perth, Australia.

Zhang, Y. and Mackworth, A. K. (1996) Specification and verification of hybrid dynamic systems with timed ∀-automata. In: Alur, R., Henzinger, T. A., and Sontag, E. D. (ed) *Hybrid Systems III*, volume 1066 of *Lecture Notes on Computer Science*. Springer, Berlin, pp. 587–603.

Chapter 10

SystemC-WMS: Mixed-Signal Simulation Based on Wave Exchanges

Simone Orcioni, Giorgio Biagetti, and Massimo Conti
Dipartimento di Elettronica, Intelligenza artificiale e Telecomunicazioni
Università Politecnica delle Marche
via Brecce Bianche, 12
I-60131 Ancona
Italy

Abstract This chapter proposes a methodology for extending SystemC to mixed-signal systems, aimed at allowing the reuse of analog models and to the simulation of heterogeneous systems. To this end, a general method for modeling analog modules using wave quantities is suggested, and a new kind of port and channel suitable to let modules communicate through waves have been defined. These entities are plugged directly on top of the standard SystemC kernel, so as to allow a seamless integration with the preexisting simulation environment, and are designed to permit total interconnection freedom to ease the development of reusable analog libraries.

Keywords: SystemC; mixed-signal simulation; system-level simulation.

1. Introduction

SystemC is an emerging tool for the description and simulation of hardware and software at system level (OSCI, 2006), and it is not rare that this high level of abstraction could require the interfacing of both digital and analog parts. Such necessity of simulating a continuous-time analog part can arise, for example, in the area of power switching control as in the automotive or RF domains. To this aim it has been proposed (Einwich, 2002) to constitute an Open SystemC Initiative (OSCI) Working Group devoted to the development of an extension of SystemC to mixed-signal simulation: SystemC analog and mixed signal (AMS). (Vachoux et al., 2003b) describes in detail the

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 171–185.

SystemC-AMS requirements and objectives. The first aspect considered is the need to encompass a variety of models of computation (MoCs) that can be used in order to describe any kind of system (discrete-event, data-flow, finite-state machines (FSMs), analog signal flow, generic continuous-time, etc.). Furthermore, SystemC must also extend to heterogeneous domains of application (i.e., electrical, mechanical, fluidic), due to the increasing complexity of nowadays devices.

The OSCI Working Group claims that SystemC-AMS, besides being suitable for the description and the simulation of heterogeneous systems and supporting continuous-time MoCs, must also meet the following objectives: it must be an extension of the current SystemC; it must provide a (possibly generic) way to handle interactions between MoCs; it must provide appropriate views for the description of continuous-time models; and, finally, it must support the coupling with existing continuous-time simulators. A recent description of the state of the art of this initiative can be found in Vachoux et al. (2003a).

The current implementation is structured into different layers. The solver layer provides simple but efficient solvers for linear differential equations and for explicit-form transfer functions. The synchronization layer provides a simple and fast synchronization scheme that executes analog solvers before the first delta cycle of each time step, scheduling them using static data flow. Finally, a view layer provides means for specifying equations, for instance, using netlists.

In addition to the activity performed by OSCI, different papers (Aljunaid and Kazmierski, 2004; Bjørsen et al., 2003; Bonnerud et al., 2001) aimed at the extension of SystemC to analog environments have been published. In Bjørsen et al. (2003), a mixed-signal simulation framework oriented to the simulation of signal processing–dominated applications is presented. The library modules proposed do not provide a real continuous-time modeling but a discrete-time domain regulated by a virtual clock or a multiphase clocking scheme. More recently, a mixed-signal extension using a general-purpose analog solver coupled with SystemC kernel by a lock-step synchronization algorithm has been proposed in Aljunaid and Kazmierski (2004). This implied a modification of the SystemC 2.0 kernel to invoke and synchronize the operation of an analog solver together with that of the core kernel.

The basic SystemC methodology (OSCI, 2006) makes use of modules and interfaces to describe complex systems. Modules communicate through interfaces, implemented in channels, by calling methods in the channels themselves. Conversely, events occurring in a channel can activate modules connected to that channel. The present work proposes a methodology for the description of analog blocks using only such instruments and libraries. Taking advantage of this communication scheme and of the underlying SystemC

kernel, we implement the various analog parts of a system as analog modules, which communicate by exchanging energy waves through wavechannel interfaces. The use of energy waves permits the definition of a standard analog interface that allows the interconnection of modules belonging to different domains as well as of modules developed independently. Furthermore, interconnections of analog blocks giving rise to simple kinds of nonlinear differential algebraic equations (DAEs) can also be simulated.

2. Description and Modeling of Analog Modules in SystemC

SystemC is essentially a library of C++ classes developed to build, simulate, and debug a system-on-chip (SoC) described at system level. It provides an event-driven simulation kernel and the functionality of the system derives from the interaction of concurrent processes that describe the behavior of individual modules subject to stimuli sent to them by other modules.

The core SystemC simulation paradigm assumes that modules have clearly defined inputs and outputs, and that they communicate between one another by means of appropriate channels. This paradigm allows the simulation to be carried out by a simple time-marching algorithm that only needs to take care of interactions between modules and the channels directly connected to them, without the need of dealing with the global system topology.

In order to be able to simulate systems containing analog modules some extensions to the base kernel are necessary. In cases where it is easy to obtain a signal-flow graph (SFG) representation of the system, this simulation paradigm can be coupled with an appropriate ordinary differential equation (ODE) solver as in Biagetti et al. (2004). This enables an efficient simulation of continuous-time analog modules described by a system of nonlinear ODEs of the following type:

$$\begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \\ \boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u}) \end{cases} \tag{10.1}$$

where $\boldsymbol{f}$ and $\boldsymbol{g}$ are vector expressions describing system dynamics, while $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{u}$ are state, output, and input vectors, respectively.

Equation 10.1 should describe a part of the system under consideration, like an N-port modeled at circuit level, or it may represent a high-level macromodel describing the part functionality. This description is not able to take into account parts that need a DAE system to be described, neither conservative-law systems; however, it is quite general and will thus be used to describe the behavior of a single module.

Nevertheless, a SFG representation is not always the most suited to model the interaction between modules representing analog units, since it can be hard to account for load effects or other interactions that might occur as they are

interconnected. The goal of the next section is to propose an extension of this approach to cases where a SFG representation of the modules is undesirable or excessively demanding.

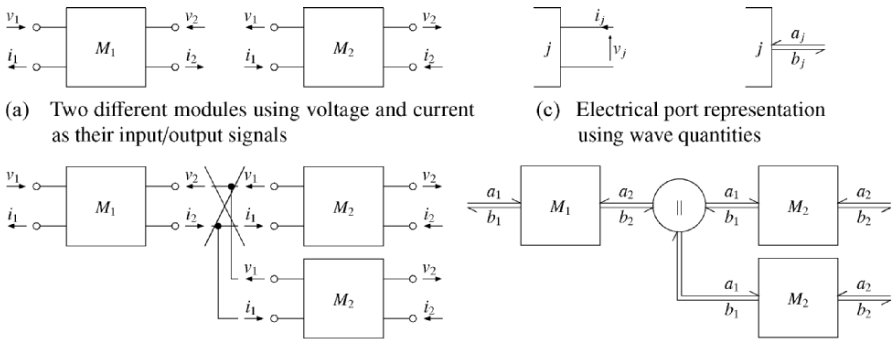
2.1 Module Representation with *a b* Parameters

The first problem that needs to be solved is the possibility of interconnecting modules written independently. Figures 10.1a and b depict one possible problem that can arise trying to bind together electrical modules that use currents or voltages as their input/output signals. Whatever the designer's choice was regarding what to consider input or output, it would not be possible to simultaneously connect them in series or in parallel, as well as to cascade them.

Furthermore, in non-SFG representations there can be no physical clue on which quantity to consider input or output of a given module. Even if in principle it is feasible to write a specialized channel that can handle all the possible combinations arising from a random choice, the resulting interconnection model would lack a physical meaning and would likely be cumbersome.

The use of an incident/reflected wave model (Kurokawa, 1965) for the description of analog modules allows us to avoid this difficulty since it can be mandated that modules use incident waves as inputs and produce reflected waves as outputs. This immediately solves the problem for cascaded modules, and the parallel or series connection can be accounted for by using an appropriate channel that dispatches waves to the modules it connects together (Figure 10.1d) and permits the formulation of a generic and standard analog interface usable across a variety of domains.

Such channel behaves similarly to the scattering junction of wave digital filters (WDFs; Fettweis, 1973), which are digital models of analog filters,



(a) Two different modules using voltage and current as their input/output signals

(c) Electrical port representation using wave quantities

(b) Some complex adaptor is needed to tie together the two output voltages

(d) Solution to the interconnection problem by a wave adaptor

Fig. 10.1 Example of interconnection problem

obtained through the discretization of individual circuit components by a methodology that can also be extended to circuits in which mildly nonlinear elements are present (Sarti and De Poli, 1999).

The proposed approach makes use, like in the WDF theory, of the $a\,b$ parameters as input/output signals and implements the duties of the scattering junction in a new entity called wavechannel, complying with SystemC conventions for channels. Furthermore, the user can choose the level of abstraction at which to model the system and the integration method (ODE solver) used to solve the continuous-time system.

Without loss of generality, we can fix our attention to an N-port in the electrical domain, described through its port quantities v_j and i_j, $j = 1, \ldots, N$. Figure 10.1c depicts the situation for a single port. The relation between electrical quantities and wave quantities can be obtained from the following definition of incident (a_j) and reflected (b_j) wave:

$$\begin{aligned} a_j &= \frac{1}{2}\left(v_j/\sqrt{R_j} + i_j\sqrt{R_j}\right) \\ b_j &= \frac{1}{2}\left(v_j/\sqrt{R_j} - i_j\sqrt{R_j}\right) \end{aligned} \tag{10.2}$$

so that $a_j^2 - b_j^2$ is the instantaneous power entering port j and R_j is a normalization resistance. Similar relations hold for other domains as well. In the frequency domain, this representation leads to the commonly adopted description with a scattering matrix, and the normalization resistance can be assumed like the characteristic impedance of the transmission line connected to the port.

Solving the system Equation 10.2 for the electrical quantities gives the inversion formulae:

$$\begin{aligned} v_j &= \left(a_j + b_j\right) \cdot \sqrt{R_j} \\ i_j &= \frac{\left(a_j - b_j\right)}{\sqrt{R_j}} \end{aligned} \tag{10.3}$$

that can be useful when translating module descriptions from one set of quantities to the other.

Let us suppose that a port is defined by means of a relation of the type of Equation 10.1, where in the electrical domain $\{u, y\} = \{v, i\}$ (while in other domains we can find, for example, force and velocity or pressure and volume velocity as port variables). It is straightforward to build the representation of an N-port with the $a\,b$ parameters by using Equations 10.1 and 10.2, thus obtaining:

$$\begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{a}) \\ \boldsymbol{b} = g_1(\boldsymbol{x}, \boldsymbol{a}) \end{cases}, \tag{10.4}$$

which are the state space equations written in wave quantities.

2.2 Wavechannels

Wavechannels are the means by which modules described by wave quantities communicate. They can be thought of as a bunch of transmission lines connecting ports to a junction box, in which the lines are tied together, and their role is to model the scattering of waves that occurs at the junction.

Consider a junction between N ports, each with its own normalization factor R_j. Let $\boldsymbol{v}$ and $\boldsymbol{i}$ be the voltage and current vector, respectively, and:

$$\begin{cases} \boldsymbol{A_v}\,\boldsymbol{v} = \boldsymbol{0} \\ \boldsymbol{A_i}\,\boldsymbol{i} = \boldsymbol{0} \end{cases} \tag{10.5}$$

be a complete and minimal set of Kirchhoff's equations describing the junction ($[\boldsymbol{A_v}]_{ij}, [\boldsymbol{A_i}]_{ij} \in \{0, \pm 1\}$). We maintain that letting:

$$\boldsymbol{A_x} = \boldsymbol{A_v} \operatorname*{diag}_{k=1,\ldots,N} R_k \quad \text{and} \quad \boldsymbol{A_y} = \boldsymbol{A_i} \operatorname*{diag}_{k=1,\ldots,N} 1/R_k \tag{10.6}$$

the scattering matrix $\boldsymbol{S}$ (such that $\boldsymbol{a} = \boldsymbol{S}\,\boldsymbol{b}$), by substituting Equations 10.3 and Equation 10.6 into Equation 10.5, becomes:

$$\boldsymbol{S} = \begin{bmatrix} \boldsymbol{A_x} \\ \boldsymbol{A_y} \end{bmatrix}^{-1} \begin{bmatrix} -\boldsymbol{A_x} \\ \boldsymbol{A_y} \end{bmatrix} \tag{10.7}$$

where $\boldsymbol{b}$ are the waves reflected by modules and thus entering the junction, whence $\boldsymbol{a}$ are scattered back from the junction to the modules thereby interconnected.

The above formulation can be used for any kind of junction. But, although there are many possible ways in which the lines can be tied together, the most common situation is to have parallel or series connections, as shown in Figure 10.2 for channels connecting three ports. From Kirchhoff's laws, a parallel connection is characterized by the equations:

$$\sum_{j=1}^{N} i_j = 0 \qquad v_1 = v_2 = \cdots = v_N \tag{10.8}$$

for which the scattering matrix described by Equation 10.7 results in

$$a_j = \frac{2\sum_{k=1}^{N} b_k/\sqrt{R_k}}{\sqrt{R_j}\sum_{k=1}^{N} 1/R_k} - b_j. \tag{10.9}$$

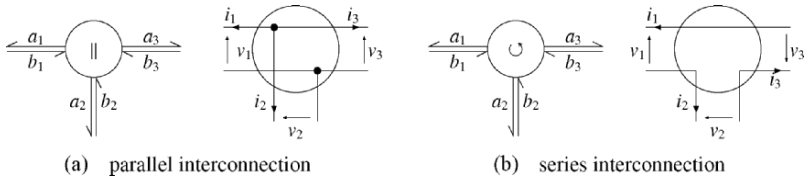


Fig. 10.2 Wavechannel symbols for parallel and series interconnections

Similarly, for a series wavechannel we have

$$\sum_{j=1}^{N} v_j = 0 \qquad i_1 = i_2 = \cdots = i_N, \tag{10.10}$$

which leads to:

$$a_j = b_j - \frac{2\sum_{k=1}^{N} b_k \sqrt{R_k}}{\sum_{k=1}^{N} R_k / \sqrt{R_j}}. \tag{10.11}$$

It may be worth noticing here that, if $N = 1$, Equations 10.9 and 10.11 simply become $a_1 = \pm b_1$, and the two channel types are thus able to model the total reflection that takes place at an open circuit or at a shunt, respectively.

In the current implementation of wavechannels, the propagation delay can be excluded so that their connection to instantaneous blocks may result in the production of delay-free loops. This is accounted for by the standard SystemC delta cycle mechanism, which, without further intervention, would just use a fixed-point algorithm to search for the solution of the instantaneous loops, provided that the embedded ODE solver does not advance its state while iterating to find the fixed-point. The fixed-point solution is equivalent to the solution of Maxwell's equations in quasistatic conditions, i.e., when it is possible to model the circuit with lumped elements. The quasistatic condition is valid if the wave propagation delay τ is much smaller than the Δt used by ODE solvers. In our fixed-point solution method this τ is approximated with a null time.

Furthermore, to increase the convergence speed of the fixed-point algorithm, a damping effect has been introduced. This has been done on the basis that, in a time-marching simulation, states between successive time steps should not be very different, and thus the fixed-point solution may take advantage of a limitation in the amount of change allowed to the variables. Let $a^{(n)}[t]$ be the wave at the n-th delta cycle of the time step t. The evaluation of the module output functions, based on the values of the inputs $a^{(n)}[t]$ and of the state $x[t]$, yields the reflected wave $b^{(n+1)}[t]$. This is used in Equations 10.9 and 10.11 to compute the scattering due to interconnections; let us call $\tilde{a}^{(n+1)}[t]$ the result. We then put

$$a^{(n+1)}[t] = a^{(n)}[t] + \lambda(\tilde{a}^{(n+1)}[t] - a^{(n)}[t]) \tag{10.12}$$

where λ is a positive constant less than 1 (we obtained good results with values close to 0.9), governing the amount of damping. The update of a is skipped altogether when the amount of change is below a predefined threshold related to the desired accuracy of the solution, so as to exit from the delta cycling and thus allowing the time to be incremented and the state of ODE solvers to be updated.

With this approach, it has been possible to obtain accurate simulations with a reasonable convergence speed of most of the systems containing delay-free loops, provided they do not contain directly coupled state variables, that is, the circuit has a solution for every possible value of the state variables.

3. SystemC-WMS Class Library

To ease the implementation of complex systems containing analog blocks, a number of templates and classes have been designed and integrated in the SystemC-wave mixed signal (SystemC-WMS) class library: a new kind of port to let modules communicate through wave quantities (`ab_port`), a channel that can interconnect them and that does the real computation of the scattering that occurs at junctions (`ab_signal`), and a template base class (`wave_module`) that takes care of handling sensitivity lists and port declarations.

Ports expose an interface that allows users to read the incident wave value and to report (write) the reflected wave value, together with utility functions to poll for changes and to get other channel properties:

```
template <class T> struct ab_signal_if : virtual sc_interface
{
  virtual bool poll () const = 0;
  virtual const T read () const = 0;
  virtual void write (const T &) = 0;
  ...
};
```

The basic `wave_module` template looks like the following:

```
template <int n, class T> struct wave_module : sc_module, ...
{
  ab_port <T> port[n];
  sc_event activation;
  ...
};
```

where `port` is the array (or possibly a single variable if `n=1`) of ports used by the module to communicate. Of course, they can be freely mixed with standard SystemC ports. The `activation` is an event that is signaled when some change occurs at the waves entering any of the ports, and the template parameter `T` must be associated to a structure, which essentially consists of a collection of typedefs, needed to define the underlying data type used for waves and to document the nature of the port. A number of predefined natures (electrical, mechanical, etc.) have been provided, and, of course, templates to ease the implementation of transducers (that is, modules with ports of different natures) have also been defined and implemented.

With this library the only thing that the user needs to do in order to model an analog module is to implement the state derivative vector field f and output transformation function g as in Equation 10.4:

```
struct example : wave_module <1, electrical>, analog_module
{
  // state variable x is inherited from analog_module
```

```
4     void field (double *var) const;
5     void calculus ();
6     SC_CTOR(example) : analog_module(...)
7     {
8       SC_THREAD(calculus);
9       sensitive << activation;
10    }
11  };
12
13  void example::calculus ()
14  {
15    x = 0;                      // state initialization here
16    while (step()) {            // perform one ODE solver step
17      double a = port->read(); // read incident wave here
18      double b = g(x, a);       // compute reflected wave
19      port->write(b);           // and send it out here
20    }
21  }
22
23  void example::field (double *var) const
24  {
25    double a = port->read();
26    var[0] = f(x, a);           // evaluate state change
27  }
```

The `step` function is inherited from the `analog_module` and contains a simple time-marching ODE solver (Biagetti et al., 2004). Currently, the user has a choice of Euler and Adams-Bashforth ODE solvers, but the implementation of other time-marching ODE solvers should be straightforward.

Finally, `ab_signal` takes care of making communication between modules possible. These signals can just be declared by specifying the nature of the ports and the kind of connection topology to make between them, with an optional default normalization resistance, for instance:

```
1    ab_signal <electrical, parallel> test_signal_1(50 ohm);
2    ab_signal <electrical, series>   test_signal_2(10 ohm);
```

and then connected to ports like ordinary SystemC signals.

4. Application Example

As an example of a possible application of this extension to a real problem, a half-bridge inverter has been chosen. A simplified schematic of the circuit is shown in Figure 10.3. It is used to drive an *RLC* load for an induction-heating appliance. The function of the circuit is to regulate the delivery of power to a

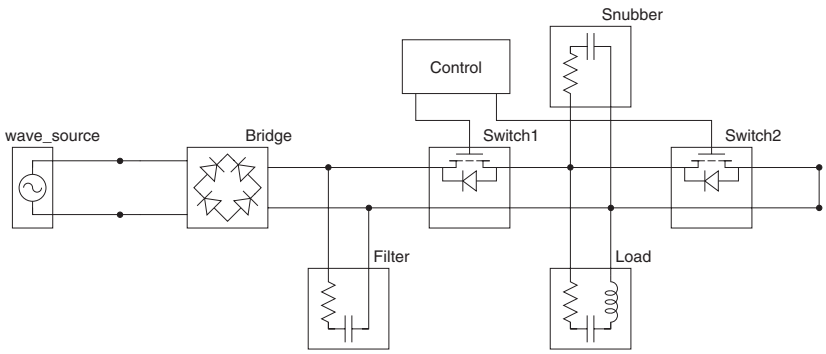


Fig. 10.3 Half-bridge inverter: electrical schematic diagram

load. The amount delivered can be set by changing the duty cycle and/or the frequency of the signals controlling the two switches with a proper algorithm that can be implemented in digital hardware. Of course, the maximum output power corresponds to a 50 % duty cycle at the resonance frequency, but in the proposed example we have not modeled the details of the digital controller and have thus chosen a 48 % duty cycle, for safe operation of the switches, and a fixed frequency of 20 kHz.

The main components of the circuit are the switches (`controlled_rectifier`), the Graetz' bridge used to rectify the line voltage (`ideal_rectifier`), and the voltage source (`wave_source`) used to convert stimuli from standard SystemC signals or an SFG representation to the wave representation. Furthermore, there are a couple of different passive reactive linear networks (`RC` and `RLC`). All of the analog stuff is connected together by means of wavechannels, as shown by the following code fragment illustrating the structure of the circuit under consideration. A brief description of the most important modules follows the code.

```
1  int sc_main (int argc, char *argv[])
2  {
3    sc_signal <electrical::wave_type> in;
4    sc_signal <bool> pulse1, pulse2;
5    ab_signal <electrical, parallel> mains;
6    ab_signal <electrical, parallel> rectified;
7    ab_signal <electrical, parallel> chopped;
8    ab_signal <electrical, series>   shunt;
9
10   generator signal_source("SOURCE1", 230 V, 50 Hz);
11   signal_source(in);
12
13   controller ctrl("CONTROL", 20 kHz);
14   ctrl(pulse1, pulse2);
```

```
15
16    source <electrical> wave_source("SOURCE2", cfg::across);
17    wave_source(mains, in);
18
19    ideal_rectifier bridge("BRIDGE");
20    bridge(mains, rectified);
21
22    RC filter("FILTER", 1 ohm, 5 uF);
23    filter(rectified);
24
25    controlled_rectifier switch1("SWITCH1");
26    switch1(chopped, rectified, pulse1);
27
28    controlled_rectifier switch2("SWITCH2");
29    switch2(shunt, chopped, pulse2);
30
31    RLC load("LOAD", 3 ohm, 80 uH, 740 nF);
32    load(chopped);
33
34    RC snubber("SNUBBER", 10 ohm, 10 nF);
35    snubber(chopped);
36
37    sc_start(150e-6, SC_SEC);
38    return 0;
39  }
```

The source class is a generic converter from standard SystemC signals to wave signals, and the second parameter to its constructor specifies an "across"-type (as opposed to a "through"-type) source, which is a voltage source in the electrical domain. The controlled_rectifier module models the behavior of an ideal switch, like an MOS switch with zero on resistance, coupled in parallel with a bypass ideal diode. It has been modeled as a 2-port module with an additional logical input to control the switch. For simplicity, the assumption that normalization resistances are the same for both ports has been made in its formulation, and so imposed in its constructor. That way, in its conducting state, whether it is due to the transistor switched on or to the diode, it simply lets waves through (like a transparent channel), otherwise it reflects them backwards (like a couple of open circuits). Its implementation is shown in the following:

```
1  struct controlled_rectifier : wave_module <2, electrical>
2  { // Ideal switch with integrated ideal diode
3    SC_HAS_PROCESS(controlled_rectifier);
4    controlled_rectifier (sc_module_name name);
5    void calculus ();
6    sc_in <bool> control;
7  };
8
```

```
controlled_rectifier::controlled_rectifier ...
{
  SC_METHOD(calculus);
  sensitive << activation << control;
  // sets normalization resistances on both ports to be the ↵
      same:
  port[0] <<= port[1] <<= 1;
}

void controlled_rectifier::calculus ()
{
  double a0 = port[0]->read(), a1 = port[1]->read();
  bool diode_on = a0 > a1, switch_on = control->read();
  bool on = diode_on || switch_on;
  port[0]->write(on ? a1 : a0);
  port[1]->write(on ? a0 : a1);
}
```

When the switch is off, the detection of the state of the diode is done by looking at port voltages; but if the diode is off too, port voltages are proportional to incident waves, since $b_j = a_j \Rightarrow v_j = 2a_j\sqrt{R_0}$, so it is perfectly legal to test the latter ones. A similar formulation models the diode bridge in the `ideal_rectifier` module.

For what concerns the linear components, they are all modeled according to Equation 10.4 and the example module reported in Section 3. In particular, the equations governing the RLC circuit in wave quantities are:

$$\begin{cases} \dot{x}_0 = 2a\sqrt{R_0} - (R + R_0)x_0/L - x_1/C \\ \dot{x}_1 = x_0/L \\ b = a - x_0\sqrt{R_0}/L \end{cases} \tag{10.13}$$

where the state vector $\boldsymbol{x}$ is composed of $x_0 = Li_L$ and $x_1 = Cv_C$ with obvious meaning of the symbols. This directly translates into the following state update and output computation functions:

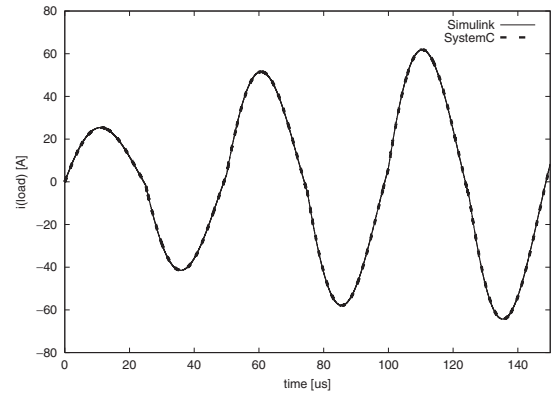
```
void RLC::field (double *var) const
{
  double a = port->read()
  var[0] = 2*a*sqrt(R0) - state[0]*(R + R0)/L - state[1]/C ;
  var[1] = state[0]/L ;
}

void RLC::calculus ()
{
  R0 = port->get_normalization();
  while (step()) {
    double a = port->read();
    double b = a - state[0]*sqrt(R0)/L;
```

```
14        port->write(b);
15      }
16    }
```

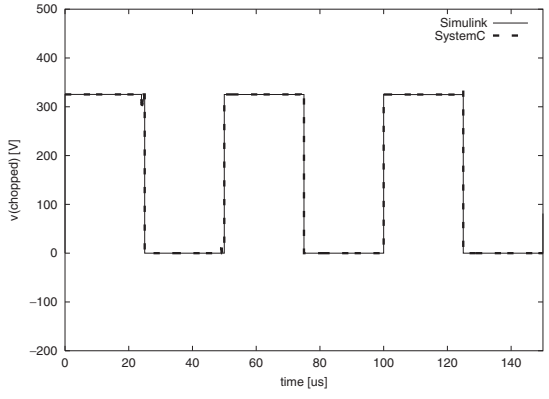
that, together with the obvious declaration of the RLC structure, complete the definition of the module. A model for the RC module can be derived similarly. The complete source code for this example, the full library, and other applications are available from the authors' website (Biagetti et al., 2006).

4.1 Simulation Results

The circuit has been simulated using the proposed SystemC extension, which uses a fourth-order Adams-Bashforth ODE solver, and the results compared to a Matlab™ simulation done with the Simulink Power toolbox using the `ode15s` stiff ODE solver. Excessively long simulation times with the Matlab



(a) Current through the load



(b) Voltage across the switches

Fig. 10.4 Simulation results of the half-bridge inverter

`ode113` solver (Adams-Bashforth), suited for nonstiff systems, led us to believe that testing both simulators with the same solver algorithm could not be very significant because of their different application contexts (which is a single module in our simulator).

Nevertheless, using an adaptive time step with the same maximum Δt of 5 ns, we obtained a simulation time of 11.3 s with SystemC-WMS and 2.3 s with Matlab™, both running on an Intel™ Pentium™ M processor at 1000 MHz.

Results are shown in Figure 10.4, where load current and chopped voltage are plotted as a function of time. The curves are completely overlapping.

5. Conclusion

The increasing complexity of systems and circuits asks for an easy way to model and simulate the overall behavior of a complex system spanning multiple domains. In order for SystemC to be able to cope with these requirements, an extension aimed at allowing the modeling and simulation of analog circuits is mandatory.

This work proposes an effective, and still not excessively complex framework, that simplify the modeling of the interaction between analog models belonging to heterogeneous domains, as well as model reuse. By using power waves as standard input/output signals for analog modules, these can be independently modeled and freely interconnected together in arbitrary topologies without having to deal with complex interface compatibility issues. Moreover, this allows for a uniform treatment of heterogeneous domains, thus paving the way to the development of truly generic and reusable model libraries.

The first results are encouraging in terms of accuracy of simulation and, despite the simplicity of the algorithms employed, the variety of the class of circuits that can be simulated.

References

Aljunaid, H. and Kazmierski, T. J. (2004) SEAMS—a SystemC environment with analog and mixed-signal extensions. In: *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS'04)*, volume 5, IEEE, pp. V–281–V–284.

Biagetti, G., Caldari, M., Conti, M., and Orcioni, S. (2004) Extending SystemC to analog modeling and simulation. In: Grimm, C. (ed) *Languages for System Specification—Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems and Property Specifications from FDL'03*, CHDL, chapter 15. Kluwer Academic Publishers, Norwell, MA, pp. 229–242.

Biagetti, G., Conti, M., and Orcioni, S. (2006) SystemC-WMS home page. `http://www.deit.univpm.it/systemc-wms/`.

Bjørsen, J., Bonnerud, T. E., and Ytterdal, T. (2003) Behavioral modeling and simulation of mixed-signal system-on-a-chip using SystemC. *Analog Integrated Circuits and Signal Processing*, 34:25–38.

Bonnerud, T. E., Hernes, B., and Ytterdal, T. (2001) A mixed-signal, functional level simulation framework based on SystemC for system-on-a-chip applications. In: *Proceedings of the 2001 IEEE Conference on Custom Integrated Circuits*. IEEE, pp. 541–544.

Einwich, K. (2002) Analog mixed signal extensions for SystemC. White paper and proposal for the foundation of the SystemC-AMS OSCI working group, Fraunhofer-Institut für Integrierte Schaltungen IIS, Außenstelle Entwurfsautomatisierung EAS, `http://mixsigc.eas.iis.fhg.de/`.

Fettweis, A. (1973) Pseudopassivity, sensitivity and stability of wave digital filters. *IEEE Transactions on Circuit Theory*, CT-19:668–673.

Kurokawa, K. (1965) Power waves and the scattering matrix. *IEEE Transactions on Microwave Theory and Techniques*, 13(2):194–202.

OSCI (2006) *SystemC Documentation*. The Open SystemC Initiative (OSCI), `http://www.systemc.org/`.

Sarti, A. and De Poli, G. (1999) Toward nonlinear wave digital filters. *IEEE Transactions on Signal Processing*, 47(6):1654–1668.

Vachoux, A., Grimm, C., and Einwich, K. (2003a) Analog and mixed signal modelling with SystemC-AMS. In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'03)*, volume 3. IEEE, pp. III–914–III–917.

Vachoux, A., Grimm, C., and Einwich, K. (2003b) SystemC-AMS requirements, design objectives and rationale. In: *Proceedings of Design Automation and Test in Europe (DATE'03)*, Paris.

Chapter 11

Automatic Generation of a Coverification Platform

Suad Kajtazovic[1], Christian Steger[1], Andreas Schuhai[2], and Markus Pistauer[2]

[1] *Institute for Technical Informatics*
Graz University of Technology
Inffeldgasse 16/I
A-8010 Graz
Austria

[2] *CISC Semiconductor Design+Consulting GmbH*
Lakeside B07
A-9020 Klagenfurt
Austria

Abstract Complex microelectronic embedded systems are mostly subdivided into several subsystems, which are designed in different hardware description languages (HDLs) to get best system performances. Verification of all subsystems in one environment presents a difficult task. One of the possibilities to solve this problem is using cosimulation techniques and standard HDL simulators. This chapter focuses on automatic building of cosimulation interfaces and model sources extracted from a HDL-independent system description. Moreover, it presents a design methodology, which introduces advanced cosimulation techniques to be used in mixed-signal system design. The proposed cosimulation technique and the automation methods have been applied on an application framework for multi-HDL system verification and later on evaluated by an example taken from the automotive industry.

Keywords: heterogeneous system design; cosimulation; design automation; distributed computing.

1. Introduction

Verification plays an important role in the design of microelectronic embedded systems that become more and more complex. The complex systems consist

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 187–203.

mostly of subsystems designed in different, hardware description languages (HDLs) to get best performance provided by used HDLs (e.g., hardware/ software subsystems). The verification of heterogeneous systems in one environment is very difficult. Currently, there are three possibilities to verify such systems. The first one is to translate corresponding subsystem descriptions from the foreign HDL into the target HDL, which can involve a degradation of model performance. The second possibility considers that the used simulator supports multi-HDL system design, which is preferred by many electronic design automation (EDA) tool vendors. A cosimulation presents the third possibility for the verification of heterogeneous systems. In a cosimulation, subsystems are simulated using HDL-specific simulators and the communication between them has been established by a cosimulation interface. In recent years, use of cosimulation techniques becomes a more and more interesting solution for the verification of heterogeneous system designs.

