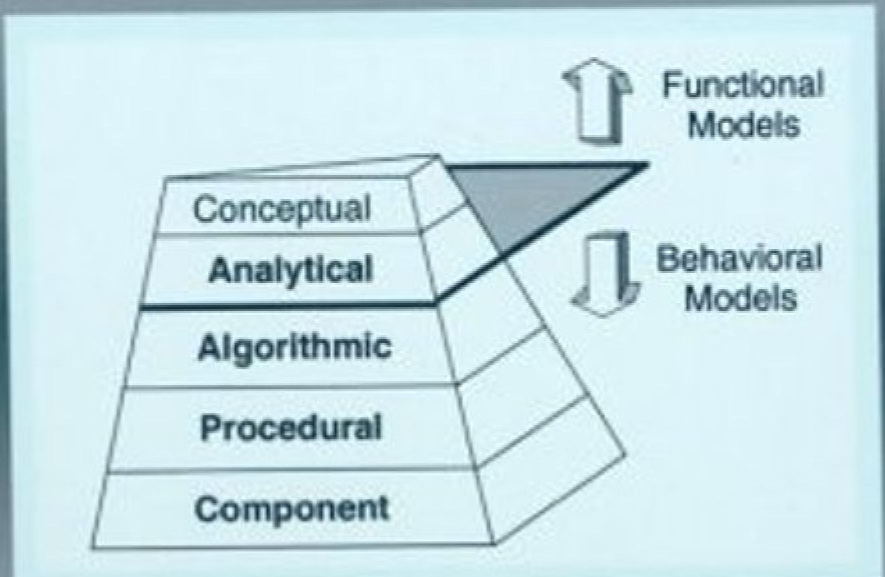
MODEL ENGINEERING IN MIXED-SIGNAL CIRCUIT DESIGN

A Guide to Generating Accurate Behavioral Models in VHDL-AMS

Sorin A. Huss



Kluwer Academic Publishers

MODEL ENGINEERING IN MIXED-SIGNAL CIRCUIT DESIGN

MODEL ENGINEERING IN MIXED-SIGNAL CIRCUIT DESIGN

A Guide to Generating Accurate Behavioral Models in VHDL-AMS

by

Sorin A. Huss
Darmstadt University of Technology

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-48101-4
Print ISBN: 0-7923-7598-X

To Monika and to our children Britta, Martin, and Michael. Thank you for your patience!

Contents

List of Figures

List of Tables

Acknowledgments

First of all I would like to thank Dr. Mohammed Ismail, Editor of the Book Series, for his enthusiastic and encouraging support in the conceptual phase of this book. Without his ongoing help this book project would have remained to be just another project. Thanks also to Mark de Jongh of Kluwer Academic Publishers for the pleasant cooperation.

This work is based on research activities in mixed-signal as well as in analog design and modeling performed at my institute over the last decade. I would like to thank my PhD students Wolfgang Boßung, Michael Goedecke, Hatem Hamad, Steffen Klupsch, and Ralf Rosenberger for their significant contributions to this area. Their research results are reflected in this book as well as the implementation work of many undergraduate and graduate students, namely Nadeem Bhatti, Oliver Glier, Karsten Grüner, Tobias Kuckuck, Felix Madlener, Kai Morich, Michael Stini, Wolfram Stumpf and Lars Wehmeyer. Philipp Hahn, Stephan Hermanns, and Stephan Klaus contributed their expertise in generating LaTeX documents, their valuable help in the final phase of this book project is gratefully acknowledged.

All bits and pieces of the considerably large design software suites available at my institute have been kept together by the efforts of Eva Glaser, the system administrator.

Special thanks goes to Elisabeth Hudson. She completed all text processing and figures drawing in an excellent way, a not always easy task. In addition, she morphed my sometimes rather basic English expressions into readable sentences. However, the responsibility for typos, mistakes, and errors remains completely with me.

Commercial trademarks as referred to in this book are assigned to their owners according to my best knowledge. I appologize in advance for any inaccuracy in this matter.

Sorin A. Huss
Darmstadt, August 2001

Foreword

Model engineering is an important activity within the design flow of integrated circuits and signal processing systems. This activity is not new at all in computer engineering, however, and takes a central role in practice. Model engineering of digital systems is based on agreed concepts of abstraction hierarchies for design object representations as well as the expressive power of hardware description languages (HDL). Since their gradual introduction over time HDL have proved to form the foundation of design methodologies and related design flows. Design automation tools for simulation, synthesis, test generation, and, last but not least, for formal proof purposes rely heavily on standardized digital HDL such as Verilog and VHDL.

In contrast to purely digital systems there is an increasing need to design and implement integrated systems which exploit more and more mixed-signal functional blocks such as A/D and D/A converters or phase locked loops. Even purely analog blocks celebrate their resurrection in integrated systems design because of their unique efficiency when is comes to power consumption requirements, for example, or complexity limitations. Examples of such analog signal processing functions are filtering or sensor signal conditioning. In general, analog and mixed-signal processing is indispensable when interfacing the real world (i.e., analog signals) to computers (i.e., digital data processing). Validation of integrated systems, an activity to be executed during the whole design flow, requires a single HDL for model representation in order to handle both partitions of the system model and especially their interaction efficiently.

Therefore, abstract descriptions of analog and mixed-signal systems and components are a new trend in model engineering. Again, modeling of such design objects is not as new as it might seem from the term of 'behavioral' modeling, an almost ubiquitous buzz word nowadays. Structural descriptions from basic components such as transistors and somewhat more abstract representations of analog circuits denoted as macro models have been used in practice for decades by analog circuit designers for analysis purposes exploiting

SPICE-like simulators. The intrinsic behavior of such models is transparent to most design engineers because it is well hidden within predefined component libraries. The availability of HDL for analog and especially for mixed-signal application domains has considerably changed this situation. Now, a modeler is enabled to express directly the behavior of parts of the integrated system without being limited to low-level model primitives such as transistor instances or controlled voltage sources. However, new questions arise, which are quite similar to those in the early days of modeling in the digital domain. These questions address abstraction level hierarchies, modeling concepts and related methods, model calibration and representation (i.e., the whole range of model engineering in mixed-signal systems).

The purpose of this book, therefore, is to combine the main issues of hardware description, characterization methods for the extraction of model parameters, and modeling methodologies for accurate high-level models of mixed-signal components and functional blocks. The work presented here emphasizes — for the first time — an engineering view on model generation and handling, thus providing a unique guide both for practitioners and students of electrical and computer engineering at graduate level. Chapter 1 presents an introduction to the model flow within integrated systems design, to generic model classes as well as to fundamental modeling concepts and representation languages. Chapter 2 is dedicated to the specification of behavior for analog and digital components. Abstraction hierarchies for these components are presented and discussed with respect to mixed-signal applications. Chapter 3 is intended to present a compact introduction to the basic concepts and to the expressivity of the HDL covered by the new IEEE standard 1076.1, also known as VHDL-AMS. Chapter 4 addresses circuit property extraction (i.e., characterization issues of analog building blocks). A new modeling methodology for mixed-signal circuits is proposed in Chapter 5. Finally, Chapter 6 presents results of the outlined model engineering methods for circuit examples of different complexity and operation domains. Several conclusions are summarized at the end of Chapter 6.

Chapter 1

INTRODUCTION

Analog and mixed-signal integrated circuits (IC) design represents a major challenge for the design of complex information processing systems, especially when it comes to efficient top-down design flows. The generic architecture of mixed analog-digital systems being integrated into one IC, which is known as the *System-on-a-Chip (SoC)* style, consists of DSP cores and microcontrollers surrounded by A/D and D/A converters, which interface the internal bulk of digital processing to the analog sources and sinks of external information. In the signal processing and integrated circuits community it is widely agreed upon that analog and mixed-signal design expertise will increasingly be exploited for an implementation of powerful and at the same time cost-effective products in the areas of communication, consumer and automotive applications. When it comes to discussions on appropriate design flows for such products then two major problem regions may be identified. First, the design tools applied for design tasks in the analog and in the digital domain have to ”talk to each other”. An explicit need for such a tool communication is present especially in converter design. Secondly, top-down design — a proven adequate design methodology at least for digital systems — has to be adopted to the mixed-signal domain. However, good model-building concepts and efficient tools are essential for a painless transfer of abstract top-down methodologies to engineering practice. In addition, appropriate calibration methods are a precondition to ensure that high-level behavioral models do not diverge from lower-level, detailed models. In this chapter, therefore, we first will be discussing some basic issues related to model flows, modeling concepts, and model representation languages, which are the main means to making models executable (i.e., to get them to work).

1.1 Model flow in Mixed-Signal Design

The design process for a technical product may, in general, be roughly subdivided into three main phases: *conceptualization, concept refinement,* and *implementation.* Inputs to the process are the design requirements, eventually yielding in the design results. A more detailed view to this 'black box' model of the design process as depicted in Fig. 1.1 results in an identification of a process chain consisting of *generating* and *analyzing activities* to be performed iteratively within the outlined steps of conceptualization, refinement, and implementation.

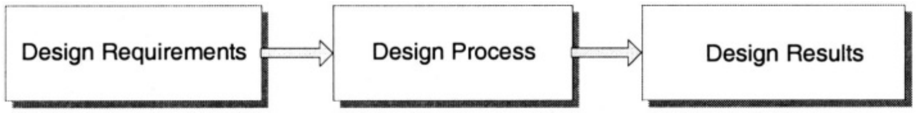


Figure 1.1. 'Black box' model of a design process

Design requirements consist of a description of the envisaged functionality of the new product and sometimes of constraints referring to the final implementation such as an exploitation of commercially available subsystems offered either as standard components or as Intellectual Property (IP) products. This set of information is commonly known as the *technical product specification* and it is still denoted in an informal way (i.e., written in a natural language and augmented by some tables and diagrams). Nontechnical specifications such as cost frames and design deadlines are important for the design process as well. They are, therefore, viewed as additional inputs to the design process as outlined in Fig. 1.1.

The first and most important activities of the systems engineer during the conceptual phase are *formalization* of the specification, *determination* of solution strategies, and *partitioning* of the overall task into independent subtasks to be forwarded to design teams specialized in different areas such as analog circuit design or real-time software engineering. Fig. 1.2 depicts the design flow during conceptualization and refinement.

Modeling plays a central role in this design phase. Different model instantiations, abstraction levels, and accuracy requirements have to be dealt with during concept refinement. In addition, multi-nature systems, in general, operate time-continuously, but the information processing inherent to most such systems consists of digital hardware and software modules, which are best represented in a time- or event-discrete way.

The design flows in state-of-the-art digital systems are based on a mature and widely accepted abstraction hierarchy and on the resulting design activi-

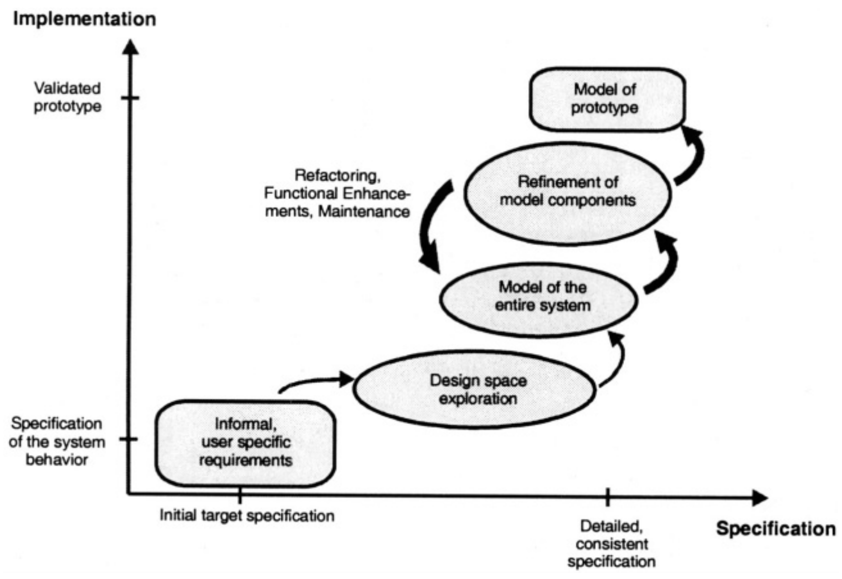

Implementation
Validated prototype
Specification of the system behavior
Model of prototype
Refinement of model components
Refactoring, Functional Enhancements, Maintenance
Model of the entire system
Design space exploration
Informal, user specific requirements
Initial target specification
Detailed, consistent specification
Specification