In this chapter we present an advanced cosimulation technique to be used in system design of microelectronic embedded systems. Automation in system design is one of the guidelines followed in this work. This work concentrates on design of a generic cosimulation interface and automatic building of a verification platform. Moreover, it presents a methodology to be used in multi-HDL system design.

This chapter is organized as follows. Section 2 describes cosimulation tools and methods, which are related to this work. The proposed design methodology has been described in Section 3. Section 4 describes the proposed generic cosimulation interface. One important topic in this work is the automatic code generation, which is described in Section 5. The proposed approach has been illustrated in Section 6 by an example taken from automotive industry. Section 7 concludes this work.

2. Related Work

In the past few years, interfacing between different simulators and cosimulation techniques have been investigated intensively. Many different solutions have been found. A short overview about the used technologies and tools is given here.

CosiMate (TNI-Software, 2005) is an EDA-tool, which uses a common cosimulation bus based on the open application programming interface (API). A graphical user interface supports the configuration of the cosimulation bus. Several simulators can be coupled into one cosimulation environment: Matlab/Simulink, Saber, SystemC, VHDL/Verilog Simulators from Mentor-Graphics, Cadence, etc. CosiMate links these simulators through a configuration file and C interfaces to create the cosimulation bus.

Seamless (Mentor Graphics, 2005) provides a cosimulation with several instruction set simulator (ISS) and HDL simulators. The cosimulation is based on the C-bridge API. The cosimulation backplane is tool-independent and supports several of today's popular simulators. The setup of the cosimulation must be done manually.

Link for ModelSim (The MathWorks, 2005) was presented by MentorGraphics and MathWorks as a solution for the cosimulation between Matlab/Simulink and the ModelSim simulator. It uses the standard communication layer (socket connection) for the data transfer. It integrates Matlab/Simulink into the hardware design flow for the development of field programmable gate array (FPGA) and application-specific integrated circuit (ASIC).

DCB (distributed cosimulation backbone; de Mello and Wagner, 2002) is based on the high-level architecture (HLA) method for the generation of distributed cosimulation interfaces. DCB serves as a common interface for different simulator types. Each simulator can be connected by ambassadors to the DCB backbone. Ambassador controls the data exchange between DCB backbone and connected simulator. DCB supports both synchronous and asynchronous simulation. Therefore, rollbacks are possible. The DCB has been defined in the scope of the SIMOO project (Copstein et al., 1997).

Ptolemy II (Brooks et al., 2004) is a platform for modeling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II supports several computation models (e.g., time-discrete, discrete-event, time-continuous, etc.), which are called domains. Many predefined execution units, called actors, are available as well as a graphical editor for the system modeling. Ptolemy II provides interfaces to most of today's popular simulators as well.

2.1 Summary

The two relevant approaches for implementing the cosimulation environment in this work are the backplane-based simulation and the direct coupling. Almost every cosimulation framework recently proposed uses the simulation backplane. For example, the centralized scheme facilitates user interaction and the open API benefits in seamless coupling of different types of simulators. However, the backplane also has considerable drawbacks, such as the performance bottleneck caused by the centralized communication.

As a consequence of the backplane's drawbacks, this work bases on direct coupling of simulators without a central control unit. The proposed communication scheme is more flexible than a cosimulation backplane. However, the cosimulation is supported by an automatic code generation, which examines the whole system and not only the interfacing between two models.

3. Design Methodology

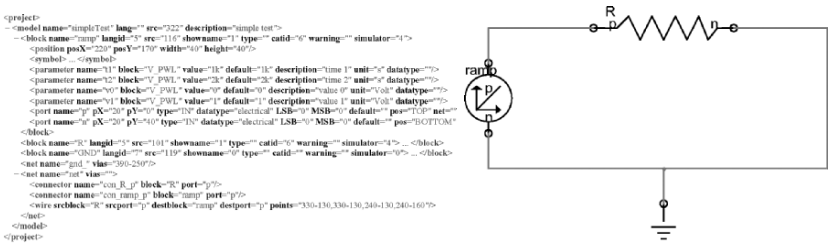
Based on top-down system design using intellectual property (IP) models, the proposed methodology subdivides the system design into three different levels:

1. System design level
2. Language level
3. Simulator level

3.1 System Design Level

At the system design level, a heterogeneous system is described using a language-independent description semantic that enables integration of subsystems designed in different HDLs. At this level the functional descriptions of models are not necessary since the system has been designed using provided IP models. Concerning that, a system description becomes language-independent. It opens a possibility to use the same description semantic for models written in different HDLs. For the system description, system hierarchy, and schematic representation eXtensible markup language (XML) has been used. The essential benefits of XML are language independency, legibility, compactness, and support of hierarchical structures. An example of a system description using XML is depicted in Figure 11.1a and its graphical representation in a schematic editor in Figure 11.1b.

The simple model consists of three blocks (`ramp`, `R`, and `GND`), which are connected with two connections (`net` and `gnd_`). As depicted, at the system design level (in this case a top-level description) no functional description of models has been used. Only the model references and model parameters have been used.



(a) Using XML (b) Graphical representation

Fig. 11.1 System description

3.2 Language Level

At the language level the system has been enhanced to meet the requirements of target simulation environment. This includes splitting of the designed system into subsystems, which are later on grouped by its HDL or by required simulators (e.g., for parallel simulation), whereby the special cosimulation blocks are inserted between foreign subsystems. At this level the model sources including cosimulation interfaces have been generated. The performed modification does not have an influence on the behavior of the designed system.

3.3 Simulator Level

At the simulator level the modified system description serves as an outline for code generation and setup of the cosimulation platform. Involved simulators communicate through integrated cosimulation interfaces. The data flow and simulation of the whole system is controlled by a synchronization mechanism.

An application framework has been developed to support system design based on the proposed design methodology. Language and simulator levels described above are design steps in a heterogeneous system design that are performed in this framework fully automatically.

4. Design of a Cosimulation Interface

The first and rather more challenging design effort of this work concentrates on a proper coupling mechanism that enables the incorporation of several different simulators into a heterogeneous cosimulation environment. The finally developed communication scheme allows flexible, reasonably efficient, and time-accurate heterogeneous cosimulation. The architecture of the implementation of the coupling mechanism should be designed as an easy-to-use, customizable, and extensible framework. The developed concept is inspired by many aspects of the previously introduced cosimulation approaches. By means of evaluations of relevant simulation principles and resulting design decisions, this section presents the overall design of the so-called cosimulation interface, which is the central component used to interconnect different simulators.

4.1 Interfacing Between Simulators

In general, there are two possibilities to interface different simulators. The first approach is based on file input/output (I/O) primitives, which is sometimes used for less complex hardware/software cosimulation applications. The more common and flexible communication approach uses open interfaces provided by the participating simulators. Communication by means of file I/O operations is rather simple and provides simulator independency, but it is inflexible and does not meet all intended requirements. Hence the coupling mechanism

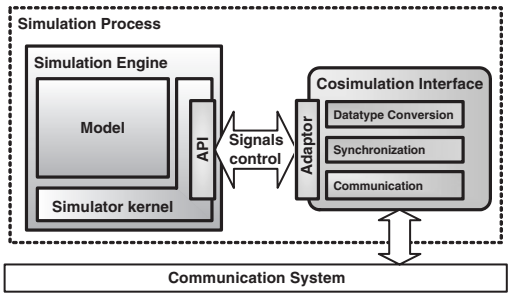


Fig. 11.2 Simulator interfacing principle

proposed by this work utilizes open interfaces of simulators. Prominent simulator interface examples are foreign language interface (FLI) (Model Technology, 2001) and programmable logic interface (PLI) (Mittra, 1999). Figure 11.2 depicts the key principle of the simulator interfacing. The cosimulation interface component (`CsInterface`) hooks on the provided open C/C++ API. As no standardized simulator API has been developed yet, an adaptor is needed to adjust the basic cosimulation interface to the simulator-specific API. All cosimulation interface components are interconnected by means of a communication system.

Communication System. Since the communication mechanism has a deep impact on simulation performance and accuracy, the communication system must be planned carefully and several aspects have to be considered. The following two requirements influence the design of the communication mechanism:

- Discrete-event simulation uses a low time abstraction, therefore the communication should be fast and efficient (i.e., low overhead and short transmission times).
- Distributed simulation possibly on different platforms demands a network-compatible communication mechanism. The used communication system has to be portable to different platforms, e.g., Linux and Microsoft Windows.

There are two main paradigms for interconnecting simulators. One is based on shared resources (shared memory) and the other one lets simulators communicate through channels. The first one is highly efficient but not applicable for distributed simulation. The latter paradigm offers the choice between several communication techniques, such as transmission control protocol/Internet protocol (TCP/IP) or user datagram protocol (UDP) socket connections, remote procedure call (RPC) or higher abstract interprocess communication

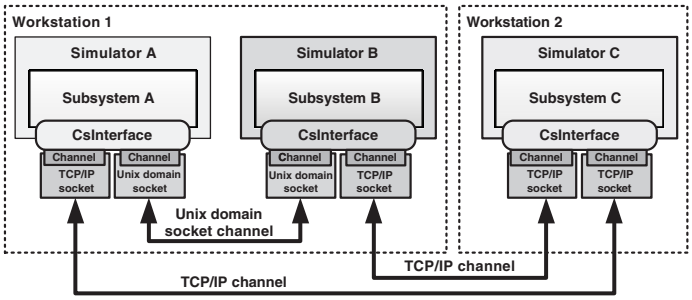


Fig. 11.3 Proposed coupling mechanism

(IPC) mechanisms. The proposed communication mechanism of this work utilizes the idea of abstract communication channels, i.e., simulators communicate with each other through abstract channels. An abstract channel provides a uniform interface and encapsulates the underlying communication mechanism. Hence different channel implementations can easily be exchanged. This provides the opportunity to select the most appropriate communication mechanism for a given configuration. This work prefers low abstract methods to high abstract and convenient communication frameworks, such as common object request broker architecture (CORBA).

Coupling Mechanism. Summarizing, the proposed cosimulation environment uses a decentralized coupling mechanism. Communication, synchronization, data conversion, and control functionality is distributed across all simulators and implemented by means of cosimulation interfaces. Cosimulation interfaces communicate through abstract, peer-to-peer channels. Therefore, each cosimulation interface establishes one connection to every other simulator. Figure 11.3 depicts the principle of the proposed coupling mechanism.

Abstract channels provide basic communication functionality by means of a simple uniform interface. Basically, the package-oriented channel interface provides two operations: `readPackage` and `writePackage`. These operations must be of blocking nature. The example shows the needed peer-to-peer connections between three simulators. Simulators A and B are placed on the same workstation, hence they are connected by a shared memory channel. Assuming that workstation 1 and 2 are both sited in a network (LAN/WAN), the peer-to-peer connections between these workstations are built on a TCP/IP channel.

4.2 Communication

The basic coupling and communication mechanism builds the foundation needed by the three main tasks of the cosimulation interface component: data communication, data type conversion, and synchronization.

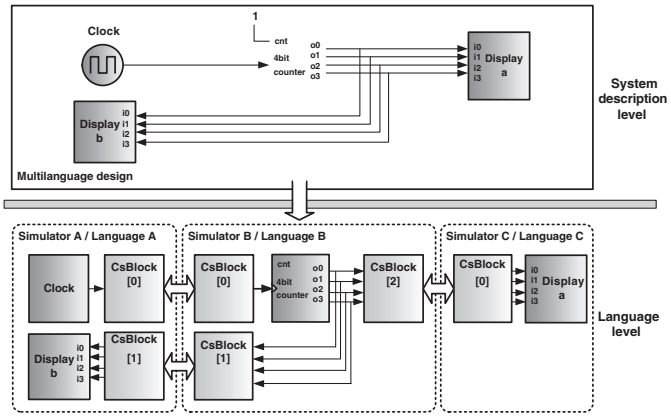


Fig. 11.4 Simple example of the `CsBlock` concept

The `CsBlock` modules are used to exchange signals between different simulation engines. It is obvious that data type conversion also takes place in the `CsBlock` module. One `CsBlock` is connected to local blocks with input and output ports and propagates signal changes to the corresponding `CsBlock` on a remote simulator. Clearly, cosimulation blocks always appear in pairs. Each `CsBlock` corresponds to a counterpart sited on a remote simulator. The simple example shown in Figure 11.4 illustrates this concept of using cosimulation blocks to exchange signals between modules on separated simulators. At the system design level only the structure of the model is represented.

4.3 Data Type Conversion

Considering data conversion more precisely, one have to distinguish between language-level and simulator-level conversion. At the language level, the cosimulation tool has to find equivalent data types for different languages. This usually happens at compile time (code generation phase). At the simulator level, the signal values have to be converted to the simulator's internal representation, which happens at run time. Converting data types is the next step done by the interface modules. This step is required to convert values from one data format to another. The data type conversion is predefined during the code generation phase at the language level. It has been implemented using a simple hash-table, which searches for equivalent data type for a certain language. In order to allow a coupling between several simulators with different data representations, it is a good practice to convert data values to a uniform intermediate format. This work uses the American Standard Code for Information Interchange (ASCII) format for intermediate data representation.

Listing 11.1 Basic synchronization algorithm

```
1 while unprocessed events remain do
2   send and receive messages generated in the previous iteration
3   LBTS = min_i(N_i + LA_i)              // N_i = time of next event in LP_i
4   process events with time stamp ≤ LBTS   // LA_i = lookahead of LP_i
5   barrier synchronisation
6 end while
```

4.4 Synchronization

The cosimulation environment structure is determined by the performance advantage of a decentralized communication scheme over a centralized one, so that the synchronization method has to be decentralized. The conservative protocols are relatively straightforward to implement and can be optimized by adjusting a few model-specific parameters. Basically, they have a performance advantage over synchronous protocols. The chosen synchronization method is actually implemented as a decentralized, "synchronous", conservative protocol (Calinescu, 1995). The term synchronous refers to the implementation principle that is based on barrier synchronization. The basic algorithm is shown in Listing 11.1. Each iteration consists of the following steps:

- Compute a lower bound on the time stamp (LBTS) of events that might be received later
- Events with time stamp ≤ LBTS are safe to process
- Process safe events, exchange messages
- Global synchronization (barrier synchronization)

The barrier synchronization method descends from the bulk synchronous parallel (BSP) model, which nowadays is a generic target for conservative discrete-event simulations.

4.5 Cosimulation Interface

Figure 11.5 depicts the rough architecture of entire simulator interface with its relevant components. `CsBlock` modules represent blocks simulated on a remote simulator. If a signal of a sensitive input port changes, the `CsBlock` module converts the signal value into an intermediate representation and packages it into a remote event. Afterwards it forwards the event to the `CsInterface` component. A `CsBlock` also receives event messages and writes the packaged signal values to the corresponding output ports. `CsInterface` is the central component of the simulator interface responsible for event exchanging and

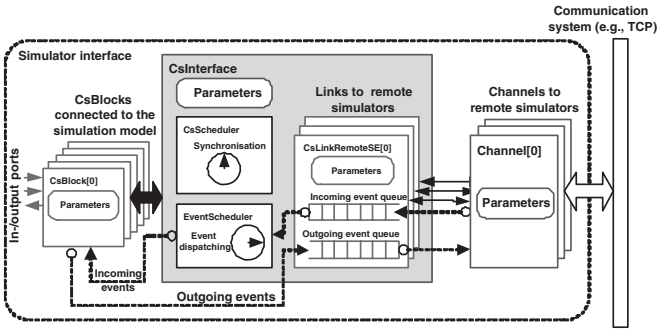


Fig. 11.5 Simulator interface structure

synchronization. It contains a `CsLinkRemoteSE` component for each remote simulator, which accumulates all incoming and outgoing events for a certain remote simulator, stores them in a queue, and exchanges them with its counterpart. Each `CsLinkRemoteSE` is directly connected to a `Channel` responsible for data exchange.

The basic synchronization algorithm, implemented in the `CsScheduler` component, conducts the `CsLinkRemoteSE` components to send and receive external events in proper time steps. Furthermore, the `CsScheduler` embeds the cosimulation interface into the simulation process, i.e., it invokes the `CsInterface` procedures at specific simulation times. All incoming events are dispatched according to their time stamps and destination blocks by the `EventScheduler` component. Therefore, it fetches incoming events from the `CsInterface` and propagates them to the corresponding `CsBlock` module. All components are designed to build a general framework, which is easily extensible and adaptable to interconnect distinct simulators. The simulation performance can be optimized by adjusting simulation-specific parameters of certain interface components.

5. Automatic Code Generation

One of the main tasks in this work is the automatic building of a cosimulation platform based on the proposed design methodology. The generation process of a verification platform consists of three main tasks: the cosimulation interface generation, the semiautomatic system partitioning, and the hierarchical code generation. The automatic cosimulation interface generator creates `CsBlock`s, `CsInterface`s, and `Channel`s, which are required to establish a connection between two `CsBlock`s. Figure 11.6 depicts the class diagram of the cosimulation interface generator.

As depicted, the central component in the cosimulation interface generator is `CsInterfacePairBuilder`, which is derived from the `SourceBuilder`

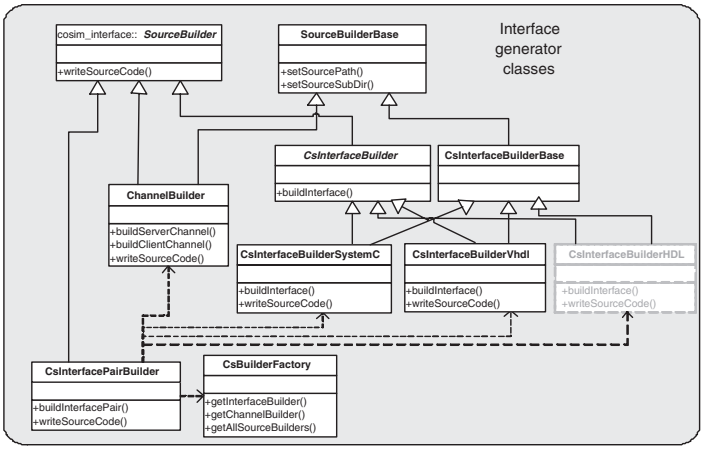


Fig. 11.6 Class diagram of the cosimulation interface generator

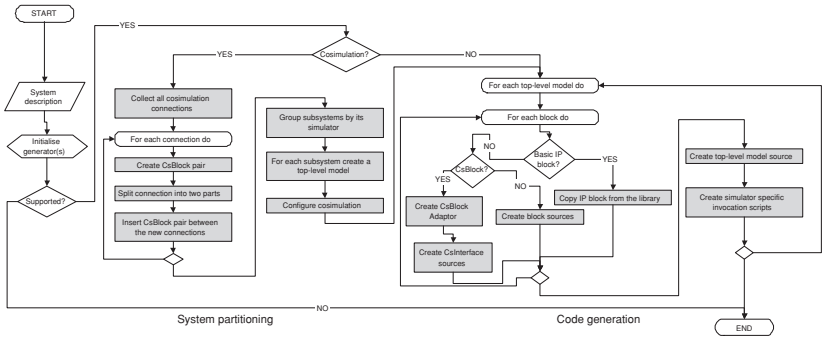


Fig. 11.7 Flow of the automatic code generator

class. Two provided methods enables simple creation of a `CsBlock` pair: `buildInterfacePair(BlockA, BlockB)` and `writeSourceCode()`. The `buildInterfacePair`-Method creates two `CsBlock`s using simulator-specific `CsInterfaceBuilder` classes. All cosimulation sources are generated by invoking the `writeSourceCode()` method. It is obvious that the cosimulation interface generator must have an abstract structure, which enables modular and generic implementation. The cosimulation interface generator serves for creating `CsBlock` pairs, `CsInterface` control structure (Cs-API) and required channels. It does not create the whole verification platform, due to hold the system is simple and clearly arranged. The cosimulation interface generator has been integrated in a more complex structure, which handles the code generation for the verification of the whole designed system. Figure 11.7 depicts the flow of the automatic code generation. The first step after getting the system description and the initialization phase in the code generator is to check the presence of a cosimulation.

In case of a cosimulation, the code generator executes a system partitioning that performs the creation of subsystems, which are later on clustered by simulator type. In this process, the cosimulation interface generator has been used to create required `CsBlock` pairs. At least the code generator invokes the source generation for each created top-level description. Recursively, all blocks are scanned and their source codes are generated. If a block is a basic IP block the source code will be taken from the provided IP library, otherwise they will be generated. The last step in the code generation flow is the creation of the simulator-specific invocation scripts. Each involved simulator uses specific call routines, which are defined in a simulator script.

Currently, the presented work provides automatic code generation for very high speed integral circuit (VHSIC) hardware description language (VHDL), VHDL analog and mixed signal (AMS), SystemC, Simulink, and SaberMAST models, whereby a cosimulation interface can be created between ADVanceMS, ModelSim, AMS Designer, Simulink, and SystemC simulators.

6. Experimental Example

An example of the code generation of a heterogeneous system using the methodology described in Section 3 is presented in this section. Figure 11.8 depicts a system overview of an automotive power management system (APMS). The system controls the power needs of the automotive electromechanical loads and the charging of the battery and prevents a complete discharging of the battery at any time. This system should work in the large signal area and should contain all important analog nonlinear effects (voltage, current), and the algorithms of the controller. The energy consumption of the car depends first on static loads, controlled by the driver (driving cycle), and at second from the behavior of the driver assistant systems (dynamic loads dependent on the environment like street condition, etc.). It contains analog and digital components as well as software components. The microcontroller developed in SystemC as a state-machine represents a software part of the system. All other components such as analog-to-digital/digital-to-analog converter (ADC/DAC), generator, speed-sensor, battery etc. are coded in VHDL-AMS.

6.1 Design Steps

Concurrently developed subsystems are integrated using the schematic editor integrated in our application framework. Figure 11.9.a depicts a schematic overview of the APMS system at the system design level.

Two D/A converters and two A/D converters build a bridge between the microcontroller and the analog part of the system. To overcome the problems of the integration of models written in different languages, four `CsBlock` pairs

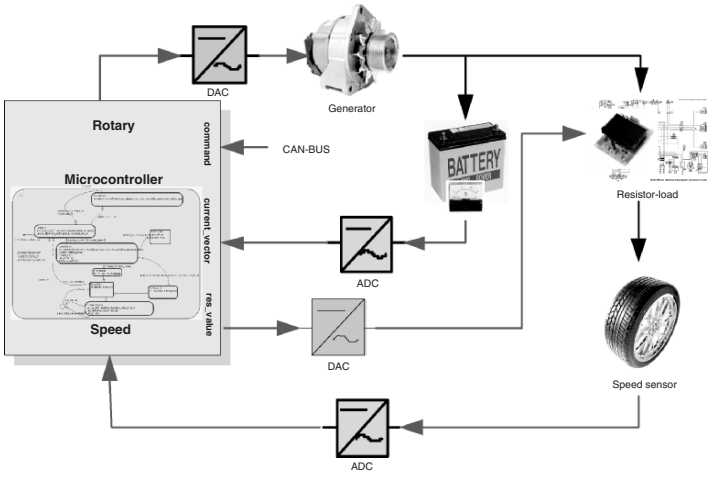
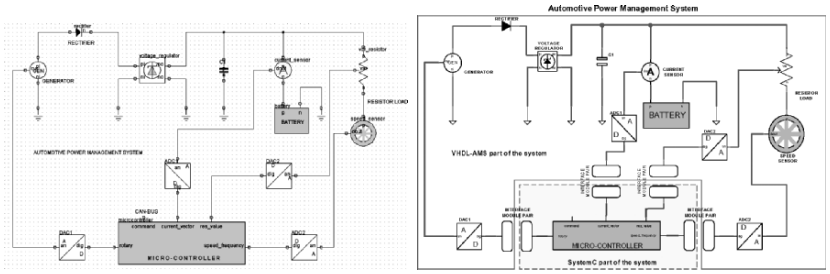


Fig. 11.8 A system overview of an automotive power management system



(a) System design level (b) Language level

Fig. 11.9 System overview

have been inserted between the microcontroller unit and four converters. With the insertion of `CsBlocks`, the APMS system has been subdivided into two language/simulator groups.

Next step is generating the top-level description of the generated language/simulator groups. In our example we have two language groups: SystemC and VHDL-AMS. The SystemC system consists of the microcontroller and four SystemC-interface modules. Respectively, the VHDL-AMS subsystem consists of VHDL-AMS components and four VHDL-AMS interface modules. Figure 11.9b depicts the system overview at the language level with inserted interface modules. The A/D and D/A converters use standard logic vectors for communicating with the microcontroller unit. The VHDL standard logic vectors (`std_logic_vector`) are converted into logic vectors (`sc_lv`) on the SystemC side. Only one peer-to-peer connection channel between

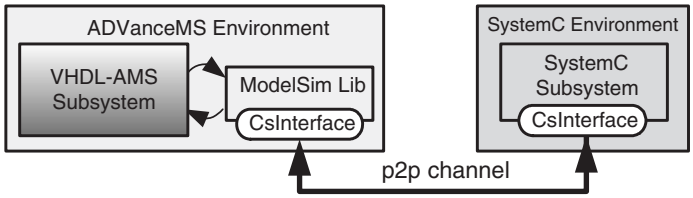


Fig. 11.10 Configuration of the Cosimulation Platform

SystemC and VHDL-AMS simulation has been created, which is used to transfer the data of all interface modules. We use ADVanceMS simulator from MentorGraphics to simulate the VHDL-AMS subsystem. Since the current version of the ADVanceMS simulator does not provide the required FLI functions for the used synchronization method, we used the ModelSim interface of ADVanceMS to get access to FLI and through FLI to SystemC. The configuration of the cosimulation platform is depicted in Figure 11.10.

This configuration enables cosimulation of analog components with software components of the system. The interface module on the VHDL-AMS side was simulated using ModelSim library. Its entity definition is described in VHDL and the functional behavior of the interface module in C++. The remaining part of the VHDL-AMS side was simulated using ADVanceMS.

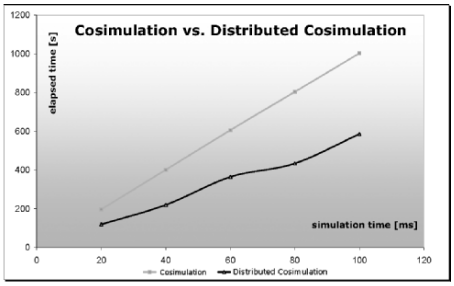
6.2 Results

The proposed cosimulation technique has been evaluated by used heterogeneous examples. Figure 11.11a depicts the comparison between performed cosimulations and its chart has been depicted on Figure 11.11b.

A cosimulation, which is distributed by TCP/IP using two workstations, has been compared with a cosimulation, whereby both simulators run on the same workstation. The best results have been retrieved by a distributed cosimulation allowing parallel execution of independent simulation tasks. The table on Figure 11.11a shows the comparison of elapsed times for specific simulation time. At least the average speed-up has been calculated, which is recently dependent on many different factors. The applied synchronization method and `CsInterface` control structures have been improved by the results retrieved from the performed cosimulation, which are compared with the results from a simulation of the same system. To improve the implemented cosimulation approach, the multipoint control unit (MCU) model has been developed in SystemC as well as in VHDL language. The computed data from both simulations are identical, which verifies the applied cosimulation method and used synchronization algorithm (Figure 11.12). The computed signals of a cosimulation are present on the upper side in the figure and the simulation results of a

No.	Simulation Time [ms]	Cosimulation Elapsed Time [s]	Distributed Cosim. Elapsed Time [s]
1	20	197	119
2	40	401	220
3	60	606	364
4	80	804	434
5	100	1,003	586
Dist. Speed-Up:		1	1.75
Cosimulation: ADVanceMS and SystemC			

(a) Measured data



(b) Chart

Fig. 11.11 Cosimulation results

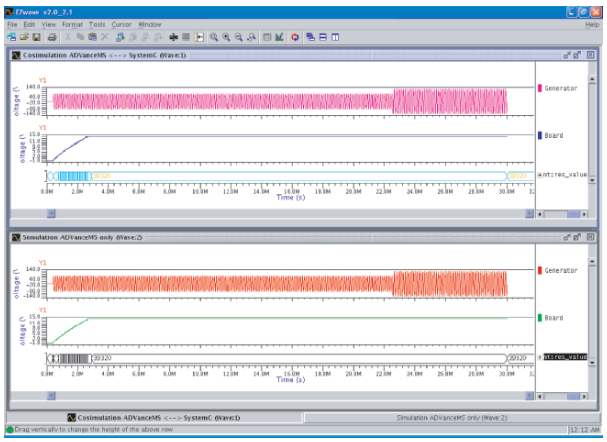


Fig. 11.12 Signal comparison: generator and board voltage in simulation and cosimulation

cosimulation are present on the lower side of the waveform viewer. It is obvious that the homogeneous simulation in this case must be faster than performed cosimulation, due to the delays of the communication and the used synchronization method.

However, the elapsed simulation time depends strongly on the structure of the simulated system, involved simulators, and many other factors. The primary goal of this work was not accelerating the simulation but more to provide the possibility to design systems that are language-independent and automatically generate the required heterogeneous verification platform, which ensures cycle accurate cosimulation. Furthermore, the proposed cosimulation method supports building distributed and parallel simulation environment, to increase the simulation time. The ADVanceMS–SystemC cosimulation example has been advised selected here to present the actually attractive and recently required combination, which integrates analog, digital, as well as software

components in one environment using today's most popular simulators. The proposed design methodology supports multi-HDL system design based on models provided by an IP library. Moreover, the verification platform has been generated fully automatically. The implemented automatic code generator reduces the system design time, which is an important issue.

7. Conclusion

The proposed design methodology using cosimulation techniques has been applied on an application framework, which generates the required verification platform fully automatically. Multi-HDL system design, generic interfacing, as well as automatic generation of a verification platform including generation of cosimulation interfaces are the key benefits of this work. A generic and modular structured code generator has been implemented. The cosimulation method as well as the code generator itself have been evaluated by an example taken from the automotive industry.

References

Brooks, Christopher, Lee, Edward A., Liu, Xiaojun, Neuendorffer, Steve, Zhao, Yang, and Zheng, Haiyang (eds) (2004) *Heterogeneous Concurrent Modeling and Design in Java*, volume 1, 2, and 3. EECS, University of California, Berkeley, CA. Memoranda UCB/ERL M04/27, M04/16 and M04/17.

Calinescu, Radu (1995) Conservative discrete event simulations on bulk synchronous parallel architectures. Technical Report PRG-TR-16-95, Computing Laboratory, Oxford University, Oxford.

Copstein, Bernardo, Wagner, Flávio Rech, and Pereira, Carlos Eduardo (1997) SIMOO: an environment for the object-oriented discrete simulation. In: *9th European Simulation Symposium (ESS'97)*, Passau, Germany.

de Mello, Braulio Adriano and Wagner, Flávio Rech (2002) A standardized cosimulation backplane. In: Robert, M., Rouzeyre, B., Piguet, C., and Flottes, M.-L. (eds) *SoC Design Methodologies*, volume 90 of *IFIP International Federation for Information Processing*, pages 181–192. Kluwer Academic Publishers, Norwell, MA.

Mentor Graphics (2005) *Seamless*. Mentor Graphics, Wilsonville, OR. Online Documentation, `http://www.mentor.com/`.

Mittra, Swapnajit (1999). *Principles of Verilog PLI*. Kluwer Academic Publishers, Norwell, MA.

Model Technology (2001) *Modelsim 5.5f: Foreign Language Interface*. Model Technology, Portland, OR.

The MathWorks (2005) *ModelSim Online Documentation*. The MathWorks, Natick, MA. `http://www.mathworks.com/`.

TNI-Software (2005) *CosiMate*. TNI-Software, Brest Cedex, France. Online Documentation, `http://www.tni-world.com/`.

Chapter 12

UML/XML-Based Approach to Hierarchical AMS Synthesis

Ian O'Connor, Faress Tissafi-Drissi, Guillaume Révy, and Frédéric Gaffiot
Laboratory of Electronics, Optoelectronics, and Microsystems
Ecole Centrale de Lyon
36 avenue Guy de Collongue
F-69134 Ecully, France

Abstract This chapter explores the suitability of unified modeling language (UML) techniques for defining hierarchical relationships in analogue and mixed signal (AMS) circuit blocks, and extensible markup language (XML) for storing soft AMS intellectual property (IP) design rules and firm AMS IP design data. Both aspects are essential to raising the abstraction level in synthesis of this class of block in SoCs. The various facets of AMS IP are discussed, and explicit mappings to concepts in UML are demonstrated. Then, through a simple example block, these concepts are applied and the successful modification of an existing analogue synthesis tool to incorporate these ideas is proven. The central data format of this tool is XML, and several examples are given showing how this metalanguage can be used in both AMS soft-IP creation and firm-IP synthesis.

Keywords: analogue and mixed-signal; synthesis; IP; UML; XML.

1. Introduction

Cost, volume, power, and pervasivity are all difficult constraints to manage in the design of new integrated systems (smart wireless sensor networks, ubiquitous computing, etc.). Along with increasingly complex functionality and human–machine interfaces, they are driving the semiconductor industry towards the ultimate integration of complete, physically heterogeneous systems on chip (SoCs). The coexistence of sensors, analogue/mixed, and radio

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 205–225.

frequency RF systems (multi-physics part commonly called AMS[1]) with digital and software IP[2] blocks cause significant design problems.

The difficulty centres on the concept of abstraction levels. To deal with increasing complexity (in terms of number of transistors), SoC[3] design requires higher abstraction levels. But at the same time, valid abstraction is becoming increasingly difficult due to physical phenomena becoming first order or even dominant at nanometric technology nodes. The rise in analogue, mixed-signal, RF[4], and heterogeneous content to address future application requirements compounds this problem. Efficient ways must be found to incorporate non-digital objects into SoC design flows in order to ultimately achieve AMS/digital hardware/software co-synthesis.

The main objective of such an evolution is to reduce the design time in order to meet the time to market constraints. It is widely recognised that for complex systems at advanced technology nodes, mere scaling of existing design technology will not contribute to reducing the "design productivity gap" between the technological capacity of semiconductor manufacturers (measured by the number of available transistors) and the design capacity (measured by the efficient use of the available transistors). Since 1985, production capacity has increased annually by between +41% and +59%, while design capacity increases annually by a rate of only +20% to +25%. The 2003 ITRS Roadmap clearly states that "cost [of design] is the greatest threat to continuation of the semiconductor roadmap". Only the introduction of new design technology (such as, historically, block reuse or IC implementation tools: each new technology has allowed design capacity to "jump" and to catch up with production capacity) can enable the semiconductor industry to control design cost. Without design technology advances, design cost becomes prohibitive and leads to weak integration of high added value devices (such as RF circuits). One of the next advances required by the electronic design automation (EDA) industry is a radical evolution in design tools and methods to allow designers to manage the integration of heterogeneous AMS content.

2. AMS IP Element Requirements for Synthesis Tools

Most analogue and RF circuits are still designed manually today, resulting in long design cycles and increasingly apparent bottlenecks in the overall design process (Gielen and Dehaene, 2005). This explains the growing awareness in industry that the advent of AMS synthesis and optimisation tools is a necessary step to increase design productivity by assisting or even automating the

[1] Analogue and mixed-signal
[2] Intellectual property
[3] System-on-chip
[4] Radio frequency

AMS design process. The fundamental goal of AMS synthesis is to quickly generate a first-time correct-sized circuit schematic from a set of circuit specifications. This is critical since the AMS design problem is typically underconstrained with many degrees of freedom and with many interdependent (and often-conflicting) performance requirements to be taken into account.

Synthesisable (soft) AMS IP is a recent concept (Hamour et al., 2003) extending the concept of digital and software IP to the analogue domain. It is difficult to achieve because the IP hardening process (moving from a technology-independent, structure-independent specification to a qualified layout of an AMS block) relies to a large extent on the quality of the tools being used to do this. It is our belief that a clear definition of AMS IP is an inevitable requirement to provide a route to system-level synthesis incorporating AMS components.

Table 12.1 summarises the main facets necessary to AMS IP. For the sake of clarity, a reference to very high speed integrated circuit (VHSIC) hardware description language (VHDL)-AMS concepts is shown wherever possible.

Figure 12.1 shows how these various facets of AMS IP should be brought together in an iterative single-level synthesis loop. Firstly, the performance criteria are used as specifications to quantify how the IP block should carry out the defined function. Performance criteria for an amplifier, for example, will include gain, bandwidth, power supply rejection ratio (PSRR), offset, etc. They can be considered to be the equivalent of generics in VHDL-AMS. They have two distinct roles, related to the state of the IP block in the design process:

1. As block parameters when the IP block is a component of a larger block, higher up in the hierarchy, in the process of being designed;

2. As specifications when the IP block is the block in the process of being designed (such as here); This role cannot be expressed with VHDL-AMS generics, although language extensions (Doboli and Vemuri, 2003; Hervé and Fakhfakh, 2004) have been proposed

It will be shown in Section 3 that this dual role requires the definition of a new data type.

The comparison between specified and real performance criteria values act as inputs to the synthesis method, which describes the route to determine design variable values. It is possible to achieve this in two main ways:

1. Through a direct procedure definition, if the design problem has sufficient constraints to enable the definition of an explicit solution

2. Through an iterative optimisation algorithm; if the optimisation process cannot, as is usually the case, be described directly in the language used to describe the IP block, then a communication model must be set up

Table 12.1 AMS IP block facets

Property	*Short description*	*VHDL-AMS equivalent*
Function definition	Class of functions to which the IP block belongs	**entity**, behavioural **architecture**
Performance criteria	Quantities necessary to specify and to evaluate the IP block	**generic**
Terminals	Input/output links to which other IP blocks can connect	**terminal**
Structure	Internal component-based structure of the IP block	structural **architecture**
Design variables	List of independent design variables to be used by a design method or optimisation algorithm	subset of **generic map**
Physical parameters	List of physical parameters associated with the internal components	**generic map**
Evaluation method	Code defining how to evaluate the IP block, i.e., transform physical parameter values to performance criteria values. Can be equation- or simulation-based (the latter requires a parameter extraction method)	(partly) **process** or **procedure**
Parameter extraction method	Code defining how to extract performance criteria values from simulation results (simulation-based evaluation methods only)	
Synthesis method	Code defining how to synthesise the IP block, i.e., transform performance criteria requirements to design variable values. Can be procedure- or optimisation-based	
Constraint distribution method	Code defining how to transform IP block parameters to specifications at a lower hierarchical level	

between the optimiser and the evaluation method; a direct communication model gives complete control to the optimisation process, while an inverse communication model uses an external process to control data flow and synchronisation between optimisation and evaluation; the latter model is less efficient but makes it easier to retain tight control over the synthesis process

The synthesis method generates combinations of design variables as exploratory points in the design space. The number of design variables defines

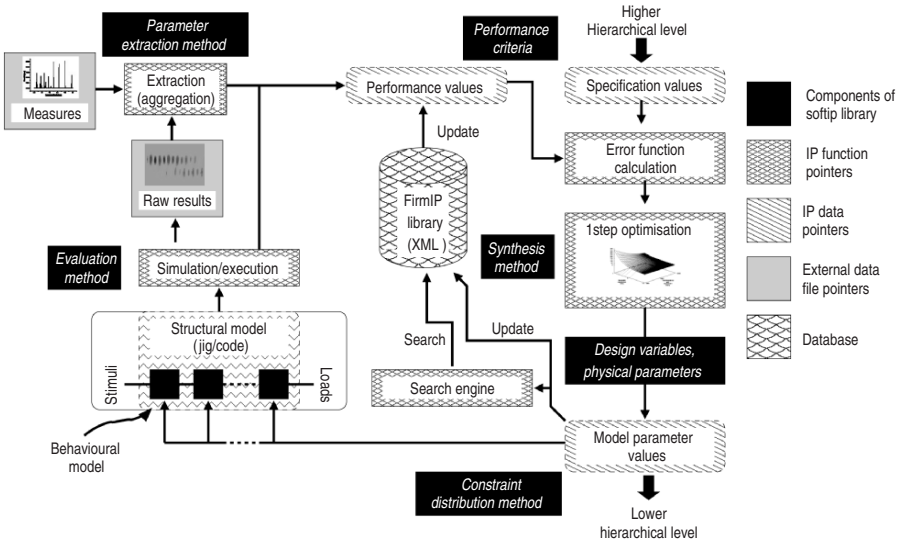


Fig. 12.1 AMS synthesis loop showing AMS IP facet use

the number of dimensions of the design space. The design variables must be independent of each other and as such represent a subset of IP block parameters (i.e., performance criteria, described earlier) in a structure definition. For example, a differential amplifier design variable subset could be reduced to a single gate length, bias voltage, and three transistor widths for the current source, matched amplifier transistors, and matched current mirror transistors. Physical variables are directly related to design variables and serve to parameterise all components in the structure definition during the IP block evaluation process. These are represented by the generic map definitions in structural architecture component instantiations in VHDL-AMS. In the above example, the design variable subset would be expanded to explicitly define all component parameters.

The evaluation method describes the route from physical variable values to the performance criteria values and completes the iterative single-level optimisation loop. Evaluation can be achieved in two main ways:

1. Through direct code evaluation, such as for active surface area calculations

2. Through simulation (including behavioural simulation) for accurate performance evaluation (gain, bandwidth, distortion, etc.). If the IP block is not described in a modelling language that can be understood by a simulator, then this requires a gateway to a specific simulator and to

a jig corresponding to the IP block itself. For the simulator, this requires definition of how the simulation process will be controlled (part of the aforementioned communication model). For the jig, this requires transmission of physical variables as parameters and extraction of performance criteria from the simulator-specific results file. The latter describes the role of the parameter extraction method, which is necessary to define how the design process moves up the hierarchical levels during bottom-up verification phases.

Once the single-level loop has converged, the constraint distribution method defines how the design process moves down the hierarchical levels during top-down design phases. At the end of the synthesis process at a given hierarchical level, an IP block will be defined by a set of physical variable values, some of which are parameters of an IP sub-block. To continue the design process, the IP sub-block will become an IP block to be designed and it is necessary to transform the block parameters into specifications. This requires a definition of how each specification will contribute to an error function for the synthesis method and includes information additional to the parameter value (weighting values, specification type: constraint, cost, condition, etc.).

3. UML in AMS Design

3.1 Reasons for Using UML in Analogue Synthesis

UML[5] is a graphical language enabling the expression of system requirements, architecture, and design, and is mainly used in industry for software and high-level system modelling. UML 2.0 was adopted as a standard by OMG[6] in 2005. The use of UML for high-level SoC design, in general, appears possible and is starting to generate interest in several research groups (Riccobene et al., 2005). A recent proposal (Carr et al., 2004) demonstrated the feasibility of describing AMS blocks in UML and then translating them to VHDL-AMS, building on other approaches to use a generic description to target various design languages (Chaudhary et al., 2004). This constitutes a first step towards raising abstraction levels of evaluatable AMS blocks. Considerable effort is also being put into the development of "AMS-aware" object-oriented (OO) design languages such as SystemC-AMS (Vachoux et al., 2003) and SysML (Vanderperren and Dehaene, 2005). However, further work must be carried out to enable the satisfactory partitioning of system-level constraints among the digital, software, and AMS components. At the system level, the objective in SoC design is to map top-level performance specifications among the different blocks in

[5]Unified modeling language
[6]Object Management Group

the system architecture in an optimal top-down approach. This is traditionally done by hand in an ad hoc manner. System-level synthesis tools are lacking in this respect and must find ways of accelerating the process by making reasonable architectural choices about the structure to be designed and by accurately predicting analogue/RF architectural specification values for block-level synthesis.

Therefore, to be compatible with SoC design flows, top-down synthesis functionality needs to be added to AMS blocks. Our objective in this work is to demonstrate that this is possible. Since UML is a strong standard on which many languages are based (SysML is directly derived from UML, and SystemC as an OO language can be represented in UML also), it should be possible to map the work to these derived or related languages.

3.2 Mapping AMS IP Requirements to UML Concepts

In order to develop a UML-based approach to hierarchical AMS synthesis, it is necessary to map the AMS IP element requirements given in Section 3.1 to UML concepts.

UML has many types of diagrams and many concepts that can be expressed in each—many more, in fact, than are actually needed for the specific AMS IP problem. Concerning the types of diagram, two broad categories are available:

1. Structural diagram, to express the static relationship between the building blocks of the system. We used a class diagram to describe the properties of the AMS IP blocks and the intrinsic relations between them. The tenets of this approach and how to generate UML-based synthesisable AMS IP will be described in this section, with an example in Section 5.

2. Behavioural diagram, showing the evolution of the system over time through response to requests, or through interaction between the system components. We used an activity diagram to describe the AMS synthesis process. This will be described in further detail in Section 3.3 and extensions to an existing AMS synthesis tool to incorporate these concepts will be shown in Section 4.

Class relationships. Firstly, it is necessary to establish a clear separation of a single function definition (entity and functional behavioural model for top-down flows) from *n* related structural models (for single-level optimisation and bottom-up verification). Each structural model will contain lower-level components, which should be described by another function definition. It is also necessary to establish functionality and requirements common to all structural models whatever their functions be. By representing all this in a single diagram (Figure 12.2), we are in fact modelling a library of system components,

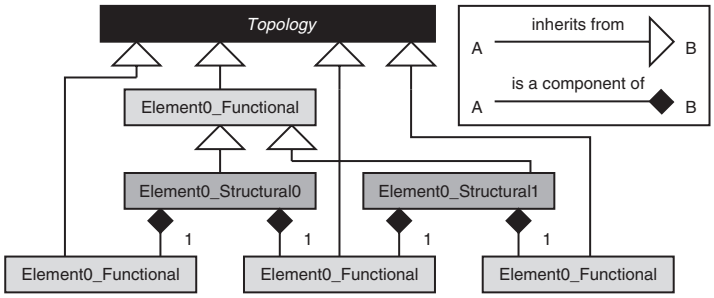


Fig. 12.2 UML representation of AMS IP hierarchical dependencies

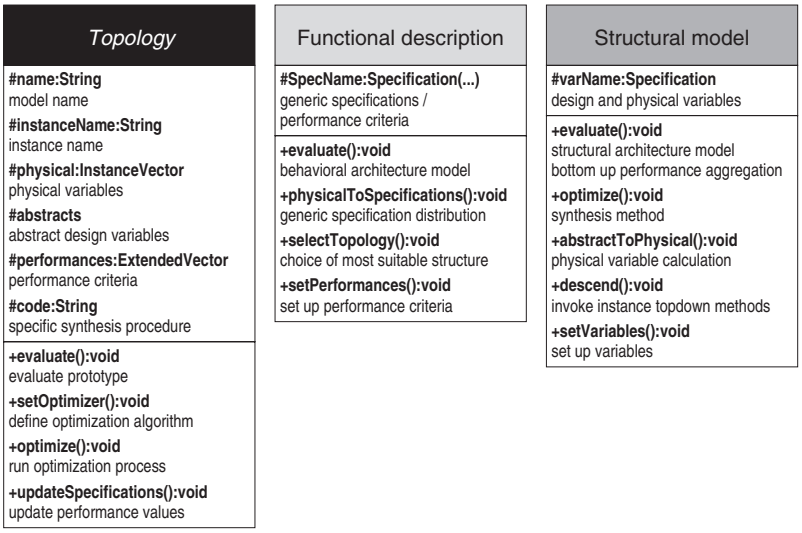


Fig. 12.3 UML class definitions for AMS IP blocks

not the actual system to be designed itself. This can be done using an object diagram—however, in this work we will focus on the broader class diagram.

A class diagram constitutes a static representation of the system. It allows the definition of classes among several fundamental types, the class attributes and operations, and the time-invariant relationships between the various classes. From the above analysis, we require (cf. Figure 12.3):

1. A single, non-instantiable (abstract) class representing common functionality and requirements, in a separate publicly accessible package. We called this class `Topology`.

2. A single class representing the function definition, which inherits from `Topology`. An alternative solution would be to separate "evaluatable"

functionality and "synthesisable" functionality through the use of interfaces. This is certainly a debatable point, but our view is that it would tend to overcomplicate the description process. Another point is that one can also be tempted to separate the entity aspect from the behavioural model aspect, which would then allow the entity class to become abstract. Again, this also appears to be somewhat overcomplicated to carry out.

3. The *n* classes representing the structural models, which all inherit from the function definition class. Each structural variant is composed of a number of components at a lower hierarchical level, represented by a single function definition class for each component with different functionality. As the structural variant cannot exist if the component classes do not exist, this composition relationship is strengthened to an aggregation relationship.

AMS IP requirement handling through definition of class attributes and methods. Having established how to separate particular functionality between common, functional, and structural parts of an AMS hierarchical model, it is now necessary to define how to include each facet of the AMS IP requirements set out in Section 2. This is summarised in Table 12.2.

Thus the performance criteria and variables are defined with the type `Specification`. This is a specific data type, which plays an important role in the definition of AMS IP. It requires a name `String`, default value, and units `String` as minimum information. When used as a performance requirement in a base class, it can also take on the usual specification definitions (<, >, =, `minimize`, `maximize`).

3.3 Modelling Analogue Synthesis with Activity Diagrams

In UML, a behavioural diagram complements structural diagrams by showing how objects or classes interact with each other and evolve over time to achieve the desired functionality. Among these, the activity diagram is useful for showing the flow of behaviour (objects, data, control) across multiple classes as a sort of sophisticated data flow diagram.

Figure 12.4 shows an example flow for two hierarchical levels. For a given hierarchical level, the process begins with specification definition—either from an external point (e.g., user) or from the design process at the hierarchical level immediately above. It then calls a number of internal methods (set performances, variables, and abstracts), all of which must have been explicitly defined by the IP creator prior to synthesis. The optimisation process can then begin with dimension (or variable) value modification according to the optimisation algorithm used, determination of the physical parameter values from

Table 12.2 Mapping of AMS IP requirements to class structure

Property	*Class*	*Attribute type*	*Method*	*Access*
Function definition			constructor	public
entity name	Functional	String		private
behavioural architecture	Functional		evaluate()	public
Performance criteria	Functional	Specification		protected
			setPerformances()	public
Terminals	Functional	DomainNode		protected
Structure				
structural architecture name	Structural	String		private
Design variables	Structural	Specification		protected
			setVariables()	public
Physical parameters	Structural	Specification		protected
Evaluation method	Structural		evaluate()	public
Parameter extraction method				
Synthesis method	Structural		optimize()	public
			abstractToPhysical()	public
Constraint	Structural		descend()	public
distribution method	Functional		physicalToSpecifications()	public

the new design variable set, and evaluation and comparison of achieved performance values with requirements. If the requirements are met, then the process can go down through the hierarchy to determine the parameters of lower hierarchical blocks, or if there are no lower levels then the verification process can begin. It should be noted that the sequence of events for a functional/structural model pair maps to the iterative loop shown in Figure 12.1.

4. Extensions to Existing Analogue Synthesis Tool (runeII)

We have incorporated these concepts into an existing in-house AMS synthesis framework, runeII. This builds on a previously published version of the tool (Tissafi-Drissi et al., 2004). The main motivation behind this evolution was to improve the underlying AMS IP representation mechanisms and to enhance

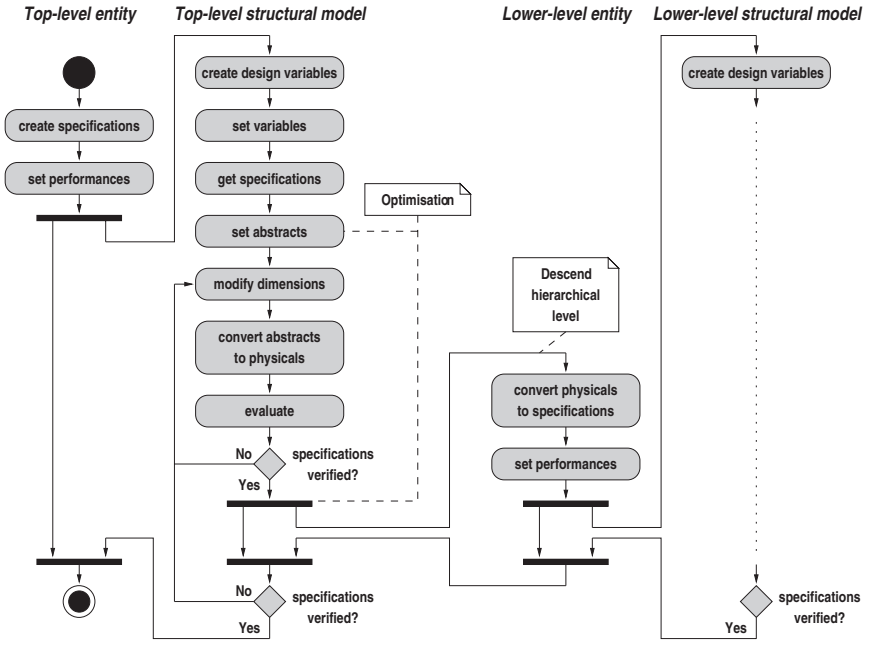


Fig. 12.4 Activity diagram for TIA block synthesis process

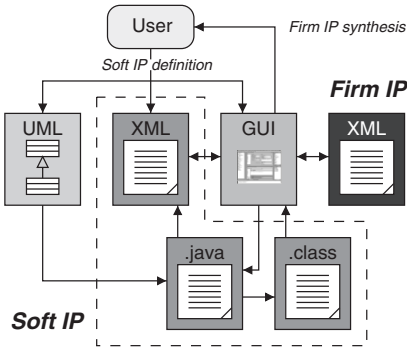


Fig. 12.5 UML/XML use flow in runeII

the input capability of the tool. A schematic showing the various inputs and data files is given in Figure 12.5.

From the user's point of view, there are two main phases to AMS synthesis: (i) AMS soft-IP definition, which can be done through UML, XML,[7] or

[7] eXtensible markup language

through a specific GUI[8]; and (ii) AMS firm-IP synthesis, which can be run from the GUI or from scenarios. XML is a text markup language for interchange of structured data specified by W3C. The Unicode Standard is the reference character set for XML content and, because of this portable format and ease of use, it is fast becoming a de facto standard for text-based IP exchange.

4.1 AMS Soft-IP Definition

The aim of the first point is to create executable and synthesisable models (here, in the form of Java `.class` files). We consider the central, portable format to be XML, which can be generated directly from the GUI and from `.java` source files.

A screenshot of the GUI enabling the creation of such files from graphical format is shown in Figure 12.6. The various zones in the figure have been numbered and the corresponding explanation is given as follows:

1. Menu bar
2. Database tree explorer (top nodes = entity/functional models; nested nodes = structural models); the user is able to process several actions: new, open, export, import, delete, rename, cut, copy, paste; these actions operate on the currently selected structure or function and are also available in the main frame toolbar
3. Entity/functional model editor; here, the IP creation process starts in earnest in defining the various performance criteria
4. Structural model editor; this window allows the creation of design variables, physical parameters, evaluation procedures, etc.
5. Preset design plans (sequences of optimisation algorithms)
6. Technology data files
7. Message window; this is to output log information, e.g., detailed information about the operation that is being made, error descriptions, etc.

4.2 AMS Firm-IP Synthesis

The second point exploits the created executable, synthesisable models in an iterative process aiming to determine the numerical parameter values necessary to optimally realise the numerical performance requirements. Again, the database format was chosen as XML for reasons of portability. Here, apart

[8]Graphical user interface

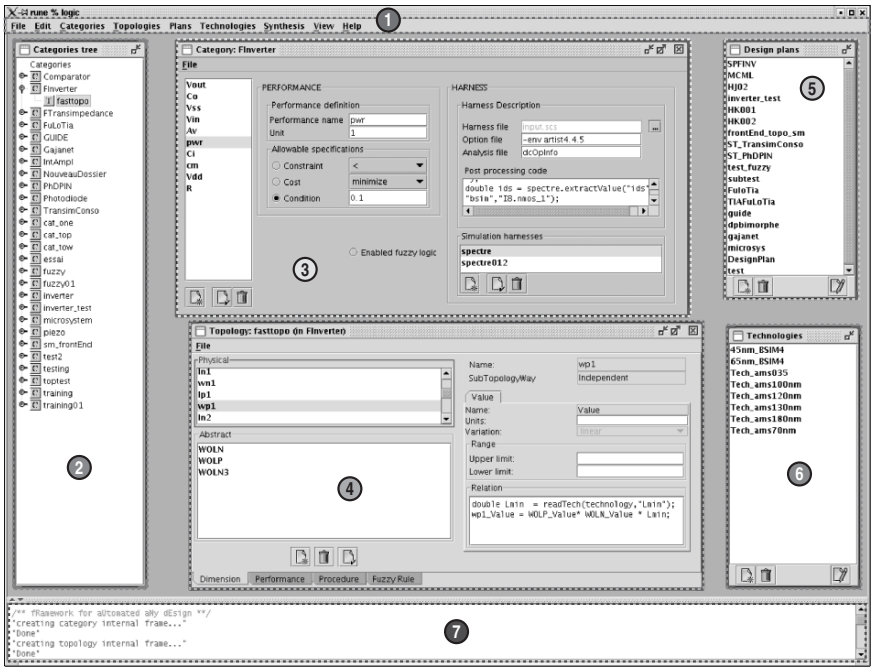


Fig. 12.6 Screenshot of the runeII GUI

Listing 12.1 Entity/functional and structural model DTD template

```
<!{ELEMENT} FunctionName (Structure1, Structure2*)>
  <!{ATTLIST} FunctionName
    PerformanceName1 CDATA ""
    ...
  >

<!{ELEMENT} StructureName (Component1, Component2, ...)>
  <!{ATTLIST} StructureName
    VariableName1 CDATA ""
    ...
  >
```

from capture of the numerical information itself in an XML document, a definition of the legal building blocks necessary to the interpretation of the XML document structure is required. This is the purpose of a DTD,[9] which can be declared inline in the XML document or as an external reference. We have chosen the latter approach, which is shown in Listing 12.1.

[9] Document type definitions

As mentioned previously, the synthesis process can be run either from the GUI or through the creation of scenarios. Scenarios are another type of class that instantiate and set-up all the components necessary for synthesis in their constructor, much as in a traditional netlist, and then define the optimisation process in the main method. The scenario actually represents the final executable and, while more difficult to generate, avoids any constrictions imposed by the GUI.

5. Example

We now introduce an example circuit to illustrate the concepts previously described. We focus on the representation of a resistive feedback TIA[10] (consisting of a non-differential inverting amplifier with feedback resistance) as part of a configuration memory operating system (CMOS) photoreceiver front-end (Figure 12.7; Tissafi-Drissi et al., 2003).

It is important to understand how a TIA is specified in the link. The main performance criteria for the TIA itself are the in-band transimpedance gain Z_{g0}, angular resonant frequency ω_0, quality factor Q, quiescent power dissipation, and occupied surface area. The first three quantities express the capacity of the TIA to convert an input photocurrent variation to an output voltage variation according to a linear second-order transfer function. The latter two criteria (power and area) can only be accurately determined by synthesising down to transistor level, constituting the main difficulty in AMS IP formulation. To

[10]Transimpedance Amplifier

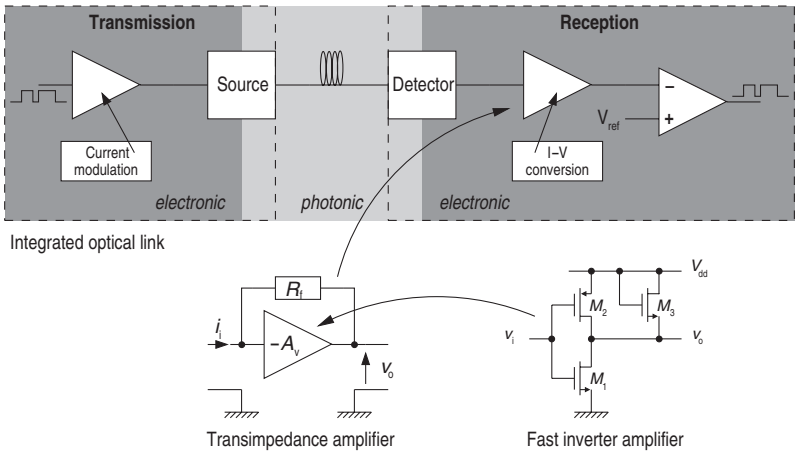


Fig. 12.7 TIA and amplifier in an integrated optical link

reach this level, the specific TIA structure (resistive feedback) is considered. The physical parameters consist of the feedback resistance value R_f and the internal amplifier performance criteria (voltage gain A_v, output resistance R_o). Concerning the design variables, it has been shown in O'Connor et al. (2003) that only one is necessary: M_f, the ratio between R_f and R_o.

5.1 Class Diagram Example

This information suffices to start building a class diagram for the TIA structure (Figure 12.8).

For clarity, only the `TransimpedanceAmplifier` functional model class, defining the TIA performance criteria, and its derived `RFeedback` structural model class, defining the physical and design variables, have been expanded. It should be noted that the physical variables related to the internal amplifier are defined in an aggregation relationship between the `RFeedback` class and the `Amplifier` functional model class. The other classes show their context in a class diagram representing an optical receiver circuit hierarchy. Some of

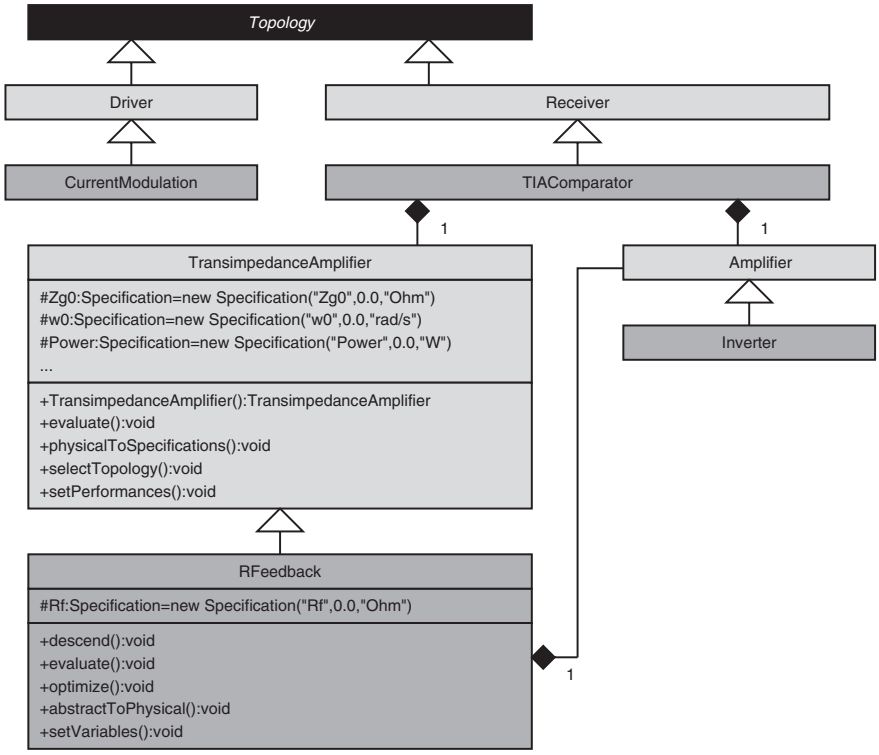


Fig. 12.8 TIA and resistive feedback classes in UML

the class methods shown are related to the specific implementation. In particular, some constructors require XML document inputs. These represent firm-IP data allowing the block to retrieve previously stored information in the format described in Section 4.2.

5.2 Soft-IP XML File Example

An example of an XML file representing (i) an entity/functional model is given in Listing 12.2, and (ii) a structural model is given in Listing 12.3. Both are based on specific DTD rules corresponding to the concepts set out in Section 4 and illustrate the various facets of AMS IP defined in Section 2.

5.3 Optimisation Scenario Example

As a simple example, Listing 12.4 shows the scenario (Java source) to optimise the `RFeedback` object.

5.4 Firm-IP XML Output File Example

The partial results, in the output XML format, of this synthesis process achieved for a 0.35 μm CMOS technology and with specifications given in the first line of the file are shown in Listing 12.5.

6. Conclusion

In this chapter, we have proved the feasibility of the use of UML for the representation of synthesisable hierarchical AMS IP blocks. A parallel between UML concepts and widely used concepts in AMS behavioural modelling languages (we used the VHDL-AMS example) was established, in particular:

- Class diagrams to represent the various ways (structural architectures) of realising a given function (entity and behavioural architectures)
- Inheritance relations to identify the relationship between an entity/behavioural model (base class) and one or more structural architectures (derived classes)
- Aggregation relations to identify the subcomponents in a structural architecture

We have successfully used these concepts to build class diagrams for a variety of AMS soft-IP blocks. Although the approach is quite straightforward, the resulting diagrams can be quite large and unwieldy. Further work is necessary to determine how to make better use of package diagrams in soft-IP library management.