Figure 1.2. Design flow for systems design

ties. A holistic thinking of systems properties and requirements is supported by coarse grain structural descriptions together with a functional partitioning of the system and by high-level language descriptions of the systems partitions or components as well as their interaction via communication. At lower abstraction levels physical behavior comes in gradually, which is supported by appropriate simulation tools and modeling paradigms. Sophisticated verification, synthesis and analysis tools tie together the level hierarchy and thus form a consistent generic design flow with a variety of application-specific instantiations.

The actual situation in analog design is completely in contrast to the well-established and elaborated digital systems design flow. This flow of design activities is much more primitive in the sense that low-level component descriptions and early considerations of physical behavior still prevail the activities of circuit design engineers. Simulation tools address mainly low-level representations of functional blocks, and synthesis is — except a few experimental approaches from academia, e.g., [FVG00] and first commercial attempts such as [Ant98] — almost not present in practice.

In mixed-signal design the problems resulting from the rather poor state of the art in analog circuit design automation are worsened by the following facts. First, the design space for possible systems implementations is considerably enlarged because of the multiple choices available for an implementation of one and the same functional block. An audio filter, for instance, may be implemented as an active RC circuit, as a switched-capacitor module, or as a digital filter, which, in turn may be instantiated by an application specific piece of hardware or an algorithm running either on a general purpose processor or a digital signal processor. Secondly, a not appropriate consideration of the sophisticated interaction of analog (i.e., time and value continuous signals) and digital (i.e., time and value discrete signals) internal to a system may result in an out-of-spec function at least, which is likely to be detected in the implementation phase only thus causing major redesign efforts.

Our interest is focused on the model flow associated to the system design phases as summarized in Fig. 1.2. Model generation and application play a central role during the whole design flow except the physical implementation of a mixed-signal system. Rather different requirements on property coverage and accuracy have to be combined into model instantiations and refinement procedures.

When introducing modeling as a central method for the support of almost the whole design task, which starts from a specification of the intended systems behavior and ends with a real product, the question arises how models relate to the physical reality.

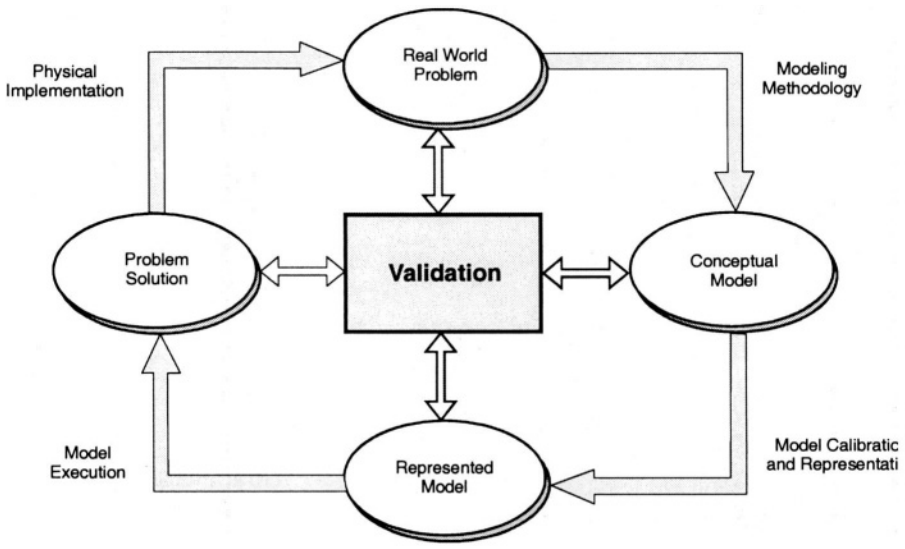


Figure 1.3. Model generation and validation

Fig. 1.3 highlights this rather complex relationship [LMO83, Rob99]. Starting from an in-depth analysis of the real world problem, an abstraction first takes place in the sense that properties and relationships are to be identified, which must at least be present in the model to be developed. The result of this activity, in general, is a set of mathematical equations denoting either first order, idealized relations, or more detailed relationships. Then an appropriate modeling concept has to be established and the set of equations has to be mapped to it accordingly. The next step consists of representing this intermediate model in an executable description language and in calibrating model parameters to data taken from the real world problem. Executing the model or, in the general case, a set of interacting models produces a simulation result, which has to be assessed as whether or not it copes with the specification extracted from the real world problem. In case model outputs and specification fit within predefined tolerances, possibly after completing several model refinement steps in between, then an implementation of the system takes place next, finally resulting in a physical representation of the systems model. Obviously, *validation* takes a central role in this modeling flow as depicted in Fig. 1.3.

1.2 Model classes

Over the past decades systems theory has elaborated an interdisciplinary approach to the modeling problem of dynamic systems. It emphasizes theoretical concepts for the specification of dynamic systems at high abstraction levels. The fundamental concepts highlighted in the following are based mainly on proposals of Zeigler et al., which are presented and discussed in detail in [ZPK00].

A generic system consists of inputs, outputs, an internal state, and a specification of the systems dynamics. The internal state denotes its behavior at a certain point in time, whereas the dynamics describes how the internal state evolves over time. Therefore, a state transition function is introduced which determines the next state depending on the actual overall state and the inputs. In turn the outputs are calculated from the actual internal state and, possibly, from the input values. The formal definition of a general *Input-Output-System* IOS is given by

$$IOS = \{t, \vec{u}, \vec{y}, \vec{x}, \delta, \lambda\}. \tag{1.1}$$

t	Time
$\vec{u}$	Vector of input parameters
$\vec{y}$	Vector of output parameters
$\vec{x}$	Vector of state variables
δ	State transition function
λ	Output function

This rather abstract model has to be refined for the description of time-continuous, time-discrete, and later on for an additional hybrid model class, thus resulting in the generic classes denoted as *DESS* (Differential Equation Specified System), *DEVS* (Discrete Event Specified System), and *DTSS* (Discrete Time Specified System).

DESS, the time-continuous dynamic system, is defined as

$$DESS = \{\vec{u}, \vec{y}, \vec{x}, f, \lambda\}. \tag{1.2}$$

$\vec{u}$	Vector of input parameters
$\vec{y}$	Vector of output parameters
$\vec{x}$	Vector of state variables
f	Function of change over time
λ	Output function

A DESS is characterized by time-continuous waveforms of its variables, whereas the time base is the set of real numbers. This means that all variables (i.e., input, output and state) may change their values an infinite number of times within a bounded time interval. The rate of change function f is, in general, represented by a set of differential equations which defines the mapping of both state and input parameters to state variables. Fig. 1.4 depicts the characteristic waveform of signals within a DESS.

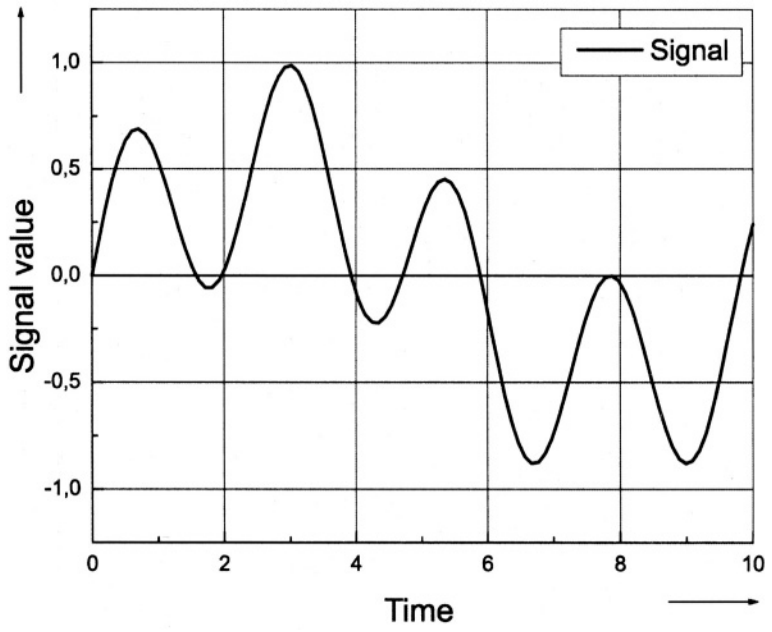


Figure 1.4. Time- and value-continuous signal waveform

DEVS, the event discrete dynamic system, is defined as

$$DEVS = \{\vec{u}, \vec{y}, \vec{x}, \delta, \lambda, ta\}. \tag{1.3}$$

$\vec{u}$	Vector of input parameters
$\vec{y}$	Vector of output parameters
$\vec{x}$	Vector of state variables
δ	Transition function for events
λ	Output function
ta	Time advance function

A DEVS is characterized by real values for variables and by a continuous time base. In contrast to a DESS there is a limited number of times for the changes of variable values within a bounded time interval. A change of variable values takes place at an *event time point* only. Events may result either from value changes at inputs or from changes of internal (i.e., state) variables of the system. This is a standard situation in digital systems, where changes of internal signals resulting from a state transition calculation are scheduled for future points in time relative to the actual time of the system thus implementing a time delay operator. Fig. 1.5 visualizes the characteristic waveform of signals within a DEVS.

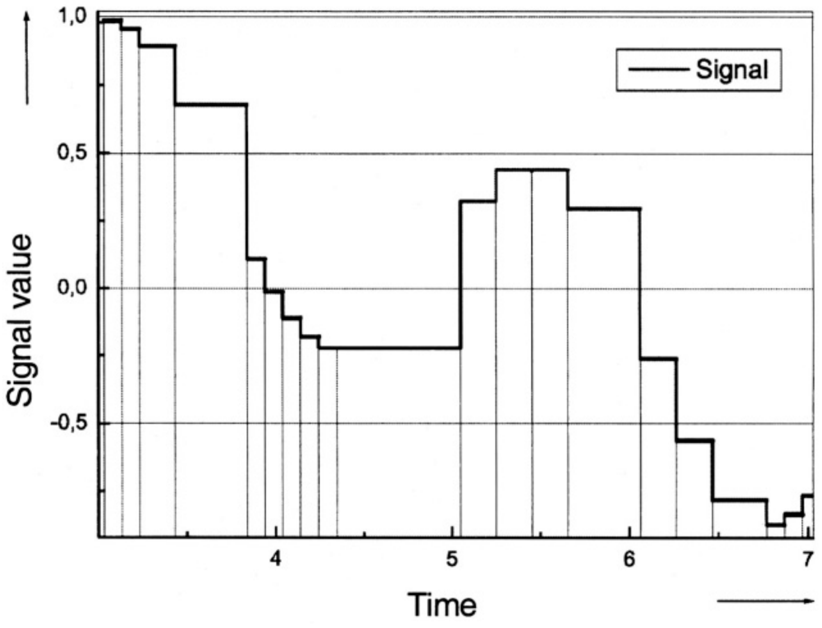


Figure 1.5. Event discrete signal waveform

Finally, DTSS, the time discrete dynamic sytem, is defined by

$$DTSS = \{\vec{u}, \vec{y}, \vec{x}, \delta, \lambda\}. \tag{1.4}$$

$\vec{u}$	Vector of input parameters
$\vec{y}$	Vector of output parameters
$\vec{x}$	Vector of state variables
δ	Transition function for events
λ	Output function

A DTSS features a discrete time base. Values of variables are defined at these discrete, in general, equidistant points in time. As a consequence, value changes may only take place at these a priori known time points. Difference equations are the usual representation means for the description of the behavior in DTSS, also known as *sampling* systems. Fig. 1.6 depicts the characteristic waveform within a DTSS. Note that DESS, DEVS, and DTSS are causal models according to the definition given in the next section.