Listing 12.2 Entity/functional model description output in XML

```
<category name="TransimpedanceAmplifier"> - <template ↵
    name="Zg0"
units="Ohm">
    <definitions constraint=">" cost="maximize" ↵
        condition="0.1"/>
  - <harness simulator="spectre" file="input.scs" ↵
      options="-env␣artist4.4.5" analysis="ac" selected="true">
    - <code>
        Zg0 = spectre.gainMax("ID:p","vo");
      </code>
    </harness>
  + <harness simulator="eldo" file="input.cir" options="" ↵
      analysis="ac" selected="false">
    </harness>
  </template>
+ <template name="QuiescentPower" units="W"></template>
  ...
```

Listing 12.3 Structural model description output in XML

```
- <topology name="TransimpedanceAmplifier-RFeedback"
instanceName="" categoryName="TransimpedanceAmplifier">
  + <physical type="dependent" name="Amplifier" ↵
      instanceName="A1" categoryName="Amplifier">
    </physical>
  + <physical type="independent" name="Resistance" ↵
      instanceName="Rf"></physical>
  - <abstract type="independent" name="Double" ↵
      instanceName="Mf">
    - <dimension name="Value" units="" lower="0.0010" ↵
        upper="100.0" variation="linear">
      </dimension>
    </abstract>
    ...
  - <performance name="Zg0" units="Ohm" heuristic="false" ↵
      enabled="false">
    - <equation>
        Zg0  =  ((Rf_Value * A1.Av())- A1.Ro())/(1 + A1.Av());
      </equation>
    </performance>
  + <performance name="QuiescentPower" units="W" ↵
      heuristic="false" enabled="false"></performance>
    ...
  </topology>
```

Listing 12.4 TransimpedanceAmplifier/RFeedback optimisation scenario description in Java

```
1  package scenarios;
2
3  import basic.*; ...
4
5  public class S_RFeedback extends TestTIA {
6    public S_RFeedback()
7    {
8      try {
9        // load specifications
10       Document TIADoc = ReadXML.loadDocument( ↵
             "/home/work/xmlFiles/TIA_specs.xml", true);
11       // create RFeedback object with specifications.
12       // Sizing is done in the constructor.
13       // Assign it to tia object
14       // (defined in TestTIA base class)
15       tia = new RFeedback("Rf",TIADoc);
16     } catch (Exception e) { e.printStackTrace(System.err); }
17   } // end constructor
18
19   public static void main(String[] args)
20   {
21     try {
22       // create scenario object.
23       // Design process defined and executed in constructor.
24       S_RFeedback scenario = new S_RFeedback();
25       // evaluation of resulting RFeedback object
26       scenario.getTIA().evaluate();
27       // store results in firm IP database
28       Document outputDocTIA = new Document( ↵
             WriteXML.XMLTopology(scenario.getTIA()));
29       WriteXML.save( ↵
             "/home/work/xmlFiles/outxml/S_TIA_perfs.xml", ↵
             outputDocTIA);
30     } catch (Exception e) { e.printStackTrace(System.err); }
31   } // end main
32 } // end S_RFeedback
```

Listing 12.5 Firm-IP synthesis results in XML

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE GenericLink
SYSTEM "Link_dtd.dtd"> <GenericLink BER="< 1.5E-18 Bit/s"
DataRate="> 1.5e9 bit/s" Abstract="True" toOptimize="True"> -
<OPPLink>
  + <Driver BiasCurrent="= 2e-3 A" ModulationCurrent="= 25e-6 A" Abstract="False" toOptimize="True">
    </Driver>
  + <WaveguideStructure Loss="2e-2 U" Length="2e-3 U"/>
  - <Receiver>
    + <Detector extinctionRatio="1.0 U" currentNoise="1.0 U" Responsivity="0.8 A/W" />
    - <TIAComparator>
        ...
        <TransimpedanceAmplifier Cd="= 400.0E-13 F" Cl="= 150E-13 F" Zg0="= 1E3 Ohm" Q="= 0.7017" Abstract="True" toOptimize="True">
          <RFeedback Rf ="1390 Ohm">
            <Amplifier Av="10" Ro="500 Ohm" Cm="8.0 E-14 F" Co="5.0 E-13 F" Ci="7.0E-13 F" QuiescentPower="0.5E-3 W" Abstract="True"/>
          </RFeedback>
        </TransimpedanceAmplifier>
        ...
      + <Comparator BW="= 3 GHz" QuiescentPower="= 164E-6" Latence="=" refVoltage="=" Vl="= 0.1 V" Vh="= 0.8 V" Lmin="= 0.35E-6 m" Abstract="True" toOptimize="True"/>
    </TIAComparator>
  </Receiver>
</OPPLink>
</GenericLink>
```

Several methods have to be written to render these model classes synthesisable (we associate UML with Java for this development task, but there is no technical reason why the same concepts cannot be developed with other OO[11] languages such as C++). We used this in the context of extending an existing AMS synthesis flow and as such have used it for low-level AMS blocks (TIA, amplifiers, filters, and duplexers). XML was used in this respect to formulate soft-IP information and to store all generated numerical firm-IP. Future work will include the use of Pareto-sets to optimally reduce the amount of information stored and data mining techniques to retrieve useful information. Application of this approach to more complex discrete-time and RF blocks is also a goal.

Acknowledgments

This work was partially funded by the European FP6 IST program under PICMOS FP6-2002-IST-1-002131, and by the French Rhône-Alpes region under OSMOSE (PRTP 2003-2005). The authors gratefully acknowledge discussions with Y. Hervé for valuable comments and insights.

References

Carr, C. T., McGinnity, T. M., and McDaid, L. J. (2004) Integration of UML and VHDL-AMS for analogue system modelling. *BCS Formal Aspects of Computing*, 16(80).

Chaudhary, V., Francis, M., Zheng, W., Mantooth, A., and Lemaitre, L. (2004) Automatic generation of compact semiconductor device models using Paragon and ADMS. In: *Proceedings of the IEEE International Behavioral Modeling and Simulation Conference 2004*, IEEE, p. 107.

Doboli, A. and Vemuri, R. (2003) Behavioral modeling for high-level synthesis of analog and mixed-signal systems from VHDL-AMS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(11):1504–1520.

Gielen, G. and Dehaene, Wim (2005) Analog and digital circuit design in 65 nm CMOS: end of the road? In: *Proceedings of Design Automation and Test in Europe (DATE) 2005*. IEEE Computer Society, Munich, Germany, pp. 36–41.

Hamour, M., Saleh, R., Mirabbasi, S., and Ivanov, A. (2003) Analog IP design flow for SoC applications. In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS 2003)*, pp. IV–676.

[11] Object-oriented

Hervé, Y. and Fakhfakh, A. (2004) Requirements and verification through an extension of VHDL-AMS. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2004*. ECSI, Lille, France, p. 91.

O'Connor, I., Mieyeville, F., Tissafi-Drissi, F., Tosik, G., and Gaffiot, F. (2003) Predictive design space exploration of maximum bandwidth CMOS photoreceiver preamplifiers. In: *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS) 2003*, IEEE.

Riccobene, E., Scandurra, P., Rosti, A., and Bocchio, S. (2005) A SoC design methodology involving a UML 2.0 profile for SystemC. In: *Proceedings of the Design Automation and Test in Europe (DATE) 2005*. IEEE Computer Society, Munich, Germany, pp. 704–709.

Tissafi-Drissi, F., O'Connor, I., Mieyeville, F., and Gaffiot, F. (2003) Hierarchical synthesis of high-speed CMOS photoreceiver front-ends using a multi-domain behavioral description language. In: *Proceedings of the Forum on Specification and Design Languages (FDL) 2003*. ECSI, Frankfurt, Germany, p. 151.

Tissafi-Drissi, F., O'Connor, I., and Gaffiot, F. (2004) RUNE: platform for automated design of integrated multi-domain systems. application to high-speed cmos photoreceiver front-ends. In: *Proceedings of Design Automation and Test in Europe (DATE) 2004—Designer's Forum*. IEEE Computer Society, Paris, pp. 16–21.

Vachoux, A., Grimm, C., and Einwich, K. (2003) SystemC-AMS requirements, design objectives and rationale. In: *Proceedings of Design Automation and Test in Europe (DATE) 2003*. IEEE Computer Society, Munich, Germany, pp. 388–393.

Vanderperren, Y. and Dehaene, W. (2005) UML 2 and SysML: an approach to deal with complexity in SoC/NoC design. In: *Proceedings of Design, Automation and Test in Europe (DATE) 2005*. IEEE Computer Society, Munich, Germany, pp. 716–717.

IV

UML-Based System Specification and Design

Introduction

This part of the book contains a selection of the most interesting work presented in the FDL'05 on the thematic area "UML-based system specification and design." This thematic area addresses specification and design methodologies such as the model-driven architecture (MDA) approach, which use unified modeling language (UML) to map abstract models of complex embedded systems to highly programmable hardware platforms and heterogeneous systems on chip (SoCs).

The first three chapters in this part have been presented in the session "Model-driven engineering chapter." The first chapter, "Compiled and synthesized UML: a practical approach for codesign" (Chapter 13) by C. Berthouzoz, F. Corthay, T. Sterren, R. Steiner, and M. Rieder, explores a practical approach for bridging the gap between UML models and VHDL. The mapping of UML on VHDL is described as a set of rules that forms the basis for a code generator. The second chapter, by O. Florescu, J. Voeten, and H. Corporaal is on "Property-preservation synthesis for unified control- and data-oriented models" (Chapter 14), focuses on the preservation of real-time system properties when developing models on the path from analysis to synthesis. The third chapter, "Traceability and interoperability at different levels of abstraction in model-driven engineering" (Chapter 15) by L. Bonde, P. Boulet, J. L. Dekeyser, describes a model-driven engineering approach of software design, in which the whole process of design and implementation is worked out around models. The focus is on the interoperability between evolving models from platform-independent to platform-dependent, using an additional traceability model.

The last two chapters selected for this part of the book have been presented in the session "Verification and validation". "Power simulation of communication protocols with StateC" (Chapter 16) by L. Negri and A. Chiarini describes a modeling and simulation flow that can evaluate policies for optimizing power consumption in communication protocol implementations. The fifth and last chapter in this part is by P. Green and K. Tasie-Amadi. "Integrating model-checking with UML-based SoC development" (Chapter 17) addresses the complexities of SoC design where rigorous development methods

and automated tools are required. This chapter presents an approach to formal verification using model-checking, designed for use in the context of a UML-based SoC design flow. By translating UML models to communicating sequential process (CSP), an failures divergences refinement (FDR) model checker can be used to verify specified properties.

Piet van der Putten
Department of Electrical Engineering
Technische Universiteit Eindhoven
Eindhoven, The Netherlands, February 2006

Chapter 13

Compiled and Synthesized UML

A Practical Approach for Codesign

Cathy Berthouzoz, François Corthay, Medard Rieder, Rico Steiner, and Thomas Sterren

Haute Ecole Valaisanne (HEVs)
Infotronics Unit
Rte du Rawyl 47
CH-1950 Sion
Switzerland

Abstract Embedded systems are complex systems with limited resources such as reduced processor power or relatively small amounts of memory and so on. The real-time aspect may also play an important role, but it is definitely not a main consideration of this work. Complexity of recent embedded systems is growing as rapidly as the demand for such systems and can be managed only by the use of a model-driven design approach. Since modeling languages such as unified modeling language (UML) are semiformal, they allow the design of systems that cannot be implemented using formal languages such as C/C++ or very high speed integrated circuit (VHSIC) hardware description language (VHDL). This chapter intends to show how the gap between model and formal language can be bridged. First, a set of rules restricts the use of model elements in a way that the model will become executable. Furthermore, a unique mapping between UML and formal language elements enables automatic code generation. Formal verification at model level is an important consideration and becomes possible by the fact that rules restrict the application of model elements. UML to software (C/C++) and UML to hardware (VHDL) mapping form the base for a practical codesign approach where a part of the system is realized through software and another part through hardware. Mapping of UML to programming languages is well known today and achieved by many tools. Mapping of UML to hardware description languages (HDLs) is less known and not realized in tools. This chapter documents an attempt to define a set of rules and to implement UML to VHDL mapping in a practical code generator. It also shows parts of a real-world sample that was realized to verify usability and stability of rules and map-

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 231–246.

ping. Finally, an outlook on further developments as improvement of the UML to VHDL mapping and a simple codesign process called 6qx will be given.

Keywords: Codesign; UML; VHDL; XML.

1. Introduction

Although embedded systems were not widespread before 1990, they have now become very popular. Affordable prices of big-sized memory and powerful processors form the ideal alchemy for the birth of numerous embedded systems. Another component of this alchemy is the fact that hardware has become programmable. Field programmable gate arrays (FPGAs) with sufficient number of gates at reasonable prices made the borderline between hardware and software vanish. Even though there are a rising number of basic components for embedded systems, and new technologies appear in rapid succession, the system development cycle is still quite traditional as illustrated by Figure 13.1.

2. Codesign

In such a traditional process, hardware and software are developed in parallel, which brings up several issues such as:

- Need of early hardware and software partitioning
- Asynchronous development of hardware and software

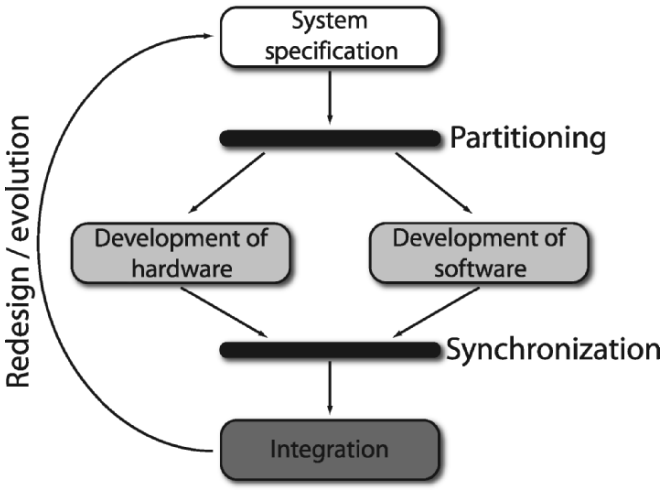


Fig. 13.1 Traditional embedded system development cycle

- Late integration with possible need of redesign
- Missing hardware prevents testing the software before integration

An important reason why development of hardware and software is not integrated is the lack of simple model-based approaches. Several reasons prevent the use of model-driven development:

Existing Codesign tools are very expensive and mostly dedicated: If codesign tools exist, these are almost certainly very expensive and dedicated to a specific thematic and platform. Readapting to other thematic and/or other platforms is practically impossible.

Developers think in terms of code and not model: Traditional thinking (Edwards, 1993; Labrosse, 1995; Perry, 1994) and often also investments that have been made into some existing platform inhibit a change in attitude. Since formal descriptions are what they are and do not heal lack of methodic approach, first experiences in modeling are mostly disappointing and many hardware and software programmers therefore fall back into well-known territory, which means thinking either in hardware or in software code.

Therefore, the Haute Ecole Valaisanne (HEVs) approach of a codesign method was to use existing software modeling techniques already established on the market and to bridge the gap between software engineering and system engineering (codesign) by adding the hardware engineering part. How this was done will be described in detail in the following chapters and sections.

3. A Theoretical Codesign Approach

As a theoretical approach, we have developed a quite simple pyramid with the integrated system model as its top.

As underlying layer, we split up the model into a hardware model and a software model section. This process is called partitioning. The partitioning is done manually to give us the most flexibility to draw the borderline (Figure 13.2) between hardware and software. However, all needed interfaces between hardware and software are automatically created. Each of the models will then be translated into either hardware code (very high speed integrated circuit (VHSIC) hardware description language (VHDL); IEEE, 1987) or software code (C/C++; Micheloud and Rieder, 2002). Afterwards, the code will get synthesized or compiled and then uploaded into the target system. These last two steps are automated and require no user interaction. Figure 13.3 shows the theoretical model.

Finally there has to be a formal verification step. The produced code has to be verified against the model. Not only the software and the hardware code

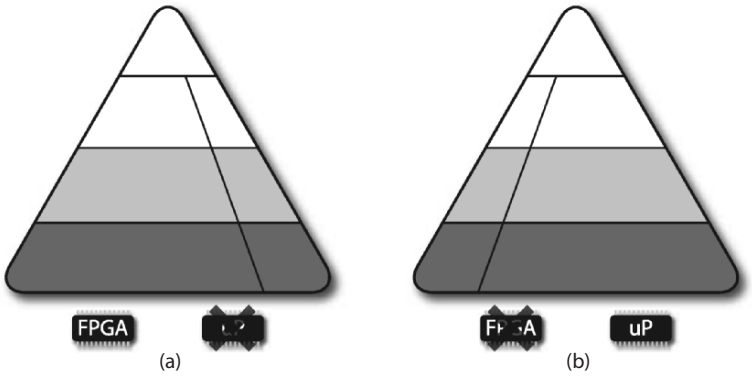


Fig. 13.2 Different degrees of partitioning

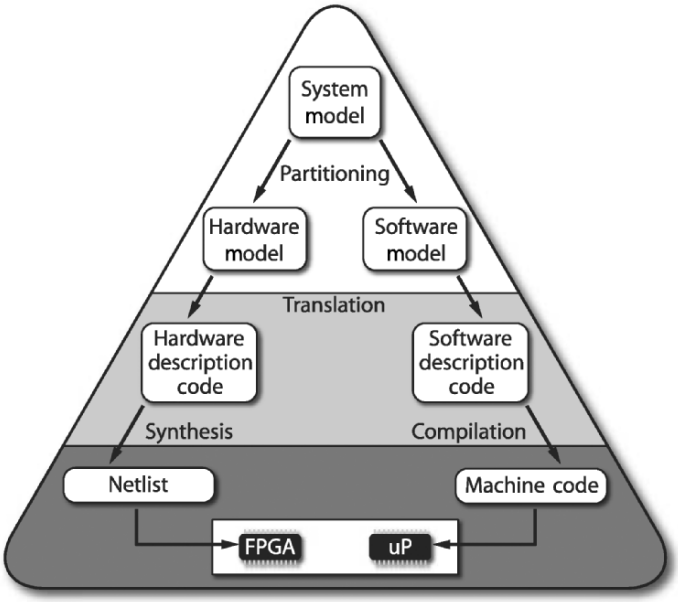


Fig. 13.3 Model-driven codesign of embedded systems

have to be verified but also the semantics of the overall system have to be tested. Timing constraints has to be tested to.

It can easily be seen that integration problems will be minimized, since integration is already part of the model. It also can be seen that different degrees of partitioning are possible throughout this model. Figure 13.2 shows both, one hardware-(a) and one software-centric (b) partition (solution) of a given system.

4. A Practical Codesign Approach

Theoretical approaches are nice to have a basic understanding, but to come to true results practical models have to be developed out of the theoretical ones. We did this by instantiating a codesign model using realistic tools and targets. Figure 13.4 shows an overview of this practical approach. To make the complexity of this problem reasonable, some constraints are introduced:

- Actually, we can do the two extreme partitions: either full hardware or full software.
- A formal verification of the produced code against the model is not yet possible.
- Real-time aspects are only partially taken into account. The system has to reproduce the behavior specified by the model. The code is not able to handle hard-real-time situations. But it is possible to generate very compact and target-specific code due to the flexible translation mechanism.

To understand Figure 13.4, one must understand the unified modeling language (UML) (Booch et al., 2005) approach we use to manage different partitions of the embedded system on model level. Packages, classes, and state charts are used to model the target-independent elements and the behavior of the system. Furthermore, one component is defined for each of the partitions

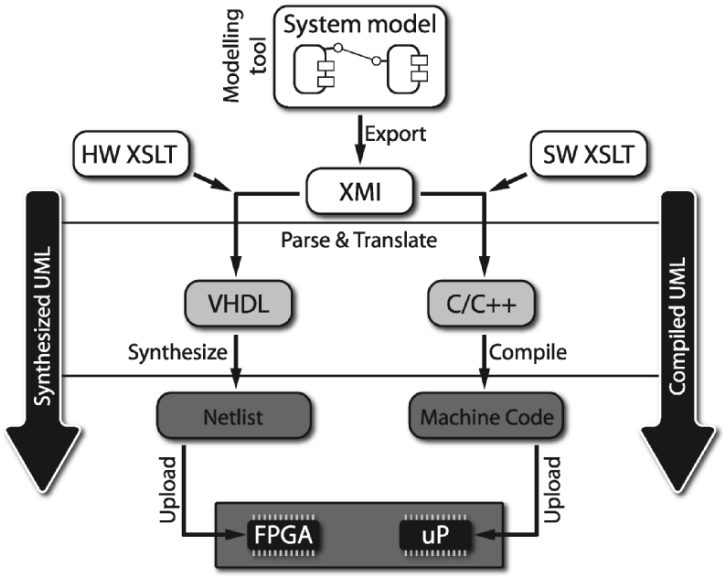


Fig. 13.4 A practical codesign approach

(hardware/software). Interfaces allow using the same system description for several partitions and targets. Doing so makes it possible to define an arbitrary number of hardware software components, respectively, for an arbitrary number of targets. A partition-specific component holds all the information that is model-level related, such as which packages or classes are part of this specific partition. It also holds all target-specific information, such as which tool will generate the code, how this code will be translated, how it will be synthesized or compiled, and how it will be uploaded to the embedded system. In this way it is possible to automate the entire build and execute command chain.

Basically, the build command is run with the components name as parameter; the entire model will then be exported and either the hardware or software translator parses and translates the information related to the specific component (partition) out of the exported model. It would also be possible to have a single translator, which receives one more parameter that determines whether to translate model information into hardware or software code. For reasons of simplicity (a translator is a quite complex matter), we decided to build two separate ones.

It has to be emphasized that the partitioning is done manually by defining components and assigning packages and classes to them. Also, components have to be equipped with target-specific information. But it also has to be emphasized that partitioning is done after modeling the system and just before generating the code.

5. Translation

The correct translation of the UML elements into code is the core problem of any realistic codesign approach. Many researchers have already worked on this problem. The translation of UML to software code has been thoroughly researched and offers good stability and performance today. When it comes to the translation of UML into hardware the papers, McUmber and Cheng (1999) and McUmber and Cheng (2001), are good examples. Unfortunately, many of these findings lead toward code that cannot be compiled/synthesized because their main focus is on the model level. Our work requires that the generated code can be compiled/synthesized. To do real codesign, both the model and the generated code have to be adapted to each other. Therefore, we focused our research on solving this problem (Steiner, 2004). The following sections describe the main results of this work.

5.1 Hardware Thinks Differently

It is very common in UML to communicate using events. But the concept of events known from software does not exist in hardware. The only hardware event is the continuous clock signal, the system clock. All other

communication is done using signals that hold their value until they are told to change it.

In software, events can be used for communication tasks. But that is not true in hardware. Even if one defines a signal with a pulse width of one period of the system clock as an event, this will not be an event because only the value of the signal is taken into consideration and not the pulse width. In UML this would mean that state transitions are only decorated with a guard but no trigger.

This, and the fact that UML is closely related to software, brings us to fill the gap between UML and hardware. Therefore, we need to develop a communication mechanism that can act as expected in UML (see Section 5.3). We need also to define some rules to coordinate the use of UML for hardware and software systems. There are three reasons for doing so:

1. In every hardware description language (HDL), one can describe functions and situations that cannot be synthesized. But if the designer follows some basic and simple rules, he can be sure that the design will be synthesizable. In UML, the same situation exists. One can design a model that can be translated neither to software nor to hardware.

2. Until now, UML was used to design software (Douglass, 1999, 2002) only. There is a lack of experience when it comes to creating models that can be translated to hardware and software. Defining some rules will help the designers to improve their know-how and it will add quality to their models.

3. Guided by these rules, the designers can be sure that their models will be suitable for software/hardware codesign.

The above-mentioned rules would normally be part of a hardware/software process. We are currently working on such a process (see Section 7.2). However, describing these rules now would go beyond the scope of this document. Instead, we would like to concentrate on the mapping of UML elements to VHDL code, which is the subject of the next section.

5.2 UML Elements

Since there is a large number of UML elements, this first approach to UML to VHDL translation does not take them all into account. The translated elements are classes, attributes, operations, class diagrams, objects, object diagrams, associations, ports, interfaces, events, and state charts. As our real-world example shows, these elements are enough to model the behavior of simple systems (McUmber and Cheng, 1999, 2001).

There is a strong parallelism (Douglass, 2002; Naylor and Jones, 1995; Rajan, 1998) between these elements and the traditional concepts used by

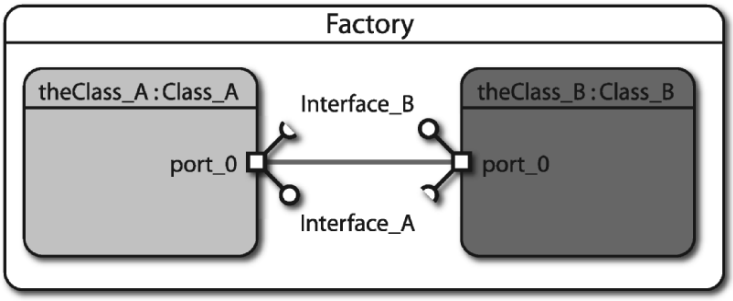


Fig. 13.5 UML object diagram

hardware designers. State charts became a very popular concept among hardware designers to describe a design before implementing it. The top-down concept is also widely used for analyzing hardware designs. The system is seen as a black box with inputs and outputs. The black box is broken down into smaller parts, each one seen as a black box itself. Inputs and outputs of the black boxes are connected in order to establish the communication. This process goes on until the desired granularity is obtained. Finally, the black boxes are given a described behavior, e.g., by a state chart. The same analysis and design process is possible with object diagrams, an object corresponding to a black box. Communication between objects is also possible using ports and interfaces. An object can be equipped with ports and interfaces, and ports are interconnected by links. Figure 13.5 shows a typical example.

The next section provides more details about the translation technique we used.

5.3 UML to VHDL Mapping

It is important to find optimal patterns to translate UML elements to VHDL. Due to the fact that an UML element can have several decorations, it is important to find a general VHDL description that can handle all the UML decorations. Without going too much into details we will now show how UML elements are translated into VHDL. To give an idea of the translations result, Listing 13.1 shows the VHDL code that corresponds to the UML elements of Figure 13.5. The following lines will give an overview and a brief description of the used patterns:

- The class `Factory` is translated into an entity construct. The entity named `Factory` consists of a list of ports and an architecture section. By default the two inputs `reset` and `clock` are added to the entity's port list. Depending on the class ports and interfaces, other input and

output ports may be added to the port list. The main implementation (the behavior) of the class can be found in the architecture section.

- An object is an instance of a class. In VHDL it is possible to create an instance of an entity. To do this, first a component has do be defined inside the architecture containing the instance. According to Figure 13.5, the components named `Class_A` and `Class_B` are defined (cf. A:, B: at Listing 13.1). The second step is to create an instance and to map the ports to other ports or to signals. The instances are called `theClass_A` and `theClass_B` (cf. C:, D: in Listing 13.1).

- The UML elements port and interface are translated into input and output ports of an entity. Provided interfaces are translated into input ports and required interfaces are translated into output ports. The names of VHDL ports are the same as the ones of the ports and interfaces in the UML model (cf. A:, B: in Listing 13.1).

- In UML, links are used to interconnect objects via ports and interfaces (see link between the two objects in Figure 13.5). Depending on the used interfaces, ports and objects, several signals will be defined in VHDL. These signals will be used in the port map sections of components to realize a connection between components (cf. C:, D: in Listing 13.1).

The second type of UML diagrams discussed here are state charts. Due to the fact that state charts are well known to hardware designers, we will briefly explore this topic and explain the main ideas and concepts we used to translate UML state charts into VHDL.

As state charts are used to describe the behavior of a class, the corresponding translation is put into the architecture of the VHDL entity. First, a new data type representing all the states of the state chart is defined.

A single UML state is translated into up to three VHDL states (cf. Figure 13.6). This procedure is necessary to handle the three different types of actions that can be embedded in a state: actions on entry, reactions in state, and actions on exit. Even if this seems to add a lot of overhead to the hardware, it is indeed not the case. As doubling the number of states requires one more single flip-flop, tripling the number of states will therefore add only two additional flip-flops. Furthermore, the translator is optimized in a way that it adds the additional states only when they are referenced. The actions on entry and actions on exit are executed once. The reactions in state are executed on each rising edge of the clock, while the system stays in the current state.

State charts are implemented as case structures where every case represents a state or one of the additional pseudostates. A state chart is described by two VHDL processes: the first one handles all the transition conditions and the

Listing 13.1 Generated code of UML diagram in Figure 13.5

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity Factory is
5    port(
6      reset : in STD_LOGIC;
7      clock : in STD_LOGIC
8      -- other ports here.
9      );
10 end Factory;
11
12 architecture FA of Factory is
13   -- Declare signals to interconnect nested blocks
14   signal link_0_Interface_A_methode_0: STD_LOGIC;
15   signal link_0_Interface_B_methode_0: STD_LOGIC;
16   -- A: Declare class Class_A
17   component Class_A
18     port(
19       clock : in STD_LOGIC;
20       reset : in STD_LOGIC;
21       port_0_Interface_A_methode_0 : in STD_LOGIC;
22       port_0_Interface_B_methode_0 : out STD_LOGIC);
23   end component;
24   -- B: Declare class Class_B
25   component Class_B
26     port(
27       clock : in STD_LOGIC;
28       reset : in STD_LOGIC;
29       port_0_Interface_B_methode_0 : in STD_LOGIC;
30       port_0_Interface_A_methode_0 : out STD_LOGIC);
31   end component;
32 begin
33   -- C: Instantiate the Class_A
34   theClass_A: Class_A
35     port map (
36       reset => reset,
37       clock => clock,
38       port_0_Interface_A_methode_0 => ↵
             link_0_Interface_A_methode_0,
39       port_0_Interface_B_methode_0 => ↵
             link_0_Interface_B_methode_0);
40   -- D: Instantiate the Class_B
41   theClass_B: Class_B
42     port map (
43       reset => reset,
44       clock => clock,
45       port_0_Interface_A_methode_0 => ↵
             link_0_Interface_A_methode_0,
46       port_0_Interface_B_methode_0 => ↵
             link_0_Interface_B_methode_0);
47 end FA;
```

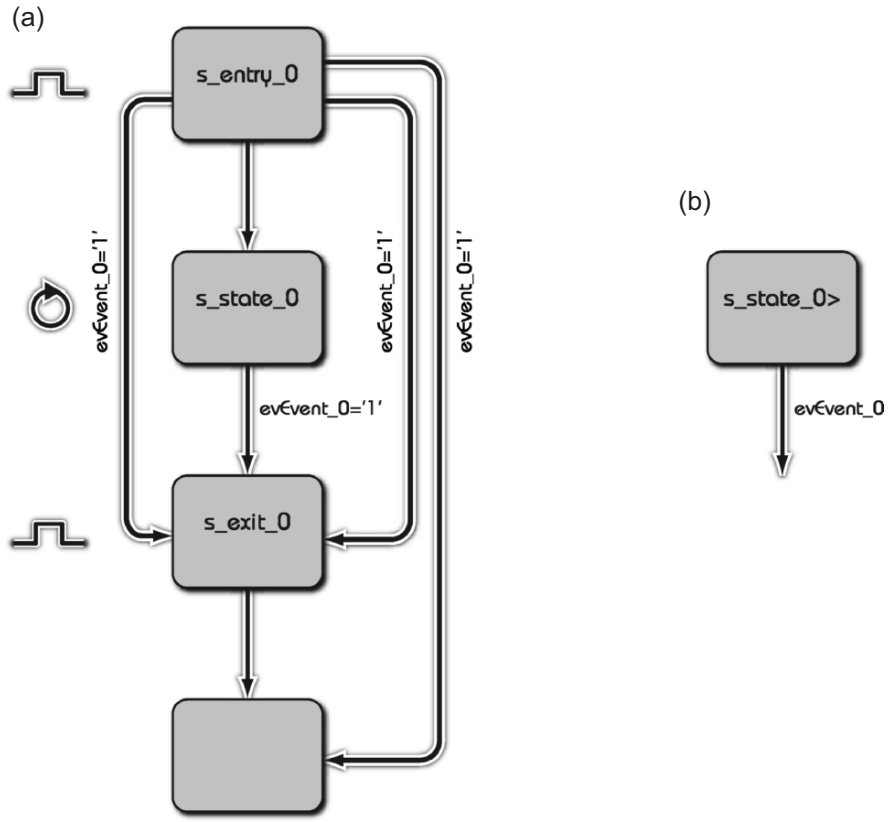


Fig. 13.6 VHDL representation (a) of a single UML state (b)

state sequencing and the second one takes care of the execution of the various actions of a state.

By doing so, we can introduce a simple communication mechanism. An event is defined as a signal set to the value '1' for exactly one period of the system clock. In the action handling process, all event signals are by default set to the value '0'. When an event has to be fired, the signal is simply set to '1'.

The result of the above translation is very generic. We use basic VHDL elements and well-known structures. This has a number of advantages.

- If the generated code is semantically correct, it is granted that the generated code is synthesizable.
- The generated code is platform-and manufacturer-independent. This is because we do not use target-specific elements such as memory or multiplier blocks.

The next section describes a demonstrator that was created to verify the techniques exposed in this section.

6. Experimentation

As usual in research projects, all obtained results have to be verified and proven at the end.

A simple chronometer demonstrator was built up. We used an ARM 7 equipped board with a minimal operating system called IDF (interrupt-driven framework) as software target, and a Xilinx Spartan II–equipped board as hardware target. The chronometer itself consists of a stepper motor, some push buttons, and an optical sensor. An UML model of the system was created, then synthesized once for the hardware target and compiled once for the software target. Both systems were working without touching the model. All code was automatically generated, compiled, uploaded, and started in the targets. Figure 13.7 shows the schematic of the demonstrator.

The following conclusions can be drawn from the chronometer experience:

1. The software code generator we used was the built-in one of the modeling tool. For future development of the codesign project, it has to be replaced by one of the same types as the hardware code generator. This is necessary to allow correct interface integration between hardware and software.

2. Moving the partition line between hardware and software means involving interfaces. For our purpose we implemented interfaces manually, but for a real-world development process it is a must to at least

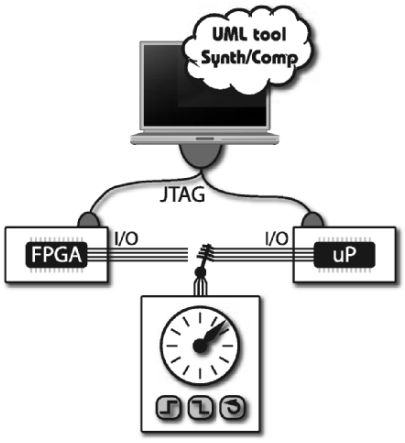


Fig. 13.7 Principle of the chronometer codesign demonstrator

semiautomate this action. This means that it would be possible to "drag and drop" readymade interface blocks into the model and to connect them to the correct locations in the hardware and software fragment of the system's model.

3. The mapping for the hardware should also be optimized. It would be especially nice to parameterize target-specific matter inside the model and not to find these adaptations somewhere in the translator.

4. Not all elements of UML 2.0 (Booch et al., 2005) have been used. This was partially due to the fact that the software built-in translator did not recognize them, at least not in the version of the tools we were using. This problem will automatically be corrected by introducing the "home-brew" software translator.

5. More complex demonstrators must be implemented to stress-test the codesign approach we are currently using. But it must be stated that the results we obtained until now are very encouraging and that the generated systems are amazingly stable.

It would be nice to have a framework inside which development is rolling down. A first approach of such an allover development process is briefly outlined in the conclusion.

7. Conclusions

The above-mentioned experiences lead to a certain number of conclusions that have to be applied in the very near future to the described codesign approach.

7.1 Tool Chain Improvement

The most important improvements that will be done around the tool chain are:

- Improved hardware code generation patterns will be implemented in the hardware translator.
- A separate software translator will be added to the tool chain.
- Standard interfaces will be defined in UML as patterns that can be applied to a given situation.
- The whole approach will be intensively tested by the means of real-world projects and demonstrators.

Another more important development will be to introduce a general formalism that embeds the present experimental codesign process. The reason for this is that any modeling activity requests formalism and sequencing. An outlook on this process is presented in the next section.

7.2 The 6qx Process

The definition of a simple codesign process is the logical consequence of this conclusion, and because we had already defined a software-centric process (6q; Rieder, 2005) we will extend this one into a codesign process. The 6q process has been developed in an embedded systems context and therefore provides a quite good potential to cover hardware development aspects also. The method of the 6q process is object-oriented and the model is incremental. It consists of six major steps: system specification, analysis, design, translation, validation, and integration.