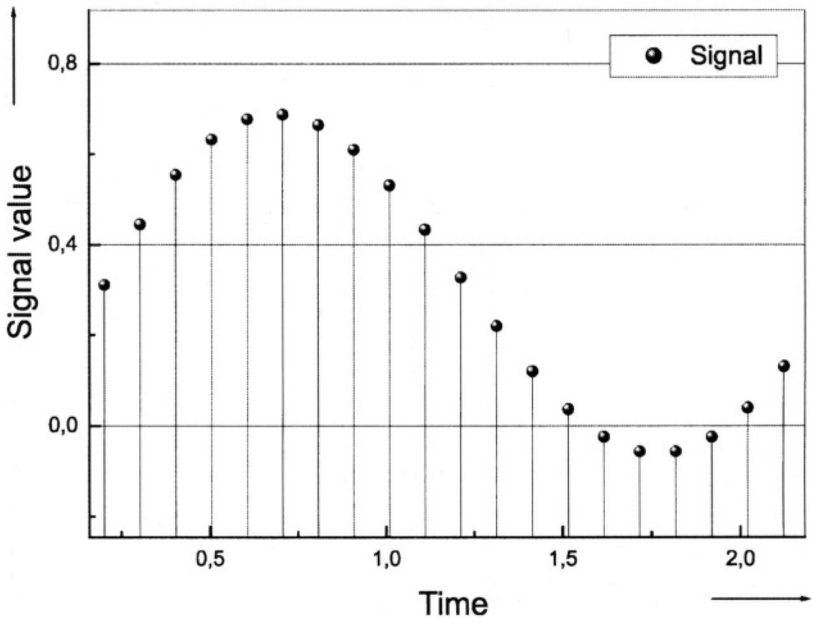


Figure 1.6. Time-discrete signal waveform

1.3 Modeling languages

Modeling languages are the primary means for awaking conceptual models as depicted in Fig. 1.3 to life (i.e., evolving their behavior over time according to excitations). Many languages and associated model execution tools usualy known as *simulators* have been developed over the past decades. They may roughly be classified into *analog*, or, more precisely, time and value continuous, and into *digital* (i.e., event discrete) systems modeling languages. In addition, one notices an application domain biasing of many of these languages, especially in the time and value continuous area.

Languages aimed at the modeling of the signal flow in continuous systems such as MatLab/Simulink[1], Mathematica[2], or MATRIXx [3] emphasize a block-oriented, equation-based modeling style, which is very useful when designing control or signal processing systems, for example. A system is thus described in terms of a priori known quantities, by unknown quantities, and by a sufficiently large set of algebraic and/or differential equations which relate known to unknown quantities. This modeling concept is denoted as *causal* because the information flow is unidirectional from the (known) inputs to the (unknown) outputs of the systems model, which in turn is composed from more or less complex functional blocks.

Fig. 1.7 visualizes this modeling concept. The quantity vectors acting as inputs and outputs to the model are not necessarily of the same dimension.

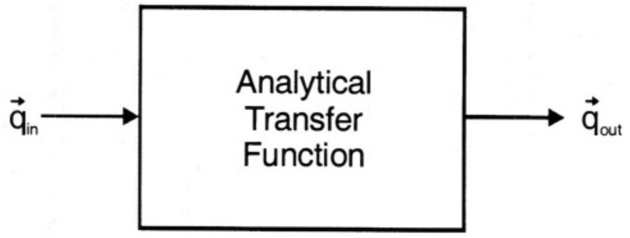


Figure 1.7. Causal model

In contrast, modeling languages for dedicated application domains such as electrical and electronic circuit engineering — an area we will focus on in the sequel — rely on the *acausal* modeling concept. The acausal (sometimes denoted as *noncausal* or *first principles* modeling) concept refers to a description of component and system behavior under consideration of *conservation laws,* such as Kirchhoff's Rules for electrical networks. Here, there is no explicit allocation to inputs and outputs of a system. Conservation laws combined with component behavior captured by the associated *constitutive equations* form the complete set of linear and/or nonlinear algebraic and differential equations

[1]MatLab and Simulink are trademarks of MathWorks, Inc.
[2]Mathematica is a trademark of Wolfson Research, Inc.
[3]MATRIXx is a trademark of Wind River Systems, Inc.

which is to be solved. The signal flow within the system is bidirectional, which is a characteristic of acausal descriptions. SPICE[4], Saber[5], and Dymola[6] are examples for such languages and simulation tools.

Fig. 1.8 highlights the connectivity situation with an acausal model. Inputs or outputs are not directly visible from this kind of model description.

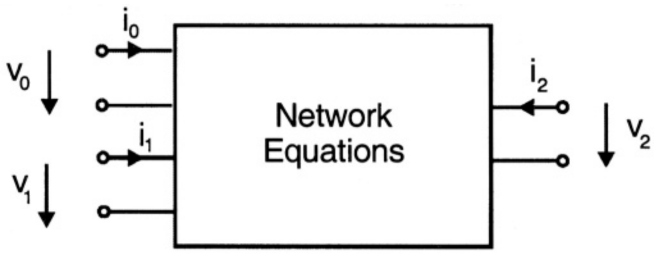


Figure 1.8. Acausal model

The fundamental differences of these modeling concepts are best demonstrated by a simple example. For this purpose, consider the electrical circuit shown in Fig. 1.9. The circuit schematic does not explicitly denote its inputs and outputs, it thus represents an acausal model. An experienced circuit designer, however, may easily conclude from her or his knowledge of network theory that the single source in Fig. 1.9 acts as an input, but outputs are still undefined.

A calculation of all voltages and currents present in the circuit of Fig. 1.9 is accomplished from the set of constitutive element equations

$$\begin{aligned} i_1 &= \frac{v_1}{R_1} \\ i_2 &= \frac{v_2}{R_2} \\ \frac{di_L}{dt} &= \frac{v_L}{L} \\ \frac{dv_C}{dt} &= \frac{i_C}{C} \end{aligned} \qquad (1.5)$$

and from the set of equations related to conservations laws.

[4]SPICE is a trademark of the University of California at Berkeley
[5]Saber is a trademark of Avant! Corp.
[6]Dymola is a trademark of Dynasim AB

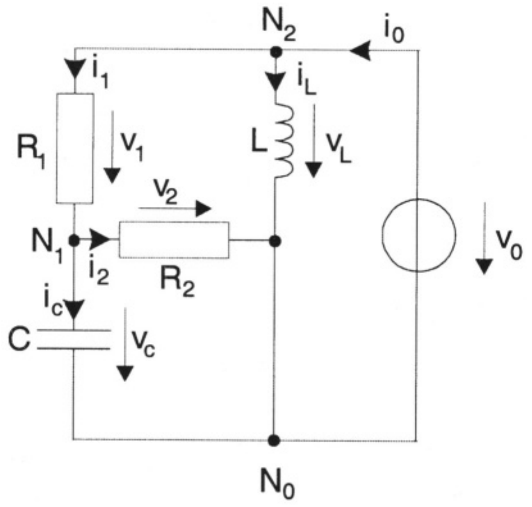


Figure 1.9. Linear network

$$
\begin{aligned}
i_0 - i_1 - i_L &= 0 \\
i_1 - i_2 - i_C &= 0 \\
v_1 + v_C - v_0 &= 0 \\
v_C - v_2 &= 0 \\
v_1 + v_2 - v_L &= 0\ .
\end{aligned}
\tag{1.6}
$$

Output quantities (i.e., some voltages and/or currents of the circuit) are then deliberately taken from the complete set of electrical voltages and currents derived from Eq. (1.5) and (1.6).

The related causal model of this linear network established from both Eq. (1.5) and (1.6) and represented as a block diagram is given in Fig. 1.10.

Now, both the input v_0 and the output i_0 are clearly identified from this model representation. However, quantities related to each other by constitutive element equations are separated by this kind of model description. This is visible from Fig. 1.10, for example, for i_C and v_C. An introduction to and an in-depth discussion of causal and acausal modeling concepts is given in [Cel91].

Causal modeling is highly appropriate in the conceptual phase of systems design when performing design space exploration under consideration of possible subsystem implementations in different engineering disciplines such as mechanical, electrical, and computer engineering, respectively. On the other hand, acausal descriptions are a prerequisite to the detailed design of systems

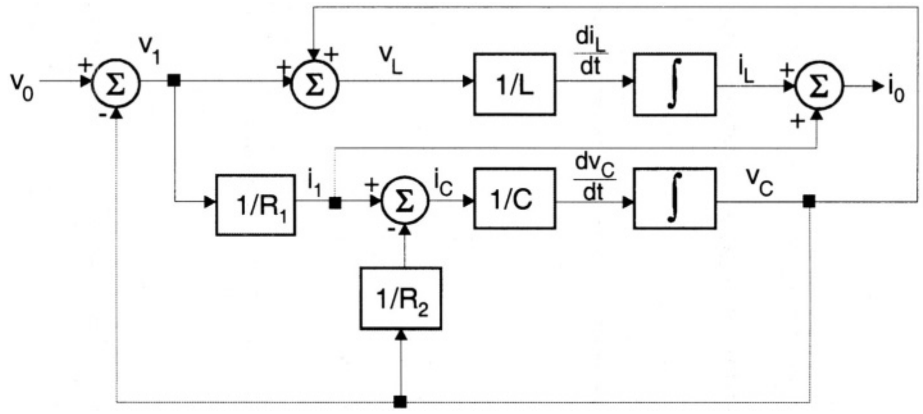


Figure 1.10. Block diagram of the linear network

components aimed to be implemented in one discipline only. A consistent and transparent transition of models over this barrier between modeling concepts is a serious problem within the general design flow as depicted in Fig. 1.2.

Due to the overwhelming success of digital technology and of computer engineering on the one hand and to an even increasing need to combine time-continuous and event-discrete operations within one system on the other hand, the objective of a combined mixed-signal modeling support became obvious. One of the early approaches to an integration of analog and digital model descriptions was PSpice[7]. Eldo-FAS [8], MAST[9], and Verilog-A/MS[10] were introduced subsequently.

Meanwhile, there are two recent modeling languages in the market place, which do not only combine time-continuous and event-discrete model descriptions, but at the same time support causal and acausal modeling concepts in different engineering disciplines. Modeling of truly heterogenous systems thus becomes possible by means of Modelica[11] and VHDL-AMS[12]. Modelica stems from the domain of general dynamic systems, whereas VHDL-AMS originates from computer engineering. Because both languages are aimed at heterogenous systems modeling, it is not surprising that component libraries of analog

[7]PSpice is a trademark of Cadence Design Systems, Inc.
[8]Eldo and Eldo-FAS are trademarks of Mentor Graphics Corp.
[9]MAST is a trademark of Avant! Corp.
[10]Verilog and Verilog-A/MS are trademarks of Cadence Design Systems, Inc.
[11]Modelica is a trademark of the Modelica Association
[12]VHDL and VHDL-AMS are the IEEE standards 1076 and 1076.1 respectively

circuits were elaborated for Modelica [CSLS00]. Mechanical models as well as thermal models are provided as VHDL-AMS packages [GCP01].