These steps will also be contained in the new 6qx process, but will be adopted to meet codesign requirements as follows:

The first two steps, system specification and analysis, gather information about the system to be developed and map results into a UML model by means of use-case diagrams, interaction diagrams, class diagrams, state charts, and deployment diagrams. Since these steps try to specify and analyze the system, they do not care about implementation details (hardware/software). The major difference of these steps compared to the original 6q process, where hardware and software are developed in parallel, will be the removal of the hardware–software partitioning decision. It is delayed into the design step, because the model covers both hardware and software of the system.

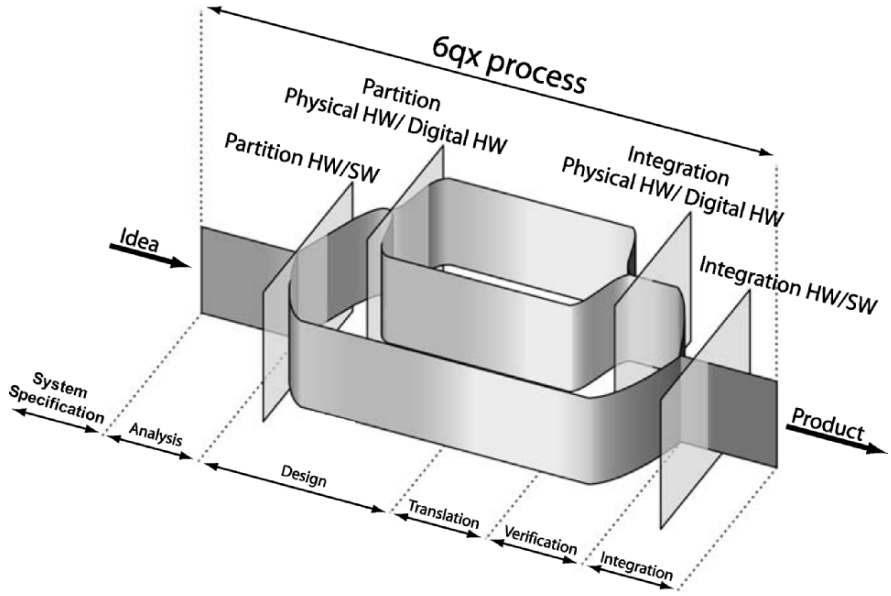


Fig. 13.8 Overview of the 6qx codesign process

The design step will transform the flat analysis model into a well-structured model that can be partitioned into a hardware (HW) and a software (SW) partition according to various criteria (costs, speed, physical limitations, etc.). The hardware partition may be split up once again into a programmable hardware (PHW) partition and an analog/digital hardware partition (DHW). Interfaces between both partitions have to be defined after partitioning or even while partitioning. The 6qx process will contain recommendations about use and implementation of interfaces in the form of interface patterns defined in UML. It will also contain hardware and software design rules (cf. Section 5.1) in the form of patterns defined in UML.

The translation step will regulate implementation details. Important elements that will be introduced here into the system model are components that will bind the hardware and the software partition to specific targets (cf. Section 5.2).

The validation step will be responsible for verifying correct functioning of the designed hardware and/or software. This will be achieved by reusing formal descriptions of behavior from the specification and analysis step. Simulators will be used to verify correct behavior.

The integration step will put it all together and finally verify the correct allover system behavior and stability. Erroneous behavior will result in feedback toward the analysis pipe; insufficient stability will result in feedback toward the design pipe. Figure 13.8 gives an idea of the 6qx process.

References

Booch, Grady, Rumbaugh, James, and Jacobson, Ivar (2005) *The Unified Modeling Language User Guide*, 2nd edn. Addison-Wesley Object Technology Series. Addison-Wesley Professional, Boston, MA.

Douglass, Bruce Powel (1999) *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley Professional, Boston, MA.

Douglass, Bruce Powel (2002) *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional, Boston, MA.

Edwards, Keith (1993) *Real Time Structured Methods: Systems Analysis*. Wiley Professional Computing. John Wiley & Sons, New York.

IEEE (1987) *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076, 1987 edn. IEEE, Los Alamitos, CA.

Labrosse, Jean J. (1995) *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. 1st edn. R&D Publications, Lawrence, KS.

McUmber, William E. and Cheng, Betty H. C. (1999) UML-based analysis of embedded systems using a mapping to VHDL. In: *Proceedings of IEEE High Assurance Software Engineering (HASE) 1999*. IEEE, Washington, DC.

McUmber, William E. and Cheng, Betty H. C. (2001) A general framework for formalizing UML with formal languages. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE) 2001*. Toronto, Ontario.

Micheloud, Marylène and Rieder, Medard (2002) *Programmation orientée objets en C++: une approche évolutive*, deuxiéme edition, Collection informatique. Presses polytechniques et universitaires Romandes, Lausanne, Suisse.

Naylor, David and Jones, Simon (1995) *VHDL: A Logic Synthesis Approach*. Chapman & Hall, London.

Perry, Douglas L. (1994) *VHDL*, 2nd edn. McGraw-Hill, New York.

Rajan, Sundan (1998). *Essentials VHDL: RTL synthesis done right*. S&G Publishing, Scotland.

Rieder, Medard (2005) A simple and complete process for embedded systems development. In: *Proceedings of the Embedded World Conference*. Nurenberg, Germany, pp. 876–885.

Steiner, Rico (2004) Hardware & software co-design. Diploma thesis, Haute Ecole Valaisanne (HEVs), Infotronics Unit, Sion, Switzerland.

Chapter 14

Property-Preservation Synthesis for Unified Control- and Data-Oriented Models

Oana Florescu, Jeroen Voeten, and Henk Corporaal
Eindhoven University of Technology and the Embedded Systems Institute
P.O. Box 513
5600 MB Eindhoven
The Netherlands

Abstract In the software/hardware engineering model-driven design methodology, preservation of real-time system properties can be guaranteed in the model synthesis up to a small time-deviation. Therefore, this methodology is well suited for the design and implementation of control systems in which execution times of actions are small; thus the time-deviations obtained are small. However, in systems containing time-intensive computations, the time-deviations become large and, consequently, the real-time properties are much weakened. This chapter proposes an approach for obtaining stronger preservation of the observable properties of the system by abstracting from its internal unobservable actions. In this way, a unified way of analysis and synthesis of a larger area of real-time applications can be obtained, which enables designers to achieve predictability in the design of many systems.

Keywords: real-time; property-preservation synthesis; observation equivalence.

1. Introduction

The main purpose of modelling is to help engineers understand the relevant aspects of a system, while avoiding the expense and trouble of actually building it. Whereas traditional forms of engineering have a well-established modelling methodology, software engineering, and particularly real-time embedded software is still an emerging discipline. Although it is applied to increasingly complex systems, its modelling techniques are neither mature nor reliable yet (Selic and Motus, 2003). Nevertheless, software models have a

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 247–262.

unique and remarkable advantage over most other engineering models; they can be used to automatically generate the realisation of the system modelled, which is an executable program for a particular platform. Starting with a simplified and highly abstract model, refinements can be carried out until a complete specification is obtained, including all the details necessary in the final product. From such a detailed specification, adequate computer tools can generate an implementation.

The model-driven architecture (MDA) initiative of the Object Management Group (Miller et al., 2001) shows that the interest in technologies for supporting model-driven development has increased. In the development trajectory proposed in MDA, system models are made from early stages to help designers in reasoning about different trade-offs. By making design decisions and adding the corresponding details to the model, the design space is narrowed. The software models are kept independent from the platform as long as possible in this design trajectory. This platform-independence provides the flexibility of reusing the design model and/or of targeting it to a different platform. Moreover, it may allow the prediction from the model of a suitable platform. Going lower in the design pyramid by increasing the number of details, a complete specification can be obtained from which the software implementation can be automatically generated.

The software components employed in the embedded systems, like the ones in cars, airplanes, printer/copier machines, or medical devices, are supposed to synchronise and coordinate different processes and activities. Therefore, their behaviour must meet hard timing constraints, either for people's safety or simply to ensure things work correctly. Usually, a real-time software component must work together with other software and hardware components to obtain the specified behaviour. Its correctness depends on both the logical result and the moment in time when the result was ready. Experience showed that existing model-driven development approaches for software systems are not suited to cope with real-time system design. Traditional design approaches proved themselves unable to capture adequately both the functional and non-functional (timing) characteristics of a system, while abstracting from low-level details. For predictably designing such systems, an appropriate methodology needs to provide (Huang et al., 2003b) (i) a suitable modelling technique that can appropriately capture functional and timing properties in models in order to formally analyse them, and (ii) a mechanism to generate the implementation from the model while preserving the properties analysed, phase also known as model synthesis.

The Software/Hardware Engineering (van der Putten and Voeten, 1997) is a model-driven design methodology suitable for analysis and synthesis of real-time systems in which actions need small execution time. In this chapter, we propose an idea for synthesis, using the same methodology, of system models

containing both short actions and time-intensive computations while still preserving the real-time properties analysed. We make observations regarding the possibility of code generation from models which are equivalent from the perspective of an external user. By applying this idea, we can have a predictable and unified trajectory from a model towards a property-preserving system realisation for a large area of real-time applications (both control-oriented and data-oriented).

The remainder of this chapter is organised as follows. Section 2 discusses related research. Section 3 presents the technique used for formal modelling of systems. Section 4 shows how the properties of control-oriented system models are preserved in their implementations. Section 5 discusses a way to synthesise models of applications that contain time-intensive computations. Conclusions are drawn in Section 6.

2. Related Research

In the context of model-driven approaches for software development, the Unified Modelling Language (UML) (OMG, 2003) has been adopted as a standard facility for constructing models of object-oriented software. UML proved to be suitable for modelling the functional aspects of a system, which can also be correctly synthesised. Moreover, extensions were defined to it to provide a standardised way of denoting non-functional (timing) aspects for real-time systems as well (OMG, 2005). Nevertheless, the application of mathematical analysis techniques remains complicated due to the difficulty of relating formal techniques to UML diagrams, whereas the synthesis of the model cannot preserve the timing properties of the system.

For modelling purposes, a number of techniques and theories were proposed, targeting a certain view over a system, e.g., correctness analysis, scheduling analysis. For example, classic scheduling theory (Buttazzo, 1997) provides techniques for the analysis of timing behaviour of a system and for the scheduling of its tasks onto the target platform such that the timing constraints are satisfied. Real-time systems are assumed to be composed of independent tasks with periodic arrival times; therefore, well-studied methods, like rate monotonic scheduling, can be applied. Nevertheless, analysis of such models often yields pessimistic results and it is not suitable for handling non-periodic tasks with non-deterministic behaviours. Moreover, the models analysed by classical scheduling analysis do not incorporate information about the functionality of tasks, which makes them unsuitable for model synthesis.

A way to relax the stringent constraints on task-arrival times is by using automata with timing constraints to model task-arrival patterns. The model can describe concurrency and synchronisation of periodic, sporadic, pre-emptive, or non-pre-emptive real-time tasks with or without precedence constraints. An

automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable (all associated tasks can be computed within deadlines). Based on the results obtained for schedulability analysis on timed automata, the TIMES tool (Amnell et al., 2003) has been designed for schedulability analysis and synthesis of real-time systems. A model consists of (i) a set of application tasks whose executions may be required to meet different timing, precedence, and resource constraints; (ii) a network of timed automata describing the task-arrival patterns; and (iii) a pre-emptive or non-pre-emptive scheduling policy. From such a model, the TIMES tool can generate a scheduler and compute the worst-case response time for all tasks. Nevertheless, TIMES tool does not have enough expressive power to describe all kinds of data computations involved in a system. This is due to the exhaustive analysis that might lead to state space explosion problems if there are many details involved. Therefore, TIMES analysis and synthesis might not scale up to any kind of system.

3. Real-Time Systems Models

The Software/Hardware Engineering (van der Putten and Voeten, 1997) is a system-level design methodology that uses a UML profile to formulate the concepts needed for the realisation of the requested functionality of a system. The UML profile smoothes the application of the Parallel Object-Oriented Specification Language (POOSL) (van der Putten and Voeten, 1997) to develop an executable model, as shown in Figure 14.1. POOSL formalises the behaviour specified in informal UML diagrams, establishing a formal executable model. The realisation of the system can be generated from this model using the Rotalumis tool (van Bokhoven, 2002).

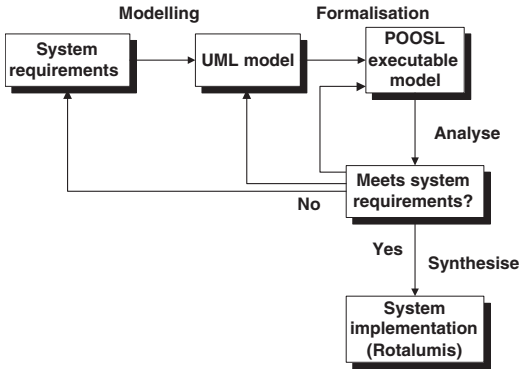


Fig. 14.1 SHE method for real-time systems design

POOSL is equipped with mathematical semantics that can formally describe concurrency, distribution, communication, timing, and functional features of a system in a single executable model, using a small set of very powerful primitives. Primitives can be combined in an unrestricted fashion and any combination has a precisely defined meaning. The formal semantics guarantees a unique and unambiguous interpretation of a POOSL model, guided by semantical axioms and rules independent of the underlying execution platform. The importance of the formal semantics of a modelling language in supporting the predictability of the system design process is investigated in (Huang et al., 2005).

POOSL consists of a process part and a data part. The process part (processes and clusters), based on a real-time extension of the process algebra Calculus of Communicating Systems (CCS) (Milner, 1989), is used to specify the real-time behaviour of active components. The data part, based upon traditional concepts of sequential object-oriented programming, is used to specify the information that is generated, exchanged, interpreted, or modified by the active components.

The semantics of POOSL is defined as a timed labelled transition system, as the example in Figure 14.2 shows, where S_1–S_8 represent states of the system, a_1–a_4 action transitions, and t_1–t_3 time transitions. A timed labelled transition system represents an abstract view over a system, considering it as an entity having some internal state and, depending on that state, it can engage in transitions leading to other states. Such a transition might be autonomous or stimulated by the environment. When action transitions take place, the state of the system changes by changing its content (e.g., when an event happens, certain parameters of the system get different values). In case of time transitions, only the time parameter changes its value according to the time interval specified, whereas the rest of the system content stays the same.

In a model based on the timed labelled transition system, the execution has two phases, as shown in Figure 14.3: (i) the state of a system changes either by asynchronously executing atomic actions, such as communication or data computation, without passage of time (phase 1), or (ii) by letting time pass synchronously without any action being performed (phase 2).

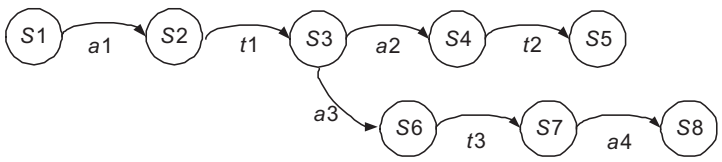


Fig. 14.2 Example of a timed labelled transition system

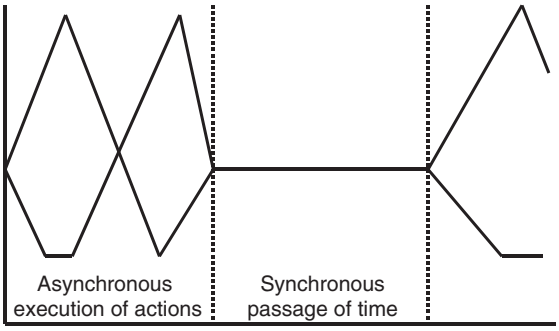


Fig. 14.3 Two phases of model execution

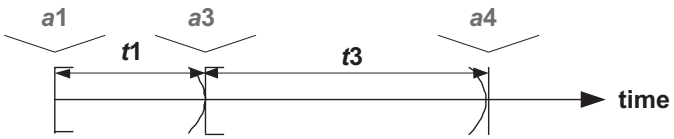


Fig. 14.4 A timed trace of the transition system

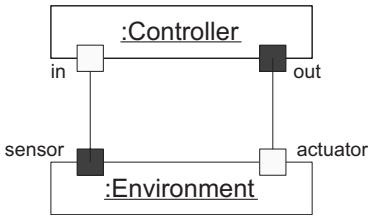


Fig. 14.5 The UML model of a simple controller

A run over a transition system represents a timed trace, as the one in Figure 14.4, where each action is executed at a particular time. As there are many possible runs due to the parallelism and non-deterministic choices that can be expressed, a POOSL model represents, in fact, a set of timed traces. If all the traces of the model satisfy a real-time property (e.g., that a particular event happens at a certain moment), then the model of a system has that particular real-time property.

For illustration purposes, a simple controller is used in the following. The UML graphical representation of this system is provided in Figure 14.5 using the UML stereotype "`capsule`". The small black squares in the figure represent output ports and the white ones represent input ports.

Listing 14.1 POOSL model of the simple controller

```
1 in?input(x);          /* x is received as a message */
2 computation(x)(y);    /* x is the input, y is the output of ↵
      computation */
3 delay deadline;       /* wait for deadline units of time */
4 out!output(y);        /* y is sent as a message */
```

Fig. 14.6 The timed labelled transition system of the model

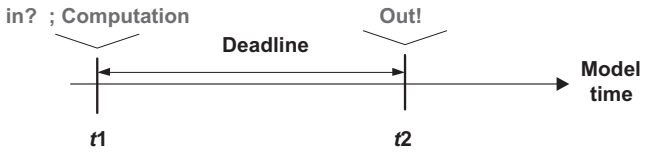


Fig. 14.7 A timed trace of the controller

The POOSL specification[1] of the system is given in Listing 14.1. The controller reads some data `x` from the environment, performs computations with it and delivers the result `y` back to the environment at a certain time.

The timed labelled transition system underlying the POOSL model looks like in Figure 14.6. According to the semantics of the language, a timed trace of the model is the one shown in Figure 14.7. `in?input(x)` and `computation(x)(y)` are executed in this exact ordering, without consuming any time and at the same instant t_1. Then, time passes for `deadline` units ($t_2 = t_1 +$ *deadline*) and, finally, `out!output(y)` is instantly performed at t_2.

4. From a Model to Its Realisation

As mentioned in the previous section, a real-time system can be formalised as a set of timed traces. If two timed traces have the same sequence of actions, a notion of distance between them is defined. The distance represents the largest deviation between the ending points of corresponding time intervals, as shown in Figure 14.8. Two timed traces whose distance between them is equal to ϵ are called ϵ-close. If two execution traces are ϵ-close and one of the traces satisfies

[1] Note that the notations in a POOSL specification are CCS alike.

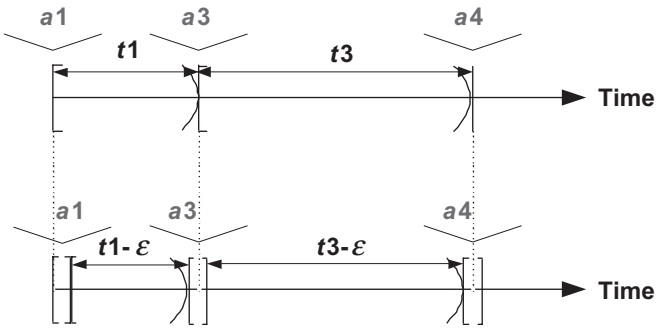


Fig. 14.8 Timed traces ϵ-close

a real-time property,[2] then this property, weakened up to ϵ,[3] is satisfied in the second trace as well. This result was mathematically proved in (Huang et al., 2003a).

Both the model and the realisation of a system can be viewed as sets of timed traces. To obtain an implementation of a system that preserves the properties analysed in its model, thus an implementation consistent with the model, two things must be achieved: (i) to generate a trace in the implementation from the set of execution traces of the model, and (ii) to make the corresponding traces in the model and in the implementation to be ϵ-close.

A mechanism of generating a trace from a POOSL model was proposed and proved correct in (Geilen, 2002). The data part of a POOSL model is directly translated into corresponding C++ expressions. Each process in the model is represented by a C++ data structure named process execution tree (PET), whose nodes represent statements in the specification of behaviour. During the evolution of the system, a PET scheduler makes choices for granting actions or time transitions, while each PET adjusts its internal state according to the choice of the PET scheduler. This mechanism guarantees that the realisation of the model generated by the code generation tool is a trace from the model.

Although actions in a model are regarded timeless, in reality, it will always take a certain amount of time to execute them; between the corresponding traces there appears a *time-deviation*. If the distance between these two traces is ϵ (ϵ-hypothesis), then *all* the properties of the model are preserved up to ϵ in the implementation.

To generate the implementation ϵ-close to the POOSL model, the code generation tool, Rotalumis (van Bokhoven, 2002), synchronises the *model time*

[2] An example of a real-time property is that a certain action happens at a particular moment in time.

[3] If a property P is true in the first trace in the interval $[t_1, t_2]$, the other trace satisfies P in the interval $[t_1 - \epsilon/2, t_2 + \epsilon/2]$.

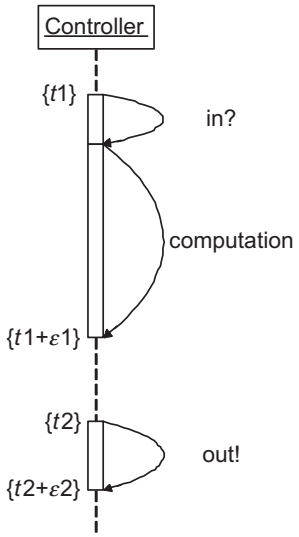


Fig. 14.9 Implementation of the controller in physical time

with the *physical time*. As shown in the UML sequence diagram from Figure 14.9, all the actions that happen instantly in the model at a certain time t (in Figure 14.7, `in?` and `computation` happen at model time t_1, `out!` at t_2) are executed within a small ϵ amount of time around the corresponding moment in physical time (`in?` and `computation` are executed in ϵ_1 around physical time t_1, `out!` in ϵ_2 around t_2). To maintain the synchronisation between model time and physical time, **`delay`**s are not executed in the implementation exactly as specified in the model (*deadline* units of time), but physical time passes until the next corresponding moment in the model time is reached (the delay $deadline = t_2 - t_1$ is shortened to $t_2 - t_1 - \epsilon_1$).

The size of the maximum time-deviation between a model and its implementation can be obtained at the time of generation and execution of code by using measurements. On the other hand, it can be estimated from the model itself, using the Y-chart scheme (depicted in Figure 14.10) approach for design space exploration. This scheme contains the models of both the real-time application and the target platform, and by analysing their mapping, as shown in (Florescu et al., 2004b), the time-deviation can be monitored. This deviation depends on how many actions need to be executed at the same time in the model as well as on their execution times. If the value obtained for ϵ is considered too large, either the implementation is generated for a higher performance platform on which the execution of all the actions takes less time, the mapping is changed, or the model is redesigned.

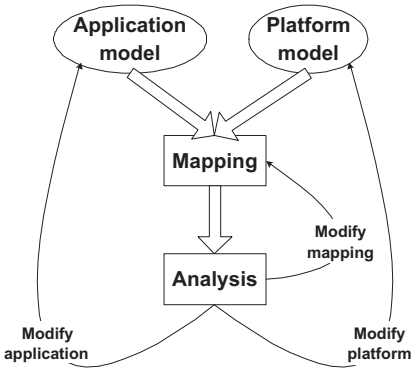


Fig. 14.10 Y-chart scheme for real-time systems design

5. Realisation of Systems with Time-Intensive Computations

In a model of a real-time system, usually a distinction can be made between *actions* and *time-intensive computations*. *Action* is the name given to an "activity" specified in the model that needs small execution time on the target platform (e.g., a control action). On the other hand, *time-intensive computations* are the "activities" specified in a model that usually need a considerable amount of time for execution, as it is the case of the `computation` in Figure 14.9. In case of real-time systems containing such computations, the time-deviation between the model and the implementation is usually large. Therefore, with the current generation of code, the properties analysed in the model will be much weakened in the implementation.

Nevertheless, in data-oriented real-time applications, many computations that take considerable amount of time are modelled (for example, different multimedia algorithms must be applied on a stream of data). For this kind of system, it is not intended for the computations to be instantaneous, but to be finished before a *deadline* when the results must be given to the environment (like in the example given in Section 3).

Two systems are called observational-equivalent if they cannot be distinguished between them through the interaction of a user with each of them. They have the same observable properties[4] and the same set of timed traces with respect to these properties. Therefore, an implementation preserving the observable properties of a model preserves the observable properties of the observational-equivalent one.

[4] A user can see the same properties by interacting with the systems.

Listing 14.2 Observational-equivalent model of the controller

```
1 in?input(x);
2 computation1(x)(y1);
3 delay deadline1;
4 computation2(y1)(y2);
5 delay deadline2;
6 computation3(y2)(y);
7 delay deadline3;
8 out!output(y);
```

Based on this insight, in case of systems with time-intensive computations, instead of generating an implementation for the original model we could generate the implementation for an observational-equivalent one. In Listing 14.2, we give a specification, which is observational-equivalent with the example given in Section 3. The `computation` is split, for example, into three smaller parts (`computation1`, `computation2`, and `computation3`) that, put in sequence, form the original computation specified in the model. After each small computation, a certain amount of time delay follows (`deadline1`, `deadline2`, and `deadline3`) and the sum of all delays makes the original delay amount ($deadline = deadline_1 + deadline_2 + deadline_3$). A timed trace of this system is given in Figure 14.11.

The two systems modelled are, obviously, observational-equivalent for a user for whom it is important when the input data `x` is read from the environment, what is the flow of computations performed on `x`, and when the final result `y` is available. For this system, the existing synthesis mechanism for POOSL models, which relies on the ϵ-closeness between traces for the properties-preservation, can be applied. To obtain an implementation trace ϵ-close to its corresponding trace in the model, as shown in the previous section, a synchronisation of each moment in the model time when an action happens with the corresponding physical time, up to ϵ, is realised, as shown in Figure 14.12. For the implementation of the original model, there are only two synchronisation points, t_1 and t_2 from Figure 14.9, and the time-deviation is

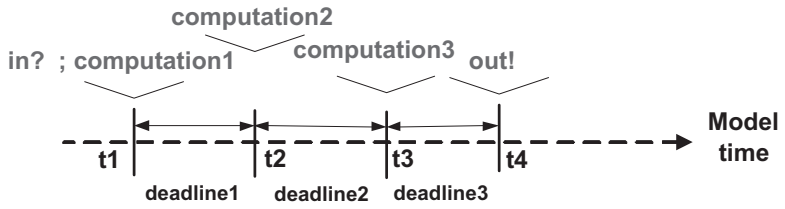


Fig. 14.11 Observational-equivalent model timed trace

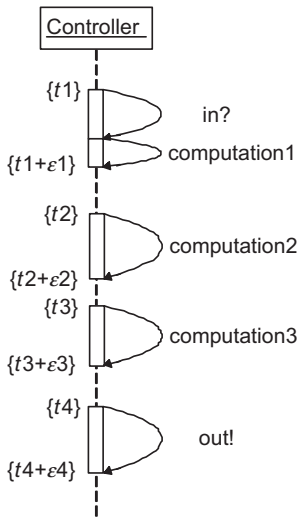


Fig. 14.12 Implementation of the equivalent model in physical time

large. For the observational-equivalent model, in Figure 14.12, there are four synchronisation points, t_1, t_2, t_3, and t_4, and the time-deviation for each of them is smaller. Therefore, over the whole system, the properties are stronger preserved in the realisation of the second model.

From the perspective of the code generation tool, looking at Figure 14.12, what it actually has to do is to generate a trace in which the execution of the `computation` (made of `computation1`, `computation2`, and `computation3`) starts immediately after reading `x` from the environment, continues more or less without stopping, and finishes before the moment the result `y` must be given back to the environment. In other words, `deadline` represents the *deadline* of the computation, and only the observable actions of the system are synchronised in the physical time as depicted in Figure 14.13. The time needed for the execution of computation does not have to count against the size of the time-deviation between model and implementation because, for the observational-equivalent model in Figure 14.12, the value of ϵ is small and it is determined by the execution time of `out!output(y)`.

As shown in this simple example, the implementation of a model containing time-intensive computations can be generated from an equivalent model that has the same observational behaviour and the same properties as the original one. Under these circumstances, we can define *actions* and *computations* slightly different than at the beginning of this section. We name *actions* those activities that can be observed by a user interacting with the system and,

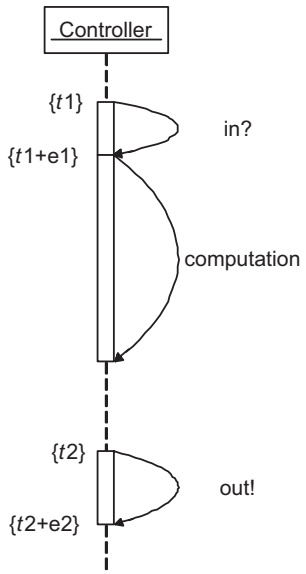


Fig. 14.13 A possible execution that still preserves the properties

therefore, their moments of execution in the model time must be synchronised with the physical time. On the other hand, *computations* are the internal activities of a system that a user cannot observe and who need to be scheduled for execution such that they can meet their deadlines. Moreover, if they are still running, they can be pre-empted by an action whose model time must be synchronised with the physical time. By abstracting from the internal actions of the model and synchronising the model time with the physical time only for the moments when observable actions happen, the observable properties of a model can be preserved; thus the code generation tool can handle the model synthesis of data-oriented applications as well.

However, it is not always possible to execute a computation within a *deadline*. For a specification like the one given in Listing 14.3, according to the formal semantics of the language, the urgent message can arrive either before the execution of `computation(x)(y)` or during the **`delay`**, which means after the execution of `computation` has finished. If the `computation` has a `deadline` in the implementation, then, at the time the `urgentMessage` appears, the execution of `computation` must be pre-empted. The `computation` will not be allowed to continue; thus the state in which the system realisation will be in that moment will not be a state present in the model. In this case, the relaxation of the timing constraints is not possible because there is no equivalence relation between this model and another one that has the computation split into smaller parts. Therefore, the execution time of the computation

Listing 14.3 Example of model without observational equivalence

```
1 abort
2   (computation(x)(y); delay deadline)
3 with p?urgentMessage;
```

contributes to the total time-deviation between the model and the implementation of this system.

Nevertheless, the mechanism that we propose in this chapter for the synthesis of real-time systems with time-intensive computations has the benefit of using an existing methodology, without changing the syntax, the semantics of the modelling language, or anything else, just by relaxing the constraints on the properties to be preserved. However, work needs to be done to formalise these ideas and to mathematically prove them. Moreover, a mechanism of identification of the observational-equivalent system whose implementation is the same with the one of the given model is required.

To analyse a model with different kinds of activities (taking longer or shorter execution time), the Y-chart scheme can be used again. Such a unified model helps designers in reasoning about aspects like what is the largest time-deviation (ϵ) that the system can allow, or what is an appropriate scheduling of the time-intensive computations, as shown in (Florescu et al., 2004a).

6. Conclusions and Future Work

To achieve a predictable design of real-time embedded systems, a unified executable model, capturing both functional and timing aspects of a system, is suitable to allow engineers to reason about different properties in a unified manner. Moreover, such a model must be easily refinable towards a complete system specification from which the implementation can be automatically obtained.

In this chapter, we have presented how the Software/Hardware Engineering methodology can be used for the modelling, analysis, and synthesis of a large area of real-time systems (control-oriented, data-oriented applications). The POOSL modelling language allows specification of both timing and functional aspects of systems, whereas the ϵ-hypothesis guarantees the preservation of properties between two timed systems with a small time-deviation. By satisfying the ϵ-hypothesis, the code generation tool, Rotalumis, succeeds in synthesising an implementation of a model preserving *all* the properties, in case the actions specified are not time-consuming. For the data-oriented applications, we propose a way to generate the realisation from a model that is observational- equivalent with the original one but which has the advantage that the time-deviation obtained for it is smaller. In fact, we suggest that it is

possible to make an abstraction from the internal actions of the system and synchronise the physical time with the model time only for the observable actions. Moreover, this realisation would preserve the observable properties of the original real-time system.

For the future research, we aim at formalising this mechanism and at giving a mathematical definition for the circumstances when computations can be safely pre-empted by actions. Furthermore, we want to adapt the code generation tool to work according to the proposed mechanism and to apply it to realistic case studies.

Acknowledgments

This work is being carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

References

Amnell, Tobias, Fersman, Elena, Mokrushin, Leonid, Pettersson, Paul, and Yi, Wang (2003) TIMES: a tool for schedulability analysis and code generation of real-time systems. In: *1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS) 2003*.

Buttazzo, Giorgio (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, MA.

Florescu, Oana, Voeten, Jeroen, and Corporaal, Henk (2004a) A unified model for analysis of real-time properties. In: *1st International Symposium on Leveraging Applications of Formal Methods (ISoLa) 2004*.

Florescu, Oana, Voeten, Jeroen, Huang, Jinfeng, and Corporaal, Henk (2004b) Error estimation in model-driven development for real-time software. In: *Forum on Specification & Design Languages (FDL) 2004*.

Geilen, Marc (2002) Formal techniques for verification of complex real-time systems. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Huang, Jinfeng, Voeten, Jeroen, and Geilen, Marc (2003a) Real-time property preservation in approximations of timed systems. In: *1st ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2003)*.

Huang, Jinfeng, Voeten, Jeroen, Ventevogel, Andre, and van Bokhoven, Leo (2003b) Platform-independent design for embedded real-time systems. In: *Forum on Specification & Design Languages (FDL) 2003*.