The following examples are intended to give some flavor to model representation in Modelica and VHDL-AMS, respectively. A more in-depth discussion of VHDL-AMS is given in Chapter 3. An excellent introduction to the Modelica language and to associated modeling concepts and many application examples may be found in [Til01]. The Modelica code outlined in the following has been extracted from this source for comparison purposes.

```
connector ElectricalPin
     Modelica.SIunits.Voltage v;
     flow Modelica.SIunits.Current i;
end ElectricalPin;

model Capacitor "Ideal component"
     import Modelica.SIunits;
     parameter SIunits.Capacitance
                        C = 1.e - 6 "Capacitance";
     ElectricalPin p, n;
     SIunits.Voltage v;
equation
     v = p.v - n.v;
     p.i = C * der (v);
     p.i + n.i = 0;
end Capacitor;
```

a)

```
entity Capacitor is
     generic (C : Real := 1e-6);
     port (terminal p, n : Electrical);
end entity Capacitor;

architecture IdealComponent
          of Capacitor is
     quantity v across i through p to n;
     begin
          i == C * v'dot;
end IdealComponent;
```

b)

Figure 1.11. Ideal electrical capacitor a) Modelica model [Til01] b) VHDL-AMS model

The first example is an ideal electrical capacitor represented by an acausal model. Fig. 1.11 shows the associated model codes. The constitutive differential equation of the component is denoted in both cases in quite a similar way — except for some syntactical details. Conservation laws, however, are specified completely different: The sum of currents has to be stated explicitly in the Modelica model, whereas this conservation law is set up in a transparent way by the VHDL-AMS language compiler according to the definition of `across` and `through` quantities.

Modeling of Boolean functions is again quite similar in both languages, but clearly not a representation of digital circuits featuring signal delays. Fig. 1.12 details these fundamental differences in the second example, the Boolean function AND, specified first without and then with a propagation delay.

```
block And
     Modelica.Blocks.Interfaces.BooleanInPort  inPort1 (n=1);
     Modelica.Blocks.Interfaces.BooleanInPort  inPort2 (n=1);
     Modelica.Blocks.Interfaces.BooleanOutPort  outPort (n=1);
equation
    outPort.signal = inPort1.signal and inPort2.signal;
end And;
```

a)

```
block Lag
     parameter Real c = 1 "lag time";
     parameter Real threshold = .7 "logical threshold";
     Modelica.Blocks.Interfaces.BooleanInPort  inPort (n=1);
     Modelica.Blocks.Interfaces.BooleanOutPort  outPort (n=1);
protected
     Real state "continuous state";
equation
     c * der (state) = if inPort.signal[1] then 1-state
                                           else -state;
     outPort.signal[1] = state >= threshold;
end Lag;
```

b)

```
entity And_gate is
     generic (Dly : Time := 1);
     port (signal  In1, In2 in : Boolean;
              signal Out out : Boolean);
end entity And_gate;
architecture DataFlow of And_gate is
begin
     Out <= In1 and In2 after Dly;
end DataFlow;
```

c)

Figure 1.12. AND gate a) Modelica function model [Til01] b) Modelica delay model [Til01] c) VHDL model

The Boolean function coded in Modelica, as depicted in Fig. 1.12 a), takes and produces Boolean signal values, whereas the output result is forwarded instantly. This functionality is represented by the VHDL model in Fig. 1.12 c) when deleting the string `after Dly` from the assignment to the output signal `Out`. In both cases there is no explicit propagation delay of the output signal, and the implemented function reflects the AND operation on Boolean signals. Modeling of delays is, however, different. The now present string `after Dly` of Fig. 1.12 c) causes the simulation system to schedule an event on `Out` subsequently at `Dly` time units. This event on `Out` takes place with reference to the input event, which causes the value change of the output signal. An *inertial delay model* is used for that purpose. Such a behavior may be represented in Modelica as well, but in a completely different way. The component denoted as `Lag` and shown in Fig. 1.12 b) has to be connected to the output port of the `And` model in Fig. 1.12a) for that purpose. `Lag` produces a delayed Boolean signal by means of a D/A conversion of the primary output of `And` to an internal signal by manipulating this continuous signal denoted as `state` in Fig. 1.12 b) and, finally, by performing an A/D conversion back to a Boolean signal. A detailed presentation of the associated waveforms is given in [Til01]. Now, it becomes obvious that the own turf of Modelica is continuous signal modeling and that digital behavior may be captured too, but not as straight forward as in VHDL. Anyway, the bottom line from these examples is that the intended model behavior may be expressed in either modeling language.

The selection of an appropriate or even the ”best” modeling language based on its semantic expressivity and syntactical details is a rather hot topic in academic discussions occasionally. However, as long as the required fundamental modeling concepts are addressed by the candidate language these criteria are not really the key issues in practice. This situation is reflected by available commercial simulators, which already support multi-language modeling. For example, ModelSim[13] or Scirocco[14] accept mixed Verilog and VHDL models for digital systems, whereas ADvanceMS[15] or SMASH[16] support combined VHDL-AMS and SPICE models. The differently coded models are jointly mapped by a front end compiler to intermediate data structures, a well-known technique in software engineering.

A design engineer is mainly interested in CAD tools that support her or his design flow at best and in how third party IP products may be exploited in order to ease her or his design burden. Tools for simulation, synthesis, and

[13]ModelSim is a trademark of Model Technology, Inc.

[14]Scirocco is a trademark of Synopsys, Inc.

[15]ADvanceMS is a trademark of Mentor Graphics Corp.

[16]SMASH is a trademark of Dolphin Integration SA

test generation are provided by many vendors for VHDL-based digital design. Representations of IP products also rely on this modeling language.

These are the reasons why VHDL-AMS is emphasized for the task of modeling mixed-signal electronic circuits in the remainder of this book.

Chapter 2

SPECIFICATION OF BEHAVIOR

Denoting the behavior of functional blocks and of complete systems is of utmost interest both to a modeler and to a circuit or system designer, who should be the same person for obvious reasons. In practice, however, this is not yet widely accepted. A separation of modeling and design tasks in many cases causes severe problems, which may be avoided by both getting modeling issues closer to a designer and physical facts closer to a modeler.

Abstraction hierarchies and resulting properties of models represented at different abstraction levels are, therefore, discussed in this chapter. Analog components and digital modules are outlined first. Mixed-signal systems need a special view on the interaction of time-continuous and event-discrete signals. Therefore, a new abstraction hierarchy is presented for such systems in order to address this problem accordingly.