Huang, Jinfeng, Voeten, Jeroen, Florescu, Oana, van der Putten, Piet, and Corporaal, Henk (2005) *Advances in Design and Specification Languages for SoCs (Best of FDL'04)*, chapter Predictability in real-time system development. Kluwer Academic Publishers, Boston, MA.

Miller, Joaquin, Mukerji, Jishnu, et al. (2001) Model driven architecture (MDA. Technical report, OMG document ormsc/2001-07-01, Object Management Group (OMG), Needham, MA.

Milner, Robin (1989) *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

OMG (2003) *OMG Unified Modeling Language (UML) Specification—Version 1.5*. OMG document formal/2003-03-01, Object Management Group (OMG), Needham, MA.

OMG (2005) *UML Profile for Schedulability, Performance, and Time Specification—Version 1.1*. OMG document formal/2005-01-02, Object Management Group (OMG), Needham, MA.

Selic, Bran and Motus, Leo (2003) Using models in real-time software design. *IEEE Control Systems Magazine*, 23(3).

van Bokhoven, Leo (2002) Constructive tool design for formal languages: from semantics to executing models. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

van der Putten, Piet and Voeten, Jeroen (1997) Specification of reactive hardware/software systems. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Chapter 15

Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering

Lossan Bondé, Pierre Boulet, and Jean-Luc Dekeyser
LIFL – Laboratoire d'Informatique Fondamentale de Lille, Equipe West
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cédex
FRANCE

Abstract Model-driven engineering (MDE) is an emerging approach of software design where the whole process of design and implementation is worked out around models. With MDE, a system is built by designing a set of models at different levels of abstraction. At the first level, only the main functionalities of the system are modeled. This first model is called according to the model-driven architecture (MDA) terminology as the platform-independent model (PIM). This PIM can be projected into one or more other models by transformations. These latter models are at lower levels of abstraction. When a model at a given level of abstraction integrates some platform (technology) information, it is called a platform-specific model (PSM). Model transformation is therefore a key issue of the MDE approach. However, many questions arise about transformations. Among these questions is: *When a model is transformed into different other models on different platforms, how to ensure the interoperability between these models?*

This chapter aims to provide an answer to the above question. Our approach is based on a traceability model. This model not only keeps links between the source and target model elements but also records the different operations that where performed in the transformation. We present a methodology for the automatic generation of the traceability model and the exploitation of this model to ensure interoperability. An example based on open core protocol (OCP) is provided to illustrate our proposal.

Keywords: MDA; MDE; model transformations; traceability; interoperability.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 263–276.

1. Introduction

Embedded system designers need to model both applications and hardware architectures. 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. The design flow of embedded systems implies the handling of many models at different levels of abstraction. The new methodologies (approaches) model-driven engineering (MDE) and model-driven architecture (MDA; Kleppe et al., 2003; OMG, 2003) are very useful in such a context (see Oliver, 2005 and Bondé et al., 2005). In many cases, one has to build simulations using components image processings (IPs) at different abstraction levels. Such simulations raise some interoperability problems between abstraction levels.

In MDE the whole process of design and implementation is worked out around models. In this vision, the model transformations play an important part. As the methodology gets mature, model transformations will involve more and more models from different metamodels at different levels of abstraction. Using MDE methodologies, we can transform models at different levels of abstraction. But the question of interoperability between different models derived from transformation remains without answer.

In this chapter we introduce a new approach of solving the interoperability problem by using a traceability model. Traceability in software modeling can be defined as the ability to trace (follow) the different model elements from the design step down to the implementation.

The chapter is organized in five sections. Section 2 we presents the metamodel for a trace model. Section 3 explains how to generate automatically the trace model. Section 4 explains how trace information can be used for interoperability bridge generation. And Section 5 concludes and discusses some perspectives and further work.

2. Metamodel for Traceability in Model Transformations

In the current state of the art, traceability is achieved manually with great efforts. Many research works are done on traceability, but most of them address the question in the area of tracing requirements, system code, and documentation (see Antoniol et al., 2000; Champeau and Rochefort, 2003; Egyed, 2001, 2002; Zamfiroiu and Prat, 2001). We believe that traceability can be defined in a more general way by a metamodel. This metamodel can then be used in model transformation to produce trace information. And the trace information can be used to solve interoperability problem.

The subject of traceability is of great interest in MDA transformations, but when it comes to defining what is to be traced, not so much is said. So before defining a metamodel for traceability, we will first define what information we

expect to get from a trace model or what kind of information about transformation we want to keep.

2.1 Concepts and Overview of the Metamodel

A good traceability model should be complete enough to record the necessary trace information. But a too complex model is difficult to exploit. To solve this trade-off between completeness and simplicity, we think that the design of such a model should be driven by the envisioned purpose of the trace model. The following are the requirements which form the basis of the traceability metamodel (TM) we suggest in this paper:

Requirement 1: A TM should be used to establish or maintain consistency between the heterogeneous models used in an MDE-based design flow.

Requirement 2: In a context of model transformation, a TM should be designed in such a way that it would be automatically generated along with the transformation itself.

To fulfill requirement 1, a traceability model should be able to find out all the elements in a model created by transformations that relate to a given element in the source model, so that when this latter is modified we can either automatically or manually propagate the modifications in the related models. It is therefore necessary during the transformation process to create links between the target model elements and all the source model elements implied in the transformation. We have captured this kind of information under the *Relationship* concept.

The *Relationship* concept simply records the dependencies between the source model elements and the target model elements, and nothing more. So if we intend to have any kind of automatic model update we need also to remember the different operations that we performed on the input model to produce the output one. Another reason that motivates the need for keeping records of operations is that, sometimes, transformations can be semiautomatic. We mean by semiautomatic a transformation which is composed of an automatic step followed by some manual tunings. In this kind of transformations, it is desirable to avoid as much as possible the repetition of the manual tunings. Keeping the operations done in the transformation can help to update the target model without replaying the transformation that could overwrite the initial manual tunings. To model the operations used in a transformation, we use the *TraceOperation*.

So far we considered that a traceability information (trace) is made of two elements: the *Relationship* and the *TraceOperation*. We use the *TraceElement* concept to represent this basic trace information. The whole set of trace

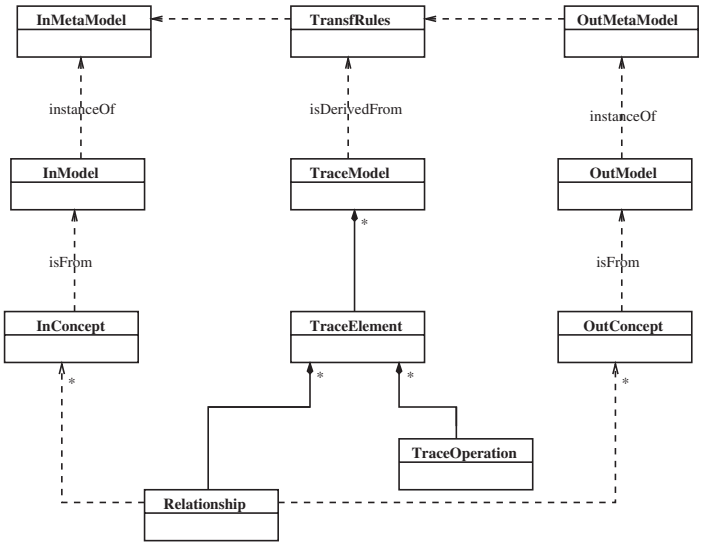


Fig. 15.1 Overview of the metamodel

information associated to a given transformation is captured under the *TraceModel* concept. A *TraceModel* is thus made of a set of *TraceElement*s.

According to our requirement 2, a traceability model should be automatically derived from the transformation. To make this possible there should be a kind of dependency between the transformation rules and the trace model. This dependency is shown in the metamodel (Figure 15.1) by the dependency relation between *TraceModel* and *TransfRules*. The issue of automatic trace model generation will be further discussed in Section 3.

The overview of the metamodel is presented in Figure 15.1, in which we have left out the attributes of the different elements in order to keep it simple.

2.2 More Details on the Metamodel

The most simple operation one can get in a model transformation is to copy the attribute of an input element to a given attribute of the output element, the two attributes being of the same type. We have modeled this simple operation using the *Copy* concept. This concept has two attributes which refer to the source attribute and the target attribute. The second kind of transformation operation is to take an input attribute of a given type and to convert it to an attribute of another type in the output element. We call such an operation a *Convert* operation. Some other less direct transformation operations take as input several input attributes to create an output attribute. The input attributes can be from one or more input elements. We refer to this type of operation

by the *Transform* concept. All the operations described so far deal with the attributes; we have generalized them under the *AttributeOperation* concept.

It is not realistic to think that model transformation can always be performed in one step. In some cases, it can be necessary to do it in many steps. For example a first step creates the output concepts (elements) and a second step creates the links or relationships between the previous created elements. In this situation, we should be able to represent operations consisting in linking the output elements. The *Link* concept was introduced in our metamodel to fit this need. The *Link* concept has only one attribute holding the list of the objects to be linked. Finally, in a model transformation, it is very possible to encounter a situation where there are some output concepts that are not directly related to some input concepts, but are rather related to other output concepts. This is likely possible in transformations where the metamodels involved are too different from each other. To record the creation of such elements we use the *Create* concept. The *Link* and *Create* operations are related to concept management, we have therefore gathered them under the *ConceptOperation* concept.

In short, a trace operation is either an operation that deals with attributes, it is in that case an *AttributeOperation*, or an operation that handles model elements, then it is classified into *ConceptOperation*.

The *TraceOperation* concept can be associated to an *ImplementationClass* which actually performs the operation. By doing so we make it possible for the user to provide his own trace operations. If no *ImplementationClass* is provided, it is up to the engine to perform the operation.

The detailed description of the proposed metamodel is given in Figure 15.2.

3. Generation of the Trace Model

In our view of model transformation, a transformation is specified through a set of rules. The model transformation engine used in this work is ModTransf (Dumoulin, 2004). ModTransf takes one or more models and one or more set of rules as input, and it produces one or more models as output. When an object (model or model element) is submitted to the engine for transformation, ModTransf looks for the appropriate rule matching the source elements and applies the rule. The rule creates target elements and fills them with the source element properties. The rule can call the engine with nested elements allowing the call of rules defined for other elements. ModTransf rules are eXtensible markup language (XML)-based. A rule is made of *leftConditions*, *rightConditions*, 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 actions specify how the output concepts are created and updated. More details on ModTransf and its XML rules formalism can be obtained in Dumoulin (2004).

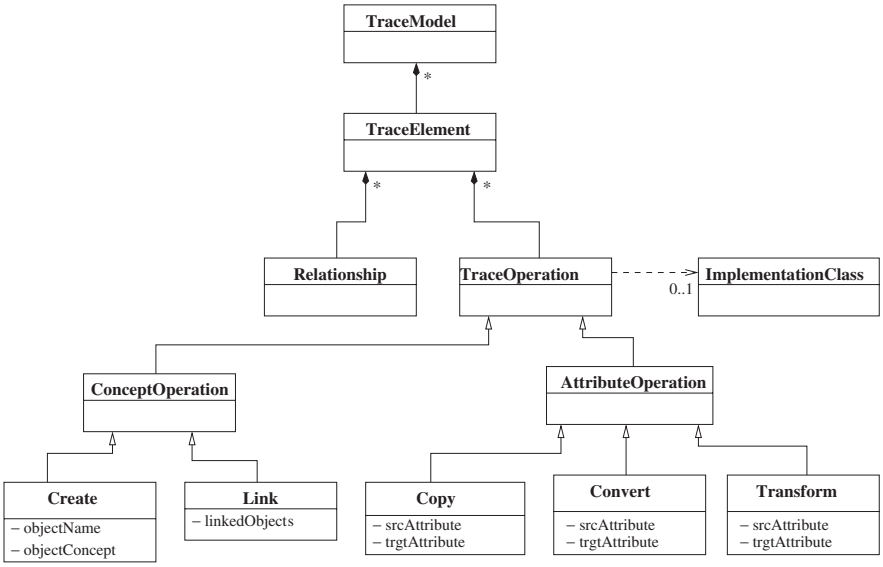


Fig. 15.2 Detailed metamodel

3.1 Principle of *TraceModel* Generation

As stated before, a transformation is the application of a set of transformation rules on input model to produce output model. When a transformation rule is executed by the transformation engine, some output model elements are created from some input model elements. The idea in our approach for trace model generation is: on completion of the application of a transformation rule, we ask the engine to perform the necessary actions to produce the trace information. These actions are:

- Record the link between the concerned input concepts and the output concepts. To do so, we use the elements in the *leftConditions* and *rightConditions* of the rule being applied to create the relationship of the trace element.

- Record the operations that actually perform the transformation. From the action parts of the ModTransf rule, we create the *TraceOperation* elements of the trace element.

At the beginning of the transformation process, we create an empty trace model, and each time a transformation rule is applied, we create a *TraceElement* and add it to the model. So when the transformation process is completed, the whole trace information is also generated as well.

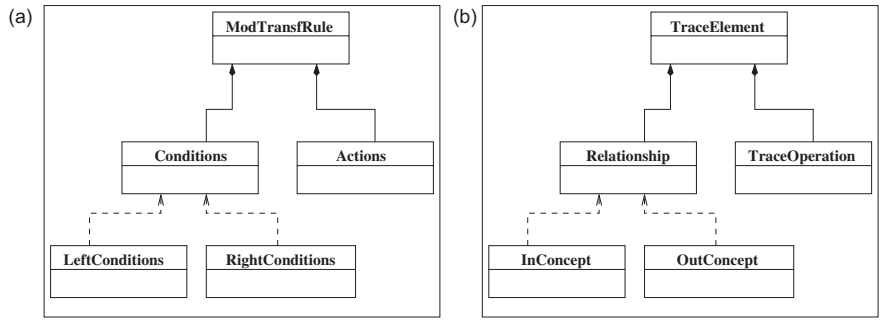


Fig. 15.3 Mapping ModTransf rules to trace element

We can observe from Figure 15.3 how straight it is to map a *ModTransfRule* to a *TraceElement*, even though the trace model is not designed only for ModTransf. In fact Figure 15.3a shows the structure of a ModTransf transformation rule whereas Figure 15.3b presents the structure of a *TraceElement* as described in Section 2.

3.2 Example

To illustrate our talk, let us consider a simple transformation where a unified modeling language *(UML) Class* concept is translated into a *Java Class*. We will not give here the UML and Java metamodels, they can be found in many relevant documentations. The example is sketched out in Figure 15.4. The *UCustomer* class is transformed into a *JCustomer* class. In this transformation, the attributes of the UML class are copied to fill the JCustomer class attributes. If we suppose that there is a Java class called *UAttribute2JAttribute*, which, given an attribute and its type, can create the necessary *getter* and *setter* for the corresponding Java class attribute, then we can set the *UAttribute2JAttribute* as the *ImplementationClass* for the *Copy* operation used for the trace operation. The ModTransf rules used to perform the transformation are as follows:

```
1  <rule ruleName="class">
2    <description> Transform a class to a class</description>
3    <leftConditions>
4      <concept type="Core.Class" model="uml" use="required"/>
5    </leftConditions>
6    <rightConditions>
7      <concept type="JavaClass" model="java"/>
8    </rightConditions>
9    <rightCreates extendsConditions="true"/>
10   <actions>
11     <copyPrimitive actionName="name" leftProperty="name" ↵
           rightProperty="name"/>
```

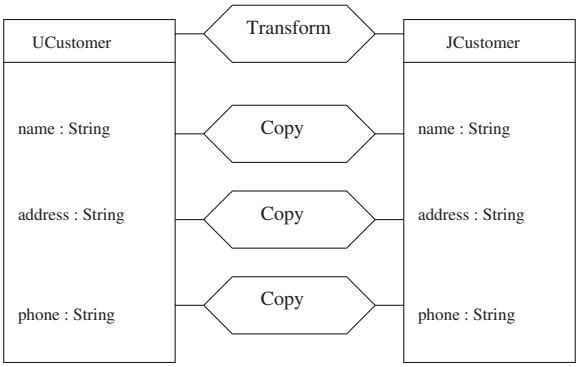


Fig. 15.4 Simple UML class translated into a Java class

```
12      <transform actionName="ownedElements" use="optional">
13        <left>
14          <expr expr="feature"/>
15        </left>
16        <right>
17          <switch>
18            <case type="Attribute">
19              <property name="attributes"/>
20            </case>
21            <case type="Method">
22              <property name="methods"/>
23            </case>
24          </switch>
25        </right>
26      </transform>
27    </actions>
28  </rule>
29  <rule ruleName="attribute">
30    <description>Transform a class attribute</description>
31    <leftConditions>
32      <concept type="Core.Attribute" model="uml"/>
33    </leftConditions>
34    <rightConditions>
35      <concept type="Attribute" model="java"/>
36    </rightConditions>
37    <rightCreates extendsConditions="true"/>
38    <actions>
39      <copyPrimitive actionName="name" leftProperty="name" ↵
            rightProperty="name"/>
40    </actions>
41  </rule>
```

The first rule states that a UML class is transformed into a Java class. The *transform* action of this rule indicates that when a feature of the class (attribute or method) is encountered, the engine will be called back with the input concept being the current feature. In case the feature type is *Attribute*, the second rule in charge of transforming an attribute will be executed. In this example of transformation, the first rule will be called only one time and the second rule three times (one time for each attribute).

At the end of this transformation, the associated trace model corresponding to this transformation is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<TraceModel name="SimpleExample">
  <TraceElement>
    <Relationship srcObjects="UCustomer" ↵
        trgtObjects="JCustomer"/>
    <TraceOperation>
      <Copy srcAttribute="name" trgtAttribute="name"/>
    </TraceOperation>
    <TraceOperation>
      <Copy srcAttribute="address" trgtAttribute="address"/>
    </TraceOperation>
    <TraceOperation>
      <Copy srcAttribute="phone" trgtAttribute="phone"/>
    </TraceOperation>
  </TraceElement>
</TraceModel>
```

4. Getting Interoperability from Traceability

The IEEE Standard Glossary of Software Engineering Terminology defines interoperability as *the ability of two or more systems or components to exchange information and to use the information that has been exchanged* (see Sanders and Hamilton, 2003). Some works related to systems interoperability can be found in Bencomo and Blair (2004) and Kleppe et al. (2003).

In this chapter we consider interoperability between two systems (at the platform-specific model (PSM) level) that are derived from one platform-independent model (PIM) model by model transformation. The two systems are interoperable if they can exchange information to fulfill the goal of the system under construction.

4.1 Proposed Approach for Interoperability

We propose to tackle the interoperability problem by concentrating on the communication between the system components. The idea behind is that, during the transformation process, the communication information between the different components are recorded in the trace model. For example, in a system where

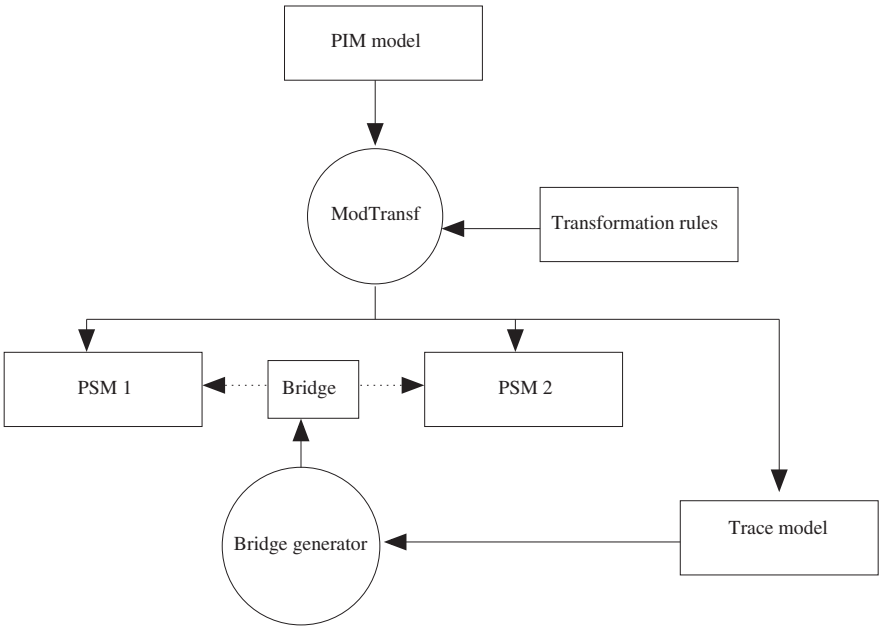


Fig. 15.5 Interoperability bridging from trace information

the different components communicate through ports that have interfaces of some types, when the system or the model is transformed, its components are transformed including the component ports. So if we trace the initial component ports, we can find out the communication scheme (model) in the new system. Once the new communication scheme is known, we generate a *Bridge* that will solve the possible incompatibilities between the new port interfaces. The *Bridge* generation as summarized in Figure 15.5 is done in three steps:

Step 1: generation of the trace model.

Step 2: analysis of the trace model to get the necessary conversions or adjustments for the bridging purpose.

Step 3: realisation of the bridge.

Using the ModTransf engine we are able to transform one PIM model into one or more PSM models. In Figure 15.5 the PIM model is transformed into PSM 1 and PSM 2. We have seen in Section 3 that alone with the transformation we generate the trace model. The first thing that the *Bridge Generator* does is to extract (retrieve) from the trace model the information related to ports and their interfaces. From this information, the new communication incompatibilities will be found out. These incompatibilities determine the functionalities that should be implemented by the *Bridge*.

The word *Bridge* in our proposal refers to a software component that has a set of communication ports and interfaces through which it can communicate with two or more other components. This software component also has the capacity of performing all the necessary type conversions to fit the compliance requirements of the communication between the different components.

Given the information from the analysis of the trace model, the bridge generation can be performed by a simple program that, given the information about the ports of the different components and their types, will create an instance of a generic class *Bridge* with the appropriate conversion methods.

4.2 Application of the Approach on an Example

We now show an application of our approach in a simple example. The example presents a PIM model that is transformed into two PSM models, and shows how the interoperability between the two PSM models is realized. Figure 15.6 presents the initial system, the result of the transformation, and the bridge between the two PSMs.

Source PIM model. At the PIM level, our system is made up of two components (*Comp1* and *Comp2*). Each component has a set of three ports, described as follows:

Port 1: It is a command port. Commands are *read* and *write*, 1 b is enough to encode the command information.

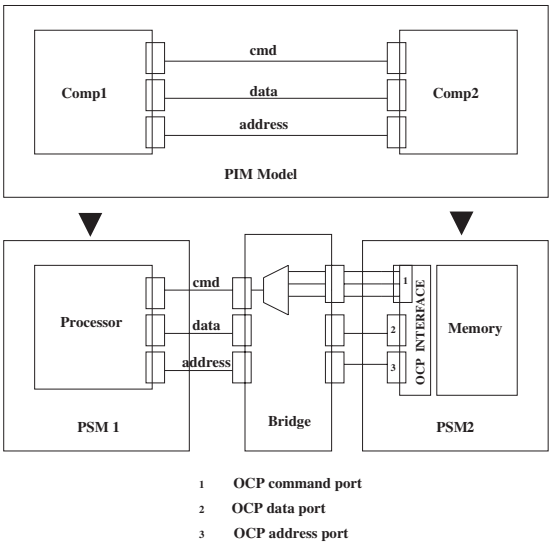


Fig. 15.6 A simple PIM model transformed into two PSM models

Port 2: It is a data port and the type of the associated interface is *Integer*. Thus the two components communicate by exchanging some integer values.

Port 3: It is an address port.

In this simple system, the communications are from *Comp1* to *Comp2*. A communication message is composed of three fields. The first field holds the command (read or write), the second contains the data. This field can be empty when the command is a read command. The last field corresponds to the address for reading or writing.

Transformation and target PSM models. The PIM model is transformed into two PSMs (TLM-PVT and TLM-CABA). For more information about the different levels of TLM, see Hardee and Colgan (2004).

The *Comp1* component is transformed into a *Processor* component in the PSM 1 model. The ports and associated interfaces of the *Processor* component are of the same type as those of the *Comp1* component in the source model.

The *Comp2* component is transformed into a *Memory* component in the PSM 2 model. This *Memory* component is associated to a bus having an OCP interface (see OCP, 2003, p. 13 and Haverinen et al., 2002). In the transformation, the ports of *Comp2* are mapped as follows:

Port 1 (command port) is mapped to the OCP command port. An OCP command is coded with 3 b with the following meanings: 000—Idle, 001—Write (WR), 010—Read (RD).

Port 2 (data port) is mapped to the OCP data port.

Port 3 (address port) is mapped to the OCP address port.

Bridging of the two PSM Models. Given that in the original PIM model, *Comp1* and *Comp2* were exchanging information, after the transformation of the initial PIM system into two PSMs we have to ensure that the two resultant models can communicate.

Let us name the three ports of the components p1, p2, and p3. From the trace model, we have the following information:

- Processor.p1 is derived from Comp1.p1, and their associated interfaces are of the same type.
- Processor.p2 is from Comp1.p2, and their interfaces are of the same type.
- Processor.p3 is from Comp1.p3, and their interfaces are of the same type.

We can see that in this example the communication protocol on the *Processor* side is the same as in the initial system.

If we assume that the OCP data and address ports are identical to p2 and p3, then the only problem to solve is to convert the 1 b information from port p1 (read/write command) of *Processor* to an OCP command coded in 3 b (see Figure 15.6).

If the OCP data or address port are not identical to the *Processor* ports p2 and p3, then we need some simple coding/decoding functions to transform the n bit information into p bits.

5. Conclusion

Building an embedded system implies dealing with many models at different levels of abstraction. A codesign environment requires methodologies and tools for model management.

In this chapter we have presented an approach to achieve interoperability in model transformations. We have defined a metamodel for traceability. With this metamodel we are able to generate automatically a traceability model during transformation. We have also shown how this traceability model can be used to generate a *Bridge* for interoperability. The approach has been put into action through an example based on OCP.

We believe that the work which is done here can be useful for other purposes:

- Our traceability model generated could be exploited to implement reversibility in model transformations.
- In a simulation environment, the trace model could be used to relate simulation information to the corresponding design or implementation elements.

We will address these issues in our future works.

References

Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. (2000) Information retrieval models for recovering traceability links between code and documentation. In: *International Conference on Software Maintenance (ICSM'00)*.

Bencomo, N. and Blair, G. (2004) Middleware unaware software development and interoperability using MDA. In: *Second European Workshop on Model Driven Architecture (MDA)*.

Bondé, L., Dumoulin, C., and Dekeyser, J.-L. (2005) Metamodels and MDA transformations for embedded systems. In: Boulet, P. (ed) *Advances in Design and Specification Languages for SoCs*, chapter 6. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Champeau, J. and Rochefort, E. (2003) Model engineering and traceability. In: *SIVOES-MDA workshop, UML 2003*. http://www-verimag.imag.fr/EVENTS/2003/SIVOES-MDA/Papers/Champeau.pdf.

Dumoulin, C. (2004) ModTransf: a model to model transformation engine. http://www.lifl.fr/west/modtransf/.

Egyed, A. (2001) A scenario-driven approach to tracability. In: *International Conference on Software Engineering*.

Egyed, A. (2002) Automating requirements traceability: beyond the record and replay paradigm. In: *17th IEEE International Conference on Automated Software Engineering (ASE)*.

Hardee, P. and Colgan, J. A. (2004) Advancing transaction level modeling (tlm): linking the OSCI and OCP-IP worlds at transaction level. http://www.opensystems-publishing.com/whitepapers/colgan_and_hardee/. White paper.

Haverinen, A., Leclercq, M., Weyrich, N., and Wingard, D. (2002) SystemC based SoC communication modeling for the OCP protocol. White paper v1.0, OCP, International Partnership. http://www.ocpip.org/data/ocpip_wp_SystemC_Communication_Modeling_2002.pdf.

Kleppe, A., Warmer, J., and Bast, W. (2003) *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, Boston, MA.

OCP (2003) *Open Core Protocol Specification 2.0*. OCP, International Partnership, http://www.ocpip.org/.

Oliver, I. (2005) Model based testing and refinement in MDA based development. In: Boulet, P. (ed) *Advances in Design and Specification Languages for SoCs*, chapter 7. Kluwer Academic Publishers, Dordrecht, The Netherlands.

OMG (2003) *The Model Driven Architecture*. Object Management Group (OMG), Inc. http://www.omg.org/mda/.

Sanders, P. A. and Hamilton, J. A. Jr. (2003) A process for interoperability. In: *Joint Command & Control Interoperability: Cutting the Gordian Knot*. Auburn University, Auburn, AL. http://www.eng.auburn.edu/users/hamilton/security/spawar/.

Zamfiroiu, M. and Prat, N. (2001) Traçabilité du processus de conception des systèmes d'information. *Ingénieurie des systèmes d'information*.

Chapter 16

Power Simulation of Communication Protocols with StateC

Luca Negri and Andrea Chiarini
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano
Italy

Abstract With the proliferation of battery-powered mobile devices relying on wireless ad hoc communication, low power consumption has become one of the primary goals in the design and implementation of communication protocols. We propose StateC, a power modeling and simulation flow well suited for high-level power modeling and optimization of communication protocols, wireless in particular. StateC can aid both in seeking power-optimized implementations and in devising optimal power management policies for existing devices. The flow relies on high-level UML statecharts models of the protocol stack and provides automatic tools for power characterization and for the generation of high-performance SystemC simulators based on such models. Preliminary application of StateC to Bluetooth and 802.11 exhibits good model accuracy (validation error below 1%) and high simulator performance. Although the flow has been targeted at wireless communication devices so far, its generality and use of widespread standards suggests its possible application to a generic hardware/software system.

Keywords: UML statecharts; power modeling; Bluetooth; IEEE 802.11.

1. Introduction

Mobile embedded devices relying on battery power and featuring wireless connectivity are becoming very popular. To cope with the power consumption issues that spoil their usability, many techniques have been addressed in the literature. These can be classified as *power-optimized designs* providing special low-power operational modes and *power management* techniques to fully exploit these modes.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 277–294.

For both strategies *power modeling* is a crucial aid. It can be performed at diverse levels of abstraction; however, low-level approaches such as transistor and gate level have proved too complex both when seeking optimal power management policies and for design space exploration in a preliminary design stage. For this reason, power management theory makes use of system-level power models (Bogliolo et al., 2004), which rely on a high-level architectural breakdown of the total power into contributions from different subsystems, where each can be in different power states.

However, most system-level power management strategies that have been proposed for wireless interfaces consider models with a limited number of power states and focus on a single subsystem, the radio interface card (Jones et al., 2001; Simunic et al., 2000). Such models are easy to build, but sometimes do not capture the full spectrum of operational states in the device. Moreover, many power optimization studies in the field of wireless networks are based on over-simplified power models (e.g., fixed ratio between transmission and reception (Ashok et al., 2003), between communication and processing (Toh et al., 2001), etc.).

In this chapter we propose a high-level power modeling and simulation flow called *StateC*, specifically aimed at communication protocols, wireless in particular. The flow is based on power models that are similar to the system-level ones, but relies on a *functional* rather than architectural breakdown of the power budget; that is the power contributions captured by the model are directly linked with protocol activity (connection setup, transmission, paging/beacon transmission, etc.) rather than with states of hardware components (on, idle, standby, etc.). Furthermore, compared with other functional models such as in Lattanzi et al. (2004), StateC leverages the syntax of UML statecharts to provide *concurrent* state machine models, well suited to model the (power) behavior of multilayer protocol stacks, possibly including the application layer.

StateC allows the user to quickly go from a behavioral specification of the system, given in *Statecharts* notation, to a fully-functional SystemC (Panda, 2001) power simulator. Other frameworks to convert UML or Harel Statecharts into C++ or SystemC have been proposed in Bjorklund et al. (2004) and in Nguyen et al. (2004); however, these are more complex than ours and aim at system *synthesis* rather than *simulation*. Other statecharts simulators, such as Matlab's StateFlow, are interpreted and pay a performance penalty. On the other hand, network-oriented simulators such as NS2 (VINT, 2006) and GloMoSim (Zeng et al., 1998) rely on proprietary formats to describe protocols. StateC combines the flexibility of graphical UML Statecharts models with the performance of a compiled SystemC simulator.

Section 2 presents the general StateC flow, whose three main phases are further described in Sections 3, 4, and 5. Section 6 presents some experimental

results for Bluetooth and 802.11 and Section 7 concludes the chapter with some remarks and future work.

2. The StateC Flow

The StateC flow can be used (i) for power management of existing communication devices, (ii) as power-optimized design aid, and (iii) for design-space exploration in the implementation of new communication protocols; this is illustrated in Figure 16.1 with three different flows. These flows share some phases and present some other distinct ones.

When StateC is used for *power management* (see flow (a) in Figure 16.1), the following steps should be followed:

- An *implementation-independent modeling* phase, with the purpose of creating a behavioral model of the communication protocol (or stack of protocols) in Statecharts notation. At this level, the model depends only on the chosen suite of protocols and can be adapted for different implementations.

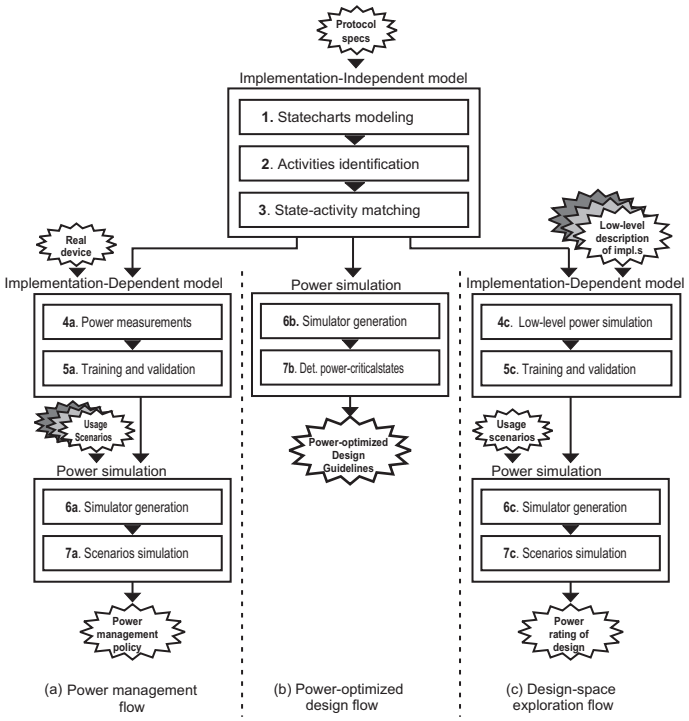


Fig. 16.1 The StateC power modeling and simulation flow

- A characterization of the model with the *implementation-dependent modeling* phase. This is performed either via experimental measurements on a chosen implementation or via low-level (e.g., very high speed integrated circuit (VHSIC) hardware description language (VHDL)) power simulation. At this stage, the model contains power figures that are specific to the tested device.

- The model can then be used for *power simulation* under different usage scenarios with the goal of devising optimal power management policies for the single nodes and for the whole network. For instance, given certain throughput or delay requirements, the simulation can help in finding the best tuning of the protocol's specific parameters (transmit power, beacon intervals, backoff rules, etc.) that satisfies the requirements while minimizing power consumption in the whole network or on selected nodes.

The same implementation-independent model can be reused and characterized for different implementations, e.g., to compare them via power simulation in selected scenarios. The other two possible usages we envision for StateC are:

- As an aid for *power-optimized design* (see flow (b) in Figure 16.1). StateC models can be used to improve existing designs. In this case, the protocol model is simulated at the implementation-independent level, providing precious information on which states of the protocol stack are worth optimizing, for instance, to grant a minimum device lifetime in a given scenario.

- For *design-space exploration* in the implementation of new communication protocols (see flow (c) in Figure 16.1). In this case the power characterization of the model is performed repeatedly on low-level descriptions (e.g., VHDL) of the candidate designs. Since the characterization procedure can be automated, flow (c) allows the powerwise comparison of the different designs under a typical usage scenario.

The implementation-independent modeling phase is the only one that requires human intervention; guidelines on how it should be carried out are given in Section 3. The implementation-dependent modeling can be fully automated for a given protocol stack, and is described in Section 4. Finally, for the power simulation phase, we have developed an *automated tool* that generates a SystemC power simulator starting from the Statecharts protocol model, whose full logic is described in Section 5.

3. Implementation-Independent Model

The implementation-independent modeling phase comprises three steps (also visible in Figure 16.1):

1. Represent protocol stack behavior with a set of finite state machines (FSMs). Different layers of the protocol stack should be modeled as concurrent FSMs that drive each other via events, using the *Statecharts* syntax.

2. Identify a set of *logical activities* (connection, transmission, etc.), which represent the sources of power consumption in the model.

3. Define a relationship between states and activities, i.e., define, for each state in the FSMs, which activities are triggered.

The result of steps 1–3 is a behavioral model of the protocol stack with some *unknown parameters*, namely the power consumption of the logical activities.

3.1 Statecharts Modeling of a Protocol Stack

Each layer of the protocol stack being analyzed is, in a real implementation, an independent entity, with its own notion of current state; interaction among layers occurs via primitive calls. In the same fashion, the Statecharts model of a stack comprising L layers is a *concurrent state* containing L substates.

Inside each layer, a number of *simple states* exist, that is only one of them is active at a time. The number and nature of such states is derived from protocol specifications. If in abundant number, states can be grouped in *composite states* according to the Statecharts syntax. Furthermore, additional concurrent states can be used within a layer to model separate threads, if this is suggested either by protocol specifications or by some preliminary experiments.

Intralayer transitions are controlled by triggering events and guard conditions, according to Statecharts syntax. Events can be classified as:

- *Internal* events from another layer of the stack, used for interlayer synchronization.

- *External* events from outside the stack; these are normally found on the transitions of the upmost layer in the stack, and are the means by which an application can control stack operation. Also, external events can originate in the *environment* around the device.

- *TIMEOUT* events, a particular category of events defined in the original Statecharts syntax by Harel (1987) and mirrored in unified modeling language (UML) with the *AFTER* keyword. A `TIMEOUT(base_event, time_units)` event fires automatically when `time_units` time units

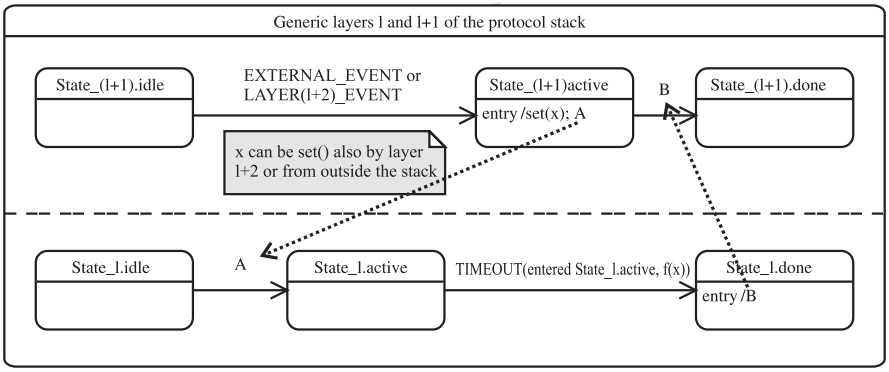


Fig. 16.2 Common pattern of communication between layers of the stack

have elapsed since `base_event` has occurred. The value of `time_units` can be (i) a constant value, (ii) a function of some variables of the Statechart, or (iii) a random number taken from a distribution that is function of some variables of the Statechart. A typical example of (iii) could be the time spent to successfully send a packet of a certain size, given the value of the model variable `BER`.

The Statecharts formalism makes concurrency possible by means of a broadcast communication mechanism. Despite this, in our models concurrency is normally handled using variations of a more restrictive communication pattern, shown in Figure 16.2, which basically simulates the call of primitives among layers that occur in a real implementation. In Figure 16.2, dotted arrows represent cause–effect relationships. Call of a primitive of layer l by layer $l + 1$: layer $l + 1$ sets some variable(s) x and fires event A, which unlocks layer l; when layer l is done it triggers B and returns to the caller, which continues execution. This is equivalent to $l + 1$ calling a primitive of l, which might actually happen when a primitive of $l + 1$ is called by some other entity (labeled `EXTERNAL_EVENT` or `LAYER(l+2)_EVENT`).

3.2 Logical Activities Identification and Localization

Let $\mathcal{S}$ be the set of all states in the model. Steps 2 and 3 in the implementation-independent modeling phase are the identification of a set $\mathcal{P}$ of N basic *logical activities* and their localization within the states. A logical activity is an operation that takes place (and consumes power) on a potential implementation of the protocol stack, and is represented in the power model by its power consumption p_i. It can directly map to architectural parts of the foreseen implementation (e.g., radio, baseband) or be purely logical (transmission, reception, scheduling, and so on).

The Statecharts syntax already has a concept of activity, which suits also the case of our logical activities, intended as sources of power consumption. In general, a state can use zero or more activities and an activity can be associated with one or more states; let $\rho \subset \mathcal{S} \times \mathcal{P}$ be a relationship capturing this association. As a consequence of this matching process each state in $\mathcal{S}$ either (i) consumes no power, (ii) consumes power directly by using some activity, or (iii) consumes power indirectly by activating, via events, other states that consume power directly.

To power-characterize a Statecharts model for a specific implementation means to assign a value to p_i ($0 \leq i \leq N$). Here p_0 is an additional activity called "Standby" that should be associated with the top-level state that contains the whole model. This models the standby power consumption on the device (i.e., with the device simply turned on).

4. Implementation-Dependent Model

The implementation-dependent modeling phase comprises two steps (see flows (a) and (c) in Figure 16.1):

4. Perform a series of experimental measurements on a real device or a series of power simulations on a low-level (e.g., VHDL) description of a possible implementation.[1] In each test, have the protocol stack perform a series of operations and record the total energy consumption. As the experiment is associated to a *path* in the Statecharts model, what is measured is a linear combination of the unknown activity power consumptions p_i.

5. Write a linear system associated with the whole set of experiments, having the activity power consumptions as unknowns and solve it with the least squares method.

The result of steps 4–5 is a device-specific Statecharts model with power consumption values assigned to the logical activities.

4.1 Model Characterization

An *experiment* consists of having a device under test (DUT) perform a series of communication tasks, and recording the total energy consumption. Let M be the number of these experiments. An experiment is associated to a path in the Statechart model. For a single layer of the stack, a *simple path* ϕ is an ordered

[1] From now on the experimental approach will be considered; however, mostly the same rules apply to the low-level power simulation one.

series of states and times (spent in the states). Formally:

$$\phi = (s_0, t_0), (s_1, t_1), \ldots, (s_k, t_k), \ldots, (s_K, t_K); 0 \leq k \leq K \tag{16.1}$$

where $s_k \in \mathcal{S}$, t_k is the time spent in state s_k, and K is said to be the *length* of path ϕ. Extending this concept to the whole model, a *composite path* Φ is a set of L simple paths $\phi_1, \phi_2, \ldots \phi_L$ (one simple path per layer, L being the total number of layers) with the additional condition that:

$$\sum_{t_k|(s_k,t_k)\in\phi_1} t_k = \sum_{t_k|(s_k,t_k)\in\phi_2} t_k = \cdots = \sum_{t_k|(s_k,t_k)\in\phi_L} t_k \tag{16.2}$$

Note that Equation 16.2 allows the simple paths of the different layers to have different lengths (in terms of number of states visited) but enforces the same *duration* in terms of total time.

Given an experiment j and the associated composite path Φ_j, it is possible to write an equation relating the energy consumption E_j measured experimentally with a linear combination of the activity power consumptions p_i ($i = 1, \ldots, N$) that represents the model prediction:

$$E_j = \sum_{i=0}^{N} p_i t_{ji} \tag{16.3}$$

where t_{ji} is a coefficient equal to the total usage time for activity p_i during test j; this is equal to the sum of the times spent in states that make use of p_i, according to ρ.

The number of experiments M must be $M > N$; in general, the higher the number of experiments compared to the number of activities, the more reliable the final model.

The calculation of the t_{ji} coefficients for a given experiment implies knowledge of the associated sequence of states and relative permanence times and is a key part of the characterization process. Depending on the protocol under examination, the available instrumentation and the target reliability of the model, the operation can be carried out in different ways. In certain cases (e.g., Bluetooth) times are well-defined in the specifications; in other cases (e.g., 802.11) random timers are involved, and either the use of average values or experimental tracing becomes necessary.

4.2 Training and Validating the Model

The final step to be carried out is the *training* of the linear power consumption model (given by the activity power consumptions p_is) using the experimental data. This can be accomplished by solving the system formed by Equation 16.3 for the M experiments, which can be conveniently expressed in matrix notation

as:

$$E = \mathbf{T} \times P \tag{16.4}$$

where E is the vector of the M total energy measurements, P is the vector of the N unknown activity power consumptions, and $\mathbf{T}$ is the $M \times N$ matrix of the t_{ji} coefficients. Since the system is overconstrained, it can be solved with the *least squares* method, which yields

$$P = (\mathbf{T}^T \times \mathbf{T})^{-1} \times \mathbf{T}^T \times E \tag{16.5}$$

If the Moore-Penrose pseudoinverse $(\mathbf{T}^T \times \mathbf{T})^{-1}$ is singular, either different activities must be chosen or different experiments be run, until all rows and columns in the matrix are linearly independent. *Validation* of the linear model can be performed using standard techniques such as cross validation or (leave one out (LOO) validation (Hassoun, 1995).

5. Power Simulation

The final stage of all flows is the execution of a number of simulations on the previously built models. This is visible in Figure 16.1 in all flows as "power simulation" and comprises two steps:

6. The *automated generation* of the SystemC power simulator starting from the Statecharts model.

7. The use of the simulator, which has different goals in the three flows of Figure 16.1.

5.1 Automatic Simulator Generation

The generation of the SystemC simulator given in the Statechart model is a fully automated procedure. This allows to easily build a simulator for any new protocol and to rapidly rebuild it after modifications to the protocol model.

Relying on UML allows the use of a broad range of graphical design tools, such as commercial, free, and open source; most of these tools allow to export the UML model in XML Model Interchange (XMI), a language coded using XML format. Alternative languages, such as GTDL (Jin et al., 2002) have been considered; however, XMI promises to become the de facto standard in the coming years for general model and metamodel communication purposes. As of date, we are working on the 1.2 XMI specification version (OMG, 2005).

The automatic translation of the XMI-coded model relies on a series of techniques originally developed in the web applications field. The syntactic and semantic rules for code translation are specified using the eXtendible stylesheet language transformations (XSLT) and XPath languages. The actual merge between XMI and the defined XSLT rules is carried out using an XSLT processor. The whole process is visualized in Figure 16.3.

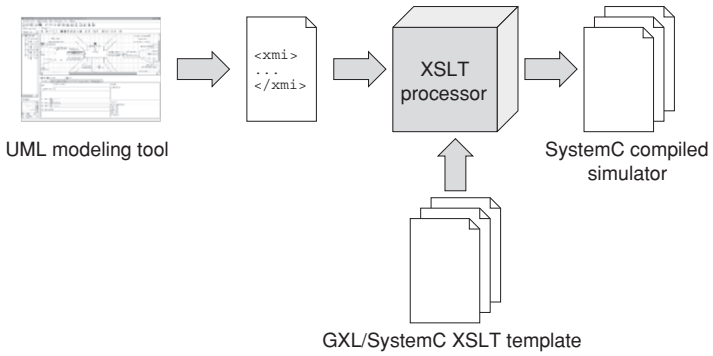


Fig. 16.3 Automated Statecharts to SystemC transformation

The SystemC production process relies on a series of consecutive *XSLT transformations*. The first conversion aims at simplifying the XMI code, which comprises large quantities of unnecessary information (e.g., graphical formatting notations) and produces an intermediate Statechart skeleton including only the necessary material. The intermediate format relies on another XML specification, graph eXchange language (GXL; Holt et al., 2000), which is sufficiently compact and easily readable. Even if GXL supports graph hierarchy, the model already at this stage is flattened out, as stack hierarchy is only indirectly present in the final SystemC code. The following XSLT passes analyze the code, parse all present variables (to be declared in the final C++ header), and then produce the header and main code file for a single SystemC module, which comprises the entire protocol stack. The Statecharts to SystemC conversion process follows these mapping rules:

Layers: Protocol hierarchy is eliminated in the final C++ code. The reasons behind this "monolithic" approach are the broadcast-style communication employed in Statecharts on one side and the scoping of events and variables in SystemC (more in general C++) on the other. The best solution we have found to cope with this misalignment is the implementation of the whole Statechart (i.e., all layers together) in the same class, which allows all events and variables to be visible to all states and transitions.

States: The translation phase is based on a *code template*, which represents a single Statechart *state* along with all its *outgoing transitions*. Replicating this template $card(\mathcal{S})$ times where $card(\mathcal{S})$ is the number of states in the whole model yields the complete simulator C++ code, ready for compilation. The template is the code of a SystemC execution *thread*, so the final simulator is actually made up of $card(\mathcal{S})$ threads executing

simultaneously (although all but the current-state threads, one per layer, will be in a waiting status).

Transitions: For each state, we make use of two SystemC events, one for state *entrance* and one for state *exit*. More specifically, we define $card(\mathcal{S})$ state entrance events, one for each state, and a single, common state exit event. The launching and catching of these two events causes one state to return in a waiting status and a new one to enter execution, thus simulating a state transition. This will be further explained.

Variables: Variables are considered global both in the Statecharts model and in the SystemC simulator and are mapped one-to-one.

Events: The Statecharts to SystemC event mapping is not symmetrical, due to the particular implementation of transitions.

In Figure 16.4, all fundamental parts of the state template are presented, including the transition mechanism:

1. The whole state can be seen as one infinite loop. After first activation (by the state's own entrance event) the state begins execution (2–5), eventually exits, and returns waiting for the next activation event.

2. Once a state is activated, place for state on-entry code is given. As example in Figure 16.4, when entering this state another transition is launched somewhere in the Statechart. This is done by setting a global string variable called `new_event`, unique within the entire class, and then launching the `exit_current_state` event. Active states will pick up the event and check the `new_event` variable to evaluate if they should undertake a transition.

3. After on-entry code execution, the state enters a loop to be exited only once when the `exit_current_state` event has been launched somewhere else and a transition has been chosen. In the case of timeout-controlled transitions, the amount of time to wait for (which is in general function of some variable) is first calculated and then added to the `wait()` call as an additional event.

4. When the internal `wait()` is activated, be it the `exit_current_state` event or a timeout, all possible outgoing transitions are evaluated. The global variable `new_event` is checked, and further guard conditions are considered. If a timeout transition is to be taken, `new_event` is expected to be found empty. If a valid transition is found it is taken, on-transition code is executed, and the destination state activation event is fired starting from the next simulation delta-cycle.

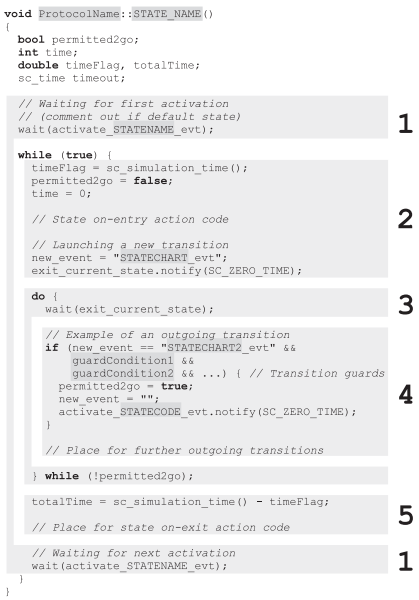


Fig. 16.4 State template for SystemC simulator. Dark gray shaded texts are the only parts that change from one state to the other

5. After the internal loop space for on-state-exit code is given. In our models we usually implement a function here to keep track of state permanence time based on the SystemC kernel time. This code is executed before the next state enters its on-entry code, as activation will be effective only after a delta-cycle.

5.2 Simulator Usage

The SystemC simulator traces the time spent in each state, for each layer of the model, and this information can be used for different purposes. According to Figure 16.1, it can be used to calculate the total power consumption of a given scenario for the power management and design-space exploration flows or simply to determine power critical states in the power-optimized design flow.

The simulation scenario is also to be implemented using Statecharts, as an additional layer above the stack of protocols, which represents *application* logic, to be translated and compiled together with the protocol stack model. This application layer controls the stack using its internal and external events; in this way, it can either control the upmost layer of the stack (which will in turn control the lower layers) or directly control any lower layer. As of date, only one device (i.e., one instance of the stack) can be simulated at the same time, and therefore the application layer must also include the behavior of the environment around the simulated device (using, for example, stochastic

TIMEOUT transitions). However, we are working on an extension of the simulator allowing multiple instances of a stack to be interconnected as SystemC modules and to exchange signals.

All variables used in the Statecharts are parsed using XSLT and listed in an external configuration file. Modifying this file causes the variables to assume different initial values upon simulation start. In this way the same compiled scenario (protocol stack plus application) can be used for parametric exploration without further recompilation cycles.

6. Experimental Results

This section presents the application of StateC to two different case studies, namely the *Bluetooth* (BT) and *WiFi 802.11* protocol stacks. Extracts of the implementation-independent models are given (Section 6.1), along with experimental validation figures (Section 6.2) and simulator performance considerations (Section 6.3).

6.1 Implementation-Independent Models

We have created implementation-independent Statecharts models of two protocol stacks so far: Bluetooth and 802.11. For Bluetooth, we presented a complete Statecharts model in Negri et al. (2004) in a simplified format (not including all states and events). Conversely, Figure 16.5 shows a part of the complete model in full Statecharts notation, including all events and guard conditions.[2]

The full statecharts model of Bluetooth comprises the three lowest layers of the stack: Radio, Baseband and Link Controller (LC); however, Figure 16.5 shows only the subset of states required by the BT *inquiry* procedure. The procedure consists of sending a burst of small packets on different frequencies to other (potentially listening) devices (Bluetooth SIG, 2003) and can be started by the application layer by setting the `Inq_TO` variable and firing the `LC_Inq_evt` (see LC layer in Figure 16.5). This triggers in turn the activation of the Baseband and Radio state machines. Power consumption is calculated by tracing the times spent in the states `LC_Inquiry`, `LC_Inq_Wait`, and `Radio_TX_Data`, where the logical activities INQ and TX are present (highlighted with bold arrows in Figure 16.5). In this case, INQ is active for the whole duration of the inquiry, whereas TX only contributes to the power consumption during each single packet and not in between packets.

For 802.11 (WiFi), we present here a small portion of the full Statecharts model as well; Figure 16.6 represents the procedure defined in the IEEE specifications for packet transmission using the distributed coordination function

[2]The complete model in full notation is too large to be presented here.

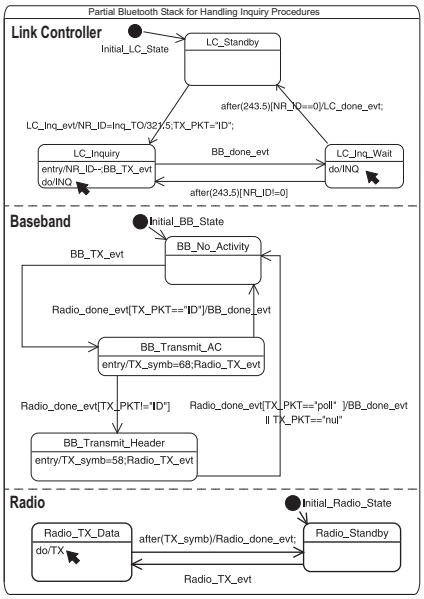


Fig. 16.5 Partial view of full Bluetooth Statecharts model; subset of states used for controlling inquiry procedures. Logical activities are highlighted with bold arrows

(DCF). This portion belongs to the second of the three layers we have defined for 802.11: one PHY layer, corresponding to physical radio packet transmission; a MAC sublayer, comprising the complex functions related to packet transmission and reception in a mainly asynchronous environment as that of WiFi; and a MAC Layer describing the main operation *modes*, as normal/power save mode, or ad hoc/managed mode, which influence the rest of the Statechart behavior.

The composite state in Figure 16.6, which can be reached from the central `Idle` state upon the launching of a `tx_pkt_evt` event, groups all substates linked with DCF transmission. It includes the pretransmission exponential back-off algorithm, the optional RTS/CTS virtual carrier sense handshake, and the post-transmission ACK packet reception. The states show how the Statechart syntax can handle random TIMEOUT transitions and interact with the underlying physical transmission layer. In this case no logical activity was directly linked with this layer, conveying instead energy consumption sources in the lower PHY state machine, not shown here.

6.2 Power Characterization of the Models

We have applied the implementation-dependent modeling phase to the BT model using two different Bluetooth modules. In both cases, characterization

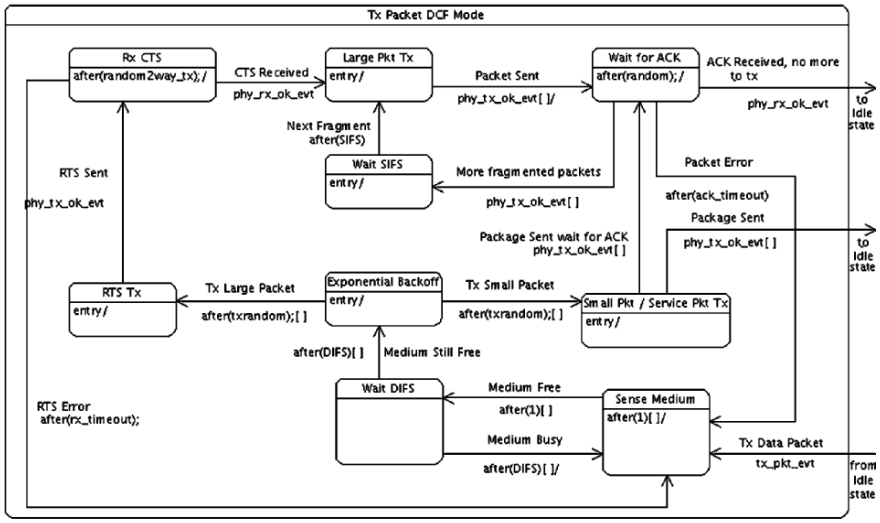


Fig. 16.6 Partial view of full 802.11 Statecharts model; subset of states used for packet transmission in distributed coordination function (DCF) mode.

was performed via experimental measurements and validation carried out with the LOO strategy (Hassoun, 1995). The results of this process are presented in Negri and Zanetti (2006), where we describe into detail the experimental testbench used to characterize the BT modules, the set of experiments performed and the outcoming model, which is made up of 12 power figures stemming from 12 logical activities. These include the INQ and TX activities presented in Section 6.1 and other activities that model the power contribution of different states in the whole model (connected as master, slave, disconnected for the LC layer; TX and RX for the Radio layer, and so on). The model is then validated achieving an RMS validation error on the predicted energy consumption of a generic task around 0.7% We consider this figure a very good result for a high-level power model.

We have recently completed the power characterization phase for actual 802.11 implementations, with similar results, namely RMS validation error just below 1%

6.3 Simulator Performance

We have started experimenting the power simulation phase as well. For simulation generation we have used a simple open-source XSLT processor, Sablotron (Ginger Alliance, 2006), which can easily be integrated in automated script procedures. In the resulting code, each state takes up about 50 to 100 lines of code (depending mainly on the number of outgoing transitions). Therefore,

given a model with n states, and with p additional states in the application layer controlling the model, the whole simulator will include at most approximately $(n + p) \cdot 100$ lines of code. Spatial complexity is thus linear with the number of states.

We ran preliminary performance tests on a P4 2.6 GHz laptop. We have verified that simulation time (temporal complexity) is linear with the number of events. The execution of simulations with a single node (module interface for simulating more than one interacting node is not yet implemented) was able to process roughly 2 million events per second, which is orders of magnitude faster than our previous StateFlow™ Bluetooth simulator for Matlab™. Direct comparisons with other network simulators such as NS2 (VINT, 2006) and GloMoSim (Zeng et al., 1998) are not possible due to their different scope and level of simulation detail.

7. Conclusions and Future Work

We have presented StateC, a power modeling and simulation flow, which can be used to define power management policies and as an aid in seeking power-optimized designs; in particular, we have presented its adoption in the case of communication-driven devices. The Statecharts modeling phase of the flow has been successfully applied to Bluetooth and 802.11, for which excerpts of the full models have been shown. Power characterization of the models has been performed via experimental measurements for the Bluetooth case, yielding a validation error below 1%; characterization of 802.11 devices produced similar results. The automatic SystemC simulator generation engine has been tested as well, and performance of the resulting simulators proved to be remarkable, especially when compared to existing (interpreted) Statecharts simulators. Ongoing and future work includes:

- On the modeling side, the full automation of the power characterization phase for Bluetooth modules and the finalization of the implementation-dependent model for 802.11 network interface cards (NICs)

- On the simulator generation side, the extension of the XSLT engine to handle composite states and internal transitions, and to allow simulation of multiple instances of a protocol stack at the same time

Finally, we believe the methodology could be easily reapplied in the near future to other emerging wireless stacks (e.g., Zigbee) and more in general to other hardware/software systems, where the concept of *layer* of the communication stack can be relaxed to a more general concept of *thread* running on the DUT.

References

Ashok, R. L., Duggirala, R., and Agrawal, D. P. (2003) Energy efficient bridge management policies for inter-piconet communication in bluetooth scatternets. In: *Proceedings of the IEEE Semiannual Vehicular Technology Conference (VTC2003-Fall)*, Orlando, FL.

Bjorklund, D., Lilius, J., and Porres, I. (2004) A unified approach to code generation from behavioral diagrams. In: *Languages for System Specification: Selected Contributions from FDL'03*. Kluwer Academic Publishers, Norwell, MA, pp. 21–34.

Bluetooth SIG (2003) *Bluetooth Core Specification v1.2*. Bluetooth Special Interest Group (SIG). `https://www.bluetooth.org/spec`.

Bogliolo, A., Benini, L., Lattanzi, E., and De Micheli, G. (2004) Specification and analysis of power-managed systems. *Proceedings of the IEEE*, 92(8):1308–1346.

Ginger Alliance (2006) *Sablotron: XSLT, DOM and XPath Processor*. Ginger Alliance. `http://www.gingerall.com/charlie/ga/xml/p_sab.xml`.

Harel, D. (1987) Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Hassoun, M. H. (1995) *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA.

Holt, R. C., Winter, A., and Schürr, A. (2000) GXL: toward a standard exchange format. In: *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE) 2000*. IEEE Computer Society, p. 162.

Jin, Y., Esser, R., and Janneck, J. W. (2002) Describing the syntax and semantics of uml statecharts in a heterogeneous modelling environment. In: *Proceedings of the Second International Conference on Theory and Application of Diagrams (DIAGRAMS) 2002*, Callaway Gardens, GA.

Jones, C. E., Sivalingam, K. M., Agrawal, P., and Chen, J. C. (2001) A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358.

Lattanzi, E., Acquaviva, A., and Bogliolo, A. (2004) Run-time software monitor of the power consumption of wireless network interface cards. In: *Proceedings of the 14th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS) 2004*. Springer, Santorini, Greece, pp. 352–361.

Negri, L. and Zanetti, D. (2006) Power/performance tradeoffs in bluetooth sensor networks. In: *Proceedings of the 39th Hawaii Conference on System Sciences (HICSS-39)*.

Negri, L., Sami, M., Macii, D., and Terranegra, A. (2004) Fsm-based power modeling of wireless protocols: the case of bluetooth. In: *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ACM Press, New York, pp. 369–374.

Nguyen, K. D., Sun, Z., Thiagarajan, P. S., and Wong, W. F. (2004) Model-driven SoC design via executable UML to systemC. In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS) 2004*.

OMG (2005) *MOF 2.0 / XMI Mapping Specification, v2.1*. Object Management Group (OMG). `http://www.omg.org/technology/documents/formal/xmi.htm`.

Panda, P. R. (2001) Systemc: a modeling platform supporting multiple design abstractions. In: *Proceedings of the 14th International Symposium on Systems Synthesis*. ACM Press, New York, pp. 75–80.

Simunic, T., Vikalo, H., Glynn, P., and De Micheli, G. (2000) Energy efficient design of portable wireless systems. In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED) 2000*, pp. 49–54.

Toh, C.-K., Cobb, H., and Scott, D. A. (2001) Performance evaluation of battery-life-aware routing schemes for wireless ad hoc networks. In: *Proceedings of the IEEE International Conference on Communications*, volume 9, pp. 2824–2829.

VINT (2006) *The Network Simulator ns-2*. The VINT Project. `http://www.isi.edu/nsnam/ns/`.

Zeng, X., Bagrodia, R., and Gerla, M. (1998) GloMoSim: a library for parallel simulation of large-scale wireless networks. In: *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, pp. 154–161.

Chapter 17

Integrating Model-Checking with UML-Based SoC Development

Establishing Consistency between Models

Peter Green[1] and Kinika Tasie-Amadi[2]

[1] *School of Electrical and Electronic Engineering*
University of Manchester
Manchester
United Kingdom

[2] *Solution Foundry Ltd.*
Park Farm Lodge
Upper Lambourn
Berkshire
United Kingdom

Abstract In order to address the complexities of system-on-chip (SoC) design, rigorous development methods and automated tools are required. This chapter presents an approach to formal verification using model-checking, designed for use in the context of a Unified Modeling Language (UML)-based SoC design flow. Different UML models can be automatically translated into Communicating Sequential Process (CSP) by the UML2CSP tool that has been developed in this work. The translated models can be checked for consistency with the model-checker Failures Divergences Refinement (FDR). Checks can also be carried out to determine whether models exhibit specified properties. The overall objective of the work is to ensure that only correct models are carried into the later stages of development where they are partitioned, and implementations are synthesized.

Keywords: UML; SoC; model-checking; formal verification.

A. Vachoux (ed.), Applications of Specification and Design Languages for SoCs – Selected papers from FDL 2005, 295–312.

1. Introduction

The transistor count of integrated circuits is rising faster than the ability of designers to exploit this increasing capacity, leading to the *design gap* (SEMATECH, 1999). It is widely believed that rigorous development methods and design automation will play a significant role in reducing this gap (Chang et al., 1999; Green et al., 2002). This chapter is concerned with applying formal verification via model-checking to unified modeling language (UML) models constructed in the context of a disciplined system-on-chip (SoC) design method. The method, hardware and software objects on chip (HASoC; Green et al., 2002), adopts an iterative, incremental approach, merging concepts from several methods (Morris et al., 1996; Booch et al., 1999; Chang et al., 1999). A key feature of HASoC is the creation of abstract object models, which are unpartitioned in the sense that their implementation in software or hardware is not fixed. The realization of objects in software or hardware is determined later in development on the basis of nonfunctional requirements.

The aim of this work is early-stage verification of UML models using model-checking techniques, so that only provably correct models are input into the partitioning and synthesis phases. Two broad areas are addressed. The first concerns the behavioral equivalence of different UML models. Most methods that use UML, including HASoC, require the construction of several models, e.g., use case, interaction, and class models. For large, complex systems, ensuring behavioral consistency between different models is difficult. The second area involves checking whether or not a system model displays desirable properties, and that undesirable properties, such as deadlock, are absent. The approach discussed in this chapter, known as model-checking for UML SoC models (MoCUS), uses similar techniques and automated tools to address both areas.

An overview of MoCUS is given in Section 2, followed by reviews of the chosen model-checker and its input language, previous attempts to couple model-checking with UML-based development, and relevant UML semantics. Sections 4 and 5 discuss aspects of the model translation process. The translation tool developed in this work is introduced in Section 6, and Section 7 discusses the checks that are applied to translated models. A partial case study is presented in Section 8, and conclusions are drawn in Section 9.

2. Overview of the Approach

The MoCUS approach can be used with use case, sequence diagram,[1] and class models. These models are developed in the usual way, and then the behaviors

[1]The complete set of sequence diagrams is termed the *Interactions Model* (IM).

of all classifiers (use cases and classes) are defined by UML state machines (USMs). The state-based use case descriptions and sequence diagrams are then automatically translated into the input language of a model-checker by the UML2CSP tool developed as part of this work. The UML2CSP tool also automatically builds the *composite object model* (COM). This contains the state machine representations of all the objects present in the sequence diagrams, obtained from the class state machines, with communications links defined by the message paths in the IM and output actions in the state machines. Once constructed, the COM is also translated into the input language of the chosen model-checker.

Consistency between models can be tested by checking sequence diagrams of use case scenarios against use case state machines, the requirement being that the sequences of transactions between the system and the actors in the environment are the same for both models. Sequence diagrams can be checked against the COM to ensure that the composition of object state machines can realize the behavior described by the sequence diagrams. Use cases can also be checked against the COM to verify that the system objects can implement the behavior implied by the specification.

Property checking is performed by defining desirable/undesirable properties as sequence diagrams, and checking them against the COM. This is particularly useful for investigating properties that the system must always (or never) display, irrespective of the use case that is executed. Examples of this type of property checking are given in Tasie-Amadi (2004). These techniques are not only restricted to the verification of functional behavior but can also be applied to communications behavior. In the HASoC method, an object model of the system's hardware platform is developed, with UML port state machines being used to describe the communications (bus) protocol. By creating a COM based on the platform port state machines, and sequence diagrams representing desirable/undesirable communications behavior, model-checking via the MoCUS approach can be applied.

MoCUS uses the model-checker FDR (failures divergences refinement checker), whose input language is CSP (communicating sequential processes). This combination was chosen because one type of FDR check, the refinement check, is in many cases, the check that is required to support the investigation of consistency between different UML models.

3. Background

This section provides an overview of CSP and FDR, and a brief review of previous works on applying model-checking to UML models. It is concluded by a brief recap of the semantics of USMs.

3.1 Overview of CSP and FDR

CSP and FDR support the specification and verification of concurrent systems. CSP specifications are based on two elements, namely *processes* and *events*. Processes are transition systems (Schneider, 2000), and so can be viewed as moving between states based upon the occurrence of atomic events. Processes can be constructed from other processes via composition operators and event-prefixing operator (->). An example of event prefixing is `u->V`, which represents a process that performs event `u` and then one of the event sequences (*traces*) of process `V`. Since CSP operators have formal definitions, the behavior of a composite process can be found from the behavior of its components and the operators used to aggregate them.

Concurrent processes interact via shared events, which occur when all the processes that share the event occupy states with outgoing transitions labelled by the event. Hence shared events act to synchronize processes. Communication between concurrent processes is achieved by one-way channels between processes, with the sending and receipt of a message being treated as a shared event.

Verification of CSP specifications makes use of a number of semantic models, the most basic of which is the *traces* model. Although the traces model can be used to answer questions about safety, more sophisticated models are required to deal with liveness issues (Schneider, 2000). The *failures* model is more powerful and can be used to investigate liveness for systems that do not experience divergence (or livelock). This model deals with traces and *refusals*, the latter being the set of events that a process can refuse to perform after executing the associated trace. For systems that can potentially diverge, then the most powerful and complex semantic model, the *failure-divergences* model, is required for verification.

The CSP *refinement* operation is important in this work. One process is said to be a refinement of another if it displays a subset of the other's behavior. Refinement can be considered with respect to traces, failures, etc.

The FDR tool (FSEL, 2003) supports the model-checking of machine-readable CSP specifications. It can be used to check for deadlock, divergence, and can perform refinement checks to determine if one process is a refinement of another. The use of FDR is discussed in Section 7.

3.2 Previous Approaches to the Checking of UML Models

Previous works on the application of model-checking to UML models have either used the SPIN or FDR model-checkers. Similar SPIN-based approaches are reported in Lilius and Paltor (1999) and Schäfer et al. (2001). Both are concerned with the automated translation and checking of interaction diagrams and class state machines. However, whilst the approach of Lilius and Paltor

(1999) addresses the issue of detecting errors such as deadlock in a network of state machines formed from a collaboration diagram and a set of class state machines, the approach in Schäfer et al. (2001) also investigates consistency between sequence diagrams and class state machines.

Engels et al. (2001) discusses a manual mapping between class state machines (without hierarchy or concurrency) and CSP. Translated models are checked with the FDR tool, and applications cited include checks for deadlock and for protocol consistency in UML-RT models. Other approaches that provide a mapping between USMs and CSP are reported in Ng and Butler (2002) and Ng et al. (2003). As with Engels et al. (2001), Ng and Butler (2002) only deal with flat state machines without concurrency, and map objects whose classes are described by state machines to CSP processes. Some of the mapping rules developed in Ng and Butler (2002) are similar to those used in this work. Ng et al. (2003) extends the rules of Ng and Butler (2002) to include hierarchy explicitly. This is found to be difficult and the mapping rules are subject to a number of restrictions.

CSP is used to specify the semantics of a subset of UML in Davies and Crighton (2002), and rules for translating classes, objects, state machines, and interactions into CSP are given. Only nonhierarchical state machines without orthogonal regions are considered, and sequence diagrams are totally ordered. Checks for consistency between sequence diagrams and class state machines, based on FDR refinement checks, are discussed.

It is believed that the work presented here represents a significant advance over previous works in terms of the range of UML models that can be included in the model-checking process. Relative to other work involving UML and CSP this work also supports a wider range of state machine features and provides an automated translation and checking capability.

3.3 UML State Machines

UML provides an extensive set of features to support state machine modeling, including hierarchical decomposition, concurrent regions, in-state activities, and actions associated with state entry/exit and also with transitions. The behavior of a USM is specified in terms of an abstract machine consisting of an event queue, a dispatcher, and an event processor (OMG, 2001)[2]. Events received by a USM are placed on its event queue. The dispatcher selects a queued event and sends it to the event processor, which makes a transition, generates actions, etc., if appropriate. The machine implements *run-to-completion semantics* (RTC) since the dispatcher only sends an event to the processor when it has completed the processing of the previous event.

[2]This work was based on the UML 1.4 standard. The used parts of UML have not changed in UML 2.0.

Many of the features of USMs are supported in MoCUS, including concurrency, hierarchy, and actions. Those that have not been incorporated into the UML2CSP tool include local variables, guards, and in-state activities. Hence the approach is restricted to checking the control-oriented behavior of a system.

Choice vertices are supported by the UML2CSP tool. However, the lack of guards means that the choice is nondeterministic. Nevertheless, since a model-checker exhaustively searches a model's state space, all paths originating from a choice vertex will be searched, and so all the possible consequences of the choice will be explored.

4. Translating State Machines to CSP

A key part of the MoCUS approach is the translation of USMs into CSP. The generic problems of translating USMs to CSP are considered in this section, and issues relevant only to the use case state machines or class state machines are presented in Sections 5.1 and 5.3.

As indicated above, UML provides an extensive set of constructs to support state machine modeling. However, the semantic gap between UML and CSP means that many of the features of state machines are difficult to map directly to CSP (Tasie-Amadi, 2004), although simple state machines may be readily represented. Consequently, USMs are translated to CSP in a two-stage process. First, they are flattened by recursively substituting child state machines for parent superstates (Figure 17.1) and also by the computation of product state machines to replace orthogonal regions (Figure 17.2).

After a state machine has been flattened, it must be mapped to CSP in such a way as to realize the semantics of the hypothetical machine specified in the UML standard, and briefly introduced in Section 3.2. Communications between state machines, or between a state machine and an actor, must also be implemented. These issues are discussed in Section 4.2.

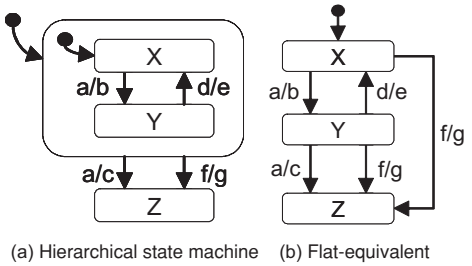


Fig. 17.1 Hierarchical state machine and its flat-equivalent

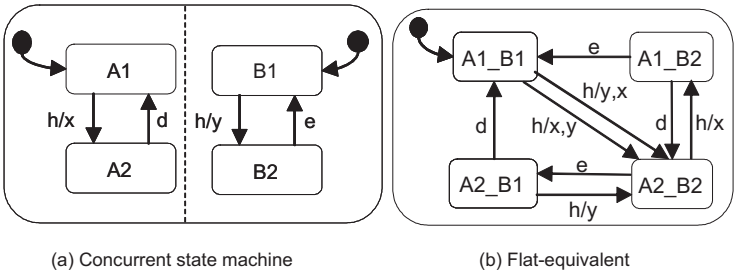


Fig. 17.2 Concurrent state machine and its flat-equivalent

4.1 Flattening State Machines

Figure 17.1 shows how the flattening algorithm used in this work reduces a hierarchical state machine to a flattened form. Note that the algorithm gives priority to inner transitions, in accordance with UML semantics (OMG, 2001), since there is no transition from state `X` to `Z` caused by event `a`. Implementing this semantics directly in CSP is extremely cumbersome (Ng et al., 2003; Tasie-Amadi, 2004).

The replacement of orthogonal regions by a product state machine is shown in Figure 17.2. This includes the case where simultaneous transitions can be caused in both regions by the same event (`h`). In the original state machine the ordering of the transition actions `x` and `y` is nondeterministic, and so two transitions triggered by `h` are included in the product state machine to represent this nondeterminism.

Issues concerning child state machines that contain final states must also be considered. When a child state machine enters a final state it triggers a completion event that causes a transition out of the parent state. Completion events are denoted as transitions with no event triggers (see Figure 17.3). UML semantics specify that a completion event must be handled in the RTC step that follows the one in which it was generated.

Flattening the hierarchical state machine in Figure 17.3a causes the action associated with the completion transition to be included with the transition from state `B`, preserving the order of actions implied by Figure 17.3a. Strictly speaking, the state machines in Figures 17.3a and 17.3b are not completely equivalent, since in the hierarchical machine action `c` takes place in the RTC step after `a/b` whereas in the flat state machine it takes place in the same RTC step. Nevertheless, the action traces of the two machines are identical and therefore the mapping is acceptable for model-checking purposes. Other issues associated with completion events are discussed in Tasie-Amadi (2004).

The key advantage of flattening USMs is that it simplifies their translation to CSP. A disadvantage is the increase in size of a USM's internal

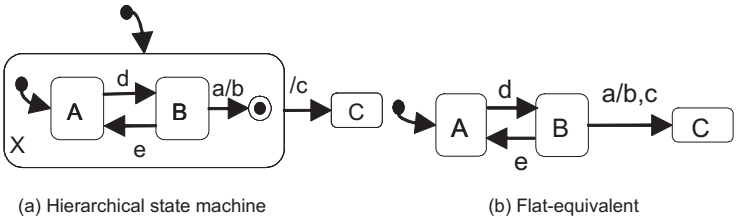


Fig. 17.3 Flattening and completion transitions

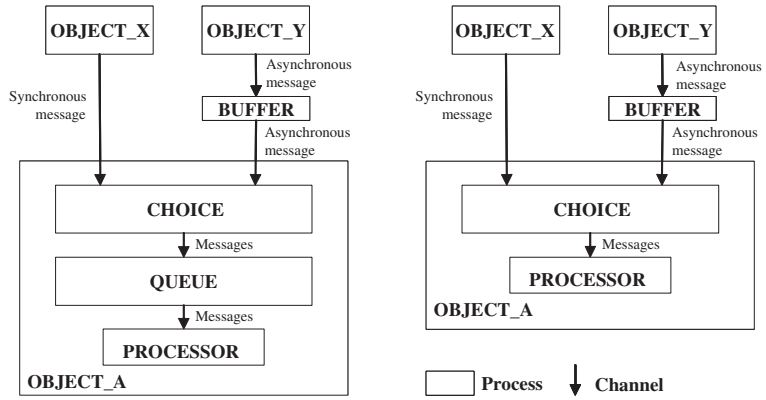
representation. However, this is unlikely to be a practical problem as use case state machines will typically represent behavior at a high level of abstraction, and well-designed individual class state machines are unlikely to have very large state spaces. In terms of model-checking, the size of the task depends on the number of *distinct* states, not the number of states appearing in the UML state diagram, and so is fixed regardless of whether or not the state machine has hierarchy and concurrency, or is flat.

4.2 Realizing State Machine Semantics in CSP

After flattening, USMs are translated into behaviorally equivalent CSP processes. The translation provides for communication between state machines, or state machines and actors, and both synchronous and asynchronous communication are supported. In the CSP specifications, all communication is performed over channels and synchronous sends are realized by following a channel send with a channel receive, causing the sender to await the return message. Asynchronous communications are implemented via the automatic insertion of buffer processes. The overall scheme for the realization of object state machines is shown in Figure 17.4a. This is the most general case, since, as discussed in Tasie-Amadi (2004), a slight simplification is possible for use case state machines.

Figure 17.4 shows that `OBJECT_A` receives synchronous messages from `OBJECT_X` and asynchronous messages from `OBJECT_Y` via a buffer. The box labeled "`CHOICE`" represents the CSP deterministic choice operator (`[]`), which allows a process (`OBJECT_A`) to await input from one of a number of channels, and to respond to the channel that is supplying input. If input is pending on several channels, then one is chosen nondeterministically and input from that channel takes place.

The representation of Figure 17.4a can be simplified by eliminating the event queue (Figure 17.4b). This is possible because the order that messages are dispatched from the queue to the processor is specified by OMG (2001) to be nondeterministic, and so this semantics can be realized directly by the CSP deterministic choice construct. This reduces the state space to be searched, and



(a) CSP representation (b) Simplified representation

Fig. 17.4 Object state machine and its simplified representation in CSP

so checking is much faster. Mappings that do and do not use event queues have been implemented, with identical results being obtained. However, the implementation that did not use event queues was two orders of magnitude faster than the one that did.

The event processor is represented by a CSP process (`STATE_MACHINE`), with a parameterized subprocess representing the states in the flat state machine (`STATE_MACHINE_STATE`). The structure of the processor is:

```
STATE_MACHINE = STATE_MACHINE_STATE(InitialState)
STATE_MACHINE_STATE(s) =
  [] ev:SetOfEventsFromState(s) @ P(ev) ->
            STATE_MACHINE_STATE(NextState(s, ev))
```

where `SetOfEventsFromState(s)` defines the set of events that cause transitions from state `s` and `NextState` is the transition relation. When `STATE_0MACHINE_STATE(s)` executes it awaits one of the events that can cause transitions from state `s` to arrive from the "`CHOICE`" construct in Figure 17.4. It then executes a process `P` based on the event `ev` that has been received. `P` implements the response of the state machine to the event that has been received (e.g., in terms of actions). The state machine then moves to the target state indicated by the transition relation.

The above specification is generic and is customized for a particular flat USM through the construction of the event sets, the `P` functions and the transition relation. Generic specifications are stored in a system of files associated with the UML2CSP tool, which constructs the state machine–specific elements from the flattened automaton.

5. Mapping the Models to CSP

Issues that must be addressed when translating each of the three UML models mentioned in Section 2 to CSP will now be considered.

5.1 Use Case Models

The *Use Case Model* (UCM) consists of use case diagrams and textual and state machine descriptions of each use case. In use case state machines, transition triggers are messages from actors, and actions represent either messages sent by the use case to actors or internal operations. Translation of the UCMs to CSP is accomplished via the approach described earlier. Certain additional features are supported by the UML2CSP tool. These concern relationships between use cases, and the tool is able to create composite state machines from parent and child use case machines in the case of `include` relationships, and to a limited extent, for `extends` relationships. Other forms of use case relationship are also supported, since there may be ordering relations between use cases (Tasie-Amadi, 2004).

5.2 Interaction Models

In MoCUS, sequence diagrams are used to define use case scenarios and also desirable/undesirable properties. The latter can be useful in its own right, and also where behaviors cross use case boundaries (Tasie-Amadi, 2004).

To check a sequence diagram against the UCM, or the COM, a CSP representation of the diagram is required. The IM differs from the UCM and the COM in that sequence diagrams are not typically represented as state machines. Although it is possible to produce a state machine representation, it is simpler to generate a CSP process directly from the sequence diagram. This can be achieved by treating each actor and object in a sequence diagram as a CSP process. The behavior of each of these processes is defined by the communications that occur on the corresponding object's timeline, and the behavior of the sequence diagram is defined by the parallel composition (`||`) of these processes. For example, consider the simple sequence diagram shown in Figure 17.5. The sending of the synchronous message `m21` from `o1` to `o2` is treated as an event that is shared by `o1` and `o2`, and is the first event that can occur in a trace of either process. Hence the CSP specification for this interaction, which will be called `I1` is

```
I1 = O1 || O2 || O3
O1 = m21 -> r_m21 -> SKIP[3]
O2 = m21 -> m31 -> r_m31 -> r_m21 -> SKIP
O3 = m31 -> r_m31 -> SKIP
```

[3] `SKIP` is a primitive CSP process that represents successful termination.

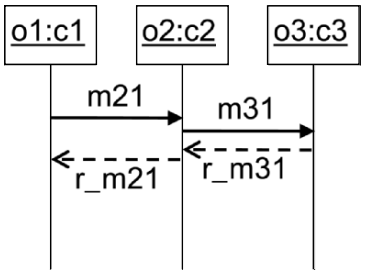


Fig. 17.5 Simple sequence diagram

This approach applies to asynchronous communication if the positions of the messages in the diagram represent a single ordering from the possible set of orderings. It can also yield partial orders relating to asynchronous message sends but is not completely satisfactory when messages may be received in various orders, when multiple sequence diagrams are required to model the different sequences in a partial order. An alternative approach that rectifies this problem is discussed in Tasie-Amadi (2004), but has not yet been implemented.

5.3 The Composite Object Model

Objects appearing in sequence diagrams must have a class state machine, which are mapped to CSP processes (Section 4). Details of communications between objects are extracted from sequence diagrams and state machines, and are used to connect the processes via communications channels and buffer processes for asynchronous messages. The CSP "object" processes are composed in parallel, and the result is the COM. Total or partial orders of message exchanges arise naturally within the model from the combination of parallel composition of objects, the type of communication between objects (synchronous or asynchronous), and the object state machines.

Many object-oriented (OO) software systems feature dynamic object creation and destruction. Although this is much more difficult if objects are implemented in hardware, recent advances in the run-time management of field programmable gate array (FPGAs) with on-chip block RAM may reduce these difficulties (Edwards and Green, 2003). Hence, dynamic creation and destruction are considered here. However, applying model-checking to systems with unrestricted dynamic creation is not possible since the state space to be checked is unbounded (Holzmann, 1997). This problem is overcome by requiring that the maximum number of run-time instances of each class be bounded, and by modifying the class state machines as shown in Figure 17.6 (Tasie-Amadi, 2004). All of the instances that can possibly exist at run-time are then included in the COM.

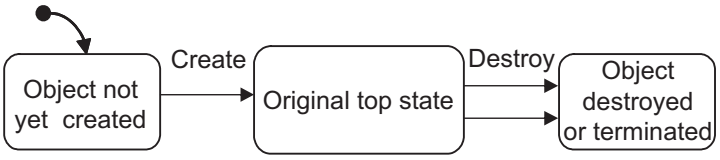


Fig. 17.6 Modified object state machine facilitating dynamic object creation/destruction

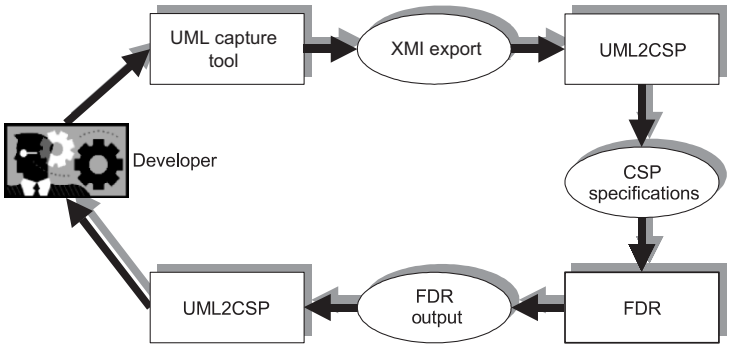


Fig. 17.7 Structure of the UML2CSP tool

Object state machines are modified by treating the original machine as a composite state, and by adding two new states as shown in Figure 17.6. Hence, an object begins life in the "object not yet created" state, and when it is brought into existence by a create call it enters the "original" state machine. If the object is deleted via a destroy message, or its state machine moves into a top-level final state, it moves into the "object destroyed or terminated" state, and cannot respond to any further messages.

6. The UML2CSP Tool

The UML2CSP tool logically sits between a UML model capture tool and the FDR model-checker (Figure 17.7). When developers have completed a set of UML models, they must export the models in XML model interchange (XMI) format (Grose, 2001). This is a file format that has been developed to support model exchange between UML tools from different vendors. UML2CSP is written in Java and is based around a public domain XMI parser. The graphics user interface (GUI) of the tool enables models in XMI format to be selected by users for loading and translation to CSP. It also enables the types of checks to be performed to be specified. Although it would be advantageous for the output of FDR to be translated back into UML form before output to the developer, this has not yet been implemented and so the developer must be able to interpret the output from the FDR tool.

7. Applying FDR to Translated Specifications

Although MoCUS was developed for use within HASoC at specific points within the design flow, model-checking with UML2CSP can be performed at any time, so long as the models to be checked have been completed, exported in XMI format, and translated into CSP.

When FDR is utilized for checking, the semantic model (Section 3.1) to be used must be specified. Since the failures-divergences model is the most powerful, refinement checks would usually be performed in this model unless there is good reason to use an alternative.

When checking an interaction against a use case, the concern is whether or not the external behavior of the two models is consistent, i.e., whether the sequence(s) of message exchanges between objects and actors in the sequence diagram is consistent with the use case state machine. The traces model is used for this type of check to verify that the traces of the interaction are a subset of those of the use case. The more complex semantic models cannot be used since they involve the requirement that the refusals of the two models should be consistent. When a use case is offered, in general, the scenario that is enacted is determined by the environment, and so typically a use case cannot refuse to engage in any of its scenarios. Hence the refusal sets of the interaction and the use case will often be different and so the failures or failures-divergences models cannot be used. This argument also applies to checking interactions against the COM.

Checks of a use case against the COM are also concerned with external sequences of events. Here, since the COM realizes all the behavior of the use case state machine, the sets of refusals should be compatible and so the above argument does not apply. Hence the failures-divergences model is used, or if divergence is absent, the failures model is sufficient.

There is a problem if several use cases realized by the COM can respond to the same event(s). For example, if use cases `A` and `B` both respond to message `x` and use case `A` is checked against the COM, then FDR may find external event sequences that contain `x`, but which do not belong to use case `A`. Consequently, when use cases respond to the same event, developers must provide distinct names in the different use cases.

It may be imagined that property checking can also be accomplished via refinement checks, since, for example, a desirable property of a process is a subtrace of that process. If all events that are not part of the property are hidden,[4] then it might be assumed that a refinement check could be applied.

[4]CSP includes a hiding operator that takes as arguments a process and a set of events, and produces a process that is a modification of the original, such that none of the events in the set appears in any trace of the modified process.

However this is not the case, because refinement checking always searches the state spaces of the two processes *from their initial states*. Hence, imagine that it is desirable to check that the COM always exhibits some property, i.e., for all executions of the COM, a certain sequence of events occurs within the COM's trace of events. In the general case, the subtrace corresponding to the property will not be possible from the initial state of the COM and so a refinement check will fail, although the property might actually hold.

Property checking uses an approach based on string matching with *deterministic finite automata* (DFA; Tasie-Amadi, 2004). In essence, the property is equivalent to a string that is sought in a character stream (process trace), and since techniques for the construction of DFA recognisers for strings are well-known (Crouchemore and Hancart, 1997), it is possible to construct a DFA from a sequence diagram description of the property. The DFA is termed as an interaction graph and is translated into CSP using the techniques discussed in Section 4. It may, however, be noted that this can only be done for properties that represent a total order. If a property is partially ordered, then a separate DFA must be generated for each possible sequence.

The CSP specifications for the model to be checked (`M`) and the interaction graph (`I`) are composed together in parallel in such a way that they synchronize on all events in `M`. `I` is constructed so that if the next event is part of the property, a transition to the next state in `I` occurs, whereas if it is not, a transition back to the initial state of `I` takes place. If the final state of `I` is entered, then the property has been found in the trace of the model `M`. If the whole state space of `M` has been searched and `I` has not entered its final state, then the model does not display the property.

8. Partial Case Study

Examples of applying MoCUS can be found in Tasie-Amadi (2004). Part of one of these studies is presented here, relating to the well-known mine-pump system. The system is installed in a mine to drain water that gathers at the bottom of the shaft. When the water reaches a given depth, a pump is switched on. When the water reaches a low water mark, the pump is switched off. This simple scenario is complicated by a number of other factors, but these are not relevant for this discussion. The entire UML model of this system has been translated into CSP and subjected to extensive checking with FDR (Tasie-Amadi, 2004).

Figure 17.8 shows the state machine representing the use case that monitors the water level. A scenario of this use case, representing the situation where the depth of water exceeds the threshold and the pump is switched on, appears in Figure 17.9a. The results of checking the interaction of Figure 17.9a against the use case of Figure 17.8 is shown in Figure 17.9b. The highlighted line with the tick indicates that the interaction is a trace refinement of the use case,

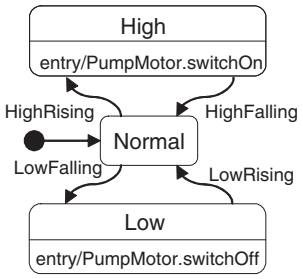
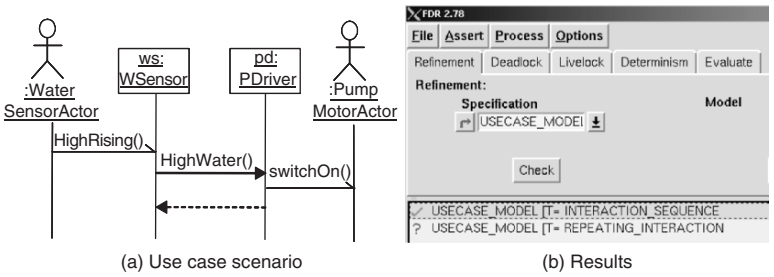


Fig. 17.8 Use case monitor water level



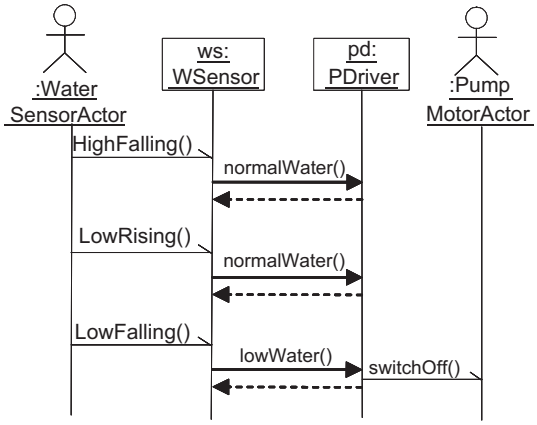
(a) Use case scenario (b) Results

Fig. 17.9 Checking an interaction against a use case

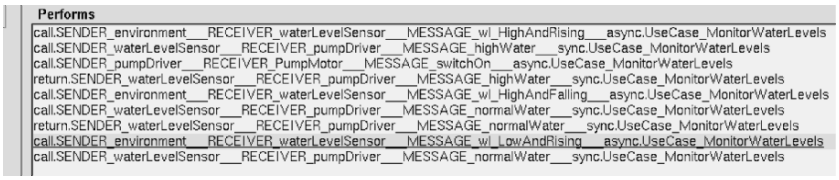
i.e., that the two models are consistent. The line below indicates that the scenario cannot be executed repeatedly. This is clearly correct—the pump cannot be repeatedly switched on, without having been first switched off.

A property, related to the use case of Figure 17.8, which the system is intended to display is shown in Figure 17.10a. It models the situation that prevails after the water depth has exceeded the high water threshold, the pump has been turned on, and the water level is falling. This property was checked against the COM. As it cannot be observed from the initial state of the use case, the check was performed using an interaction graph. The result of this check (not shown) indicated that the property does not hold in the COM. This is not surprising as there is clearly an error in Figure 17.10a.

A disadvantage of using interaction graphs is that although the technique can detect errors, it does not at present provide any indication of why or where the error occurs. However, it is possible to gain greater insight into the source of the error by performing a refinement check. Although this is bound to fail, the sequence of events (counterexample) that is produced can prove helpful in locating the source of the problem. Figure 8 shows the counterexample produced by a refinement check



(a) Desirable property



(b) Results

Fig. 17.10 Desirable property and the results of a refinement check

The highlighted line indicates that the COM cannot perform the indicated message at this point in the trace. `wl_LowAndRising`[5] is a message from the environment indicating that the water is rising from a low level. However, the property in Figure 8 is concerned with water falling after the high water threshold has been exceeded and the pump has been switched on. Hence the inclusion of the `wl_LowAndRising` message in the property is an error. Removing this message, and those that it triggers, from the property and repeating the check shows that the COM upholds the corrected property.

9. Conclusions

MoCUS facilitates the application of model checking to UML models, enabling developers to perform exhaustive consistency checks between models in an effort to uncover errors early in development. It also enables developers to specify properties that are desirable or undesirable and then to exhaustively check models to determine whether or not the model displays those properties. MoCUS has proved to be highly effective in uncovering errors, and many mistakes (both intentional and unintentional) have been found in the full mine-pump model and others that have been developed (Tasie-Amadi, 2004).

[5]This event is referred to as `LowRising` in the use case and sequence diagrams.

This enables developers to ensure that only models that are provably correct are input to the partitioning and implementation synthesis processes. Although this alone does not guarantee the correctness of the final implementation, it does remove major sources of error and so will help to reduce development timescales.

There are many avenues for further research. These include investigating the composition of use case state machines into a single automaton that specifies the externally visible behavior of the system, and using this to verify the COM. Some works on this topic are reported in Tasie-Amadi (2004), although a number of problems in achieving this goal are also reported. Another area worthy of study concerns applying MoCUS to hardware systems, rather than abstract object models, e.g., to investigate the overall communications behavior of a hardware platform.

References

Booch, G., Rumbaugh, J., and Jacobson, I. (1999) *The Unified Modeling Language Guide*. Addison-Wesley, Boston, MA.

Chang, H., Cooke, L. R., Hunt, M., Martin, G., McNelly, A., and Todd, L. (1999) *Surviving the SOC Revolution—A Guide to Platform-Based Design*, Springer, Dordrecht, The Netherlands.

Crouchemore, M. and Hancart, C. (1997) Automata for matching patterns. In: Rozenberg, G. and Salomaa, A. (eds) *Linear Modeling: Background and Application*, volume 2 of *Handbook of Formal Languages*. Springer, Berlin, pp. 399–462.

Davies, J. and Crighton, C. (2002) Concurrency and refinement in the unified modeling language. *Electronic Notes in Theoretical Computer Science*, 70(3):1–27.

Edwards, M. D. and Green, P. N. (2003) Run-time support for dynamically reconfigurable computing. *Journal of Systems Architecture*, 49:267–281.

Engels, G., Kuster, J., Heckl, R., and Groenewegen, L. (2001) A methodology for specifying and analyzing consistency in object-oriented behavioral models. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, New York.

FSEL (2003) *Homepage of the FDR2 Model-Checker and other CSP Tools*. Formal Systems (Europe) Ltd. `http://www.fsel.com/`.

Green, P. N., Edwards, M. D., and Essa, S. (2002) HASoC—towards a new method for system-on-a-chip development. *Design Automation for Embedded Systems*, 6(4):333–353.

Grose, T. (2001) *XMI Framework*. IBM Corporation, `http://alphaworks.ibm.com/tech/xmiframework/`.

Holzmann, G. (1997) The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

Lilius, J. and Paltor, I. P. (1999) vUML: a tool for verifying UML models. In: *Proceedings of the 14th IEEE International Conference on Software Engineering (ASE'99)*. IEEE, pp. 255–258.

Morris, D., Evans, D. G., Green, P. N., and Theaker, C. J. (1996) *Object Oriented Computer Systems Engineering*. *Applied Computing*. Springer, London.

Ng, M. and Butler, M. (2002) Tool support for visualizing CSP in UML. In: George, C. and Miao, H. (eds) *Formal Methods and Software Engineering—4th International Conference on Formal Engineering Methods, ICFEM 2002*, volume 2495 of *Lecture Notes in Computer Science*. Springer, Shanghai, China. pp. 287–298.

Ng, M., Butler, M., and Towards, M. (2003) Formalizing UML state diagrams in CSP. In: *Proceedings of the 1st IEEE International Conference on Software Engineering and Formal Methods*. IEEE, pp. 138–147.

OMG (2001) *OMG Unified Modeling Language Specification Version 1.4*. Object Management Group, `http://www.omg.org/`.

Schäfer, T., Knapp, A., and Merz, S. (2001) Model-checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):1–13.

Schneider, S. (2000) *Concurrent and Real-Time Systems*. Wiley, Chichester, UK.

SEMATECH (1999) *International Technology Roadmap for Semiconductors: 1999 Edition*. Semiconductor Industry Association, Austin, TX. `http://public.itrs.net/`.

Tasie-Amadi, K. (2004) *Verification of UML Behavioural Models*. PhD thesis, UMIST, Department of Computation, UMIST, Manchester, UK.