2.1 Analog Components

The abstraction hierarchy to be be applied to analog entities (i.e., electronic components operating in both time- and value-continuous domains) has no such long tradition, and it is not yet as widely spread in the analog and mixed-signal circuit design community as its digital counterpart. Because of the restricted modeling expressivity found in SPICE-like simulators — still being the workhorse of many analog circuit designers — a structural view on design entities, which exploits at the utmost two abstraction levels known as *circuit* and *macro* level, respectively, has been biased. Meanwhile, there is an abstraction hierarchy in discussion, which is structured into a total of four levels denoted as *functional*, *behavioral*, *macro*, and *circuit* level [SV95].

Specification of behavior at each of these levels is addressed in the following. There is a wealth of excellent text books available on abstract specification of time-continuous operating systems on the one hand (e.g., [Cel91, MF95, SJN94]) and circuit theory and analysis on the other hand (e.g., [VS83, CL75]). The reader is referred to this rich literature for a more in-depth study of these topics.

Table 2.1 presents a summarized view on abstractions levels, associated modeling concepts, and properties of signals of analog circuits, which is based on [SV95]. The observable signals are time and value continuous at all of these levels, but there is a significant change in the signal quality when moving in a top-down manner through this hierarchy. From the behavioral abstraction level downward, all observable signals of a model entity are subjected to conservation laws. These are the well-known Kirchhoff's Rules in the electrical domain, but similar conservation laws also exist in other physical domains.

These abstraction levels seem to be strongly related to analog circuit design only — especially levels 3 and 4. However, an introduction of the abstraction levels 1 and 2 — *functional* and *behavioral* — is well suited for modeling purposes in other engineering domains.

As already mentioned, abstraction levels for time-continuous systems are not yet that well-agreed upon as their counterparts for digital circuits and systems. According to Table 2.1 we will discuss these four levels in terms of modeling methods and observable signals beginning with the highest abstraction level.

Table 2.1. Abstraction hierarchy for analog components and blocks

Level	*Modeling Method*	*Observable Signals*
functional	description of signal flow Input/Output relations by means of mathematical functions	time and value continuous Conservative Laws **need not** to be considered
behavioral	equations that describe the relations between port variables of an entity	time and value continuous Conservative Laws **are** considered
macro	hierarchical composition using ideal functional blocks	same properties as at behavioral level
circuit	hierarchical composition based on discrete components	same properties as at behavioral level

Functional Level. The mapping from input to output signals takes place by both algebraic and differential equations. This analytical specification may be visualized by a signal flow representation as found in the bondgraph approach [Tho90], for example. When denoting the inputs by $\vec{u}$, the outputs by $\vec{y}$, the internal state signals of the model by $\vec{x}$, and the time derivative of $\vec{x}$ by $\dot{\vec{x}}$, then the specification of the block functionality generally leads to a nonlinear Differential Algebraic Equation system (*DAE*) in time domain summarized as

$$\begin{aligned} \vec{F}\left(t,\dot{\vec{x}}(t),\vec{x}(t),\vec{u}(t)\right) &= \vec{0}\,, \\ \vec{G}(t,\vec{x}(t),\vec{y}(t)) &= \vec{0}\,. \end{aligned} \tag{2.1}$$

In the presence of a block with internal states showing a linear functional relationship between inputs, states and outputs, respectively, then the general implicit DAE formulation of Eq. (2.1) may be represented by the well-known explicit form given in Eq. (2.2).

$$\begin{aligned} \dot{\vec{x}} &= A\cdot x+B\cdot\vec{u} \\ \vec{y} &= C\cdot\vec{x}+D\cdot\vec{u} \end{aligned} \tag{2.2}$$

A more complex functionality of a system is derived from an aggregation of simpler blocks which are each specified at functional level. The resulting representation yields a block diagram, a wide spread specification means in automation and control engineering as well as in digital signal processing as illustrated in Chapter 1. Several specification entry and validation tool sets support this kind of functional block aggregation such as the Matlab/Simulink suite or the Khoros[1] environment [RW91].

Behavioral Level. A specification of functionality takes place by means of DAE systems. This seems to be very similar to the functional level, but there is a significant difference in the quality of the still time- and value-continuous signals: conservation laws have to be considered. Now, each signal is composed of two quantities derived from a branch definition: an *effort* and and a *flow* partition. These physical quantities are voltage and current, respectively, in the electrical domain. In general, behavioral level models are formulated using *hybrid port definitions* as a basis. Linear n-port blocks may thus be modeled using the following well-known hybrid equations:

$$\begin{aligned} i_i &= h_{11}\cdot v_i+h_{12}\cdot i_o\,, \\ v_o &= h_{21}\cdot v_i+h_{22}\cdot i_o\,. \end{aligned} \tag{2.3}$$

[1] Khoros is a trademark of the University of New Mexico

Taking internal state variables of the model into consideration, the general relations for a nonlinear block then result in a DAE system

$$\vec{f}\left(t,\vec{v}(t),\vec{i}(t),\vec{x}(t),\dot{\vec{x}}(t)\right)=\vec{0} \tag{2.4}$$

which has to be evaluated — as for functional level models — by numerical integration and nonlinear equation solving methods during transient simulation. Note that conservation law considerations require that the previously separated input $\vec{u}$ and output $\vec{y}$ signals of an entity specified at functional level have to be combined into a voltage $\vec{v}$ and current $\vec{i}$ vector each, respectively, as given in Eq. (2.4).

Macro Level. This is the highest level of abstraction available in SPICE-like simulators due to their lack of support of direct DAE system specifications as needed at the behavioral and functional level, respectively. The modeling method simply consists of a hierarchical, stuctural composition of ideal functional blocks and components specified in their behavior by some set of linear or nonlinear equations represented in polynomial form. These relationships are then mapped accordingly to single or multiple value-controlled sources. Because of the considerable work having been done in this area over the past 20 years or so [CC92], macro modeling is still a common modeling approach in integrated circuit design practice as will be demonstrated in Chapter 6. Its relevance, however, is expected to decrease in the years to come.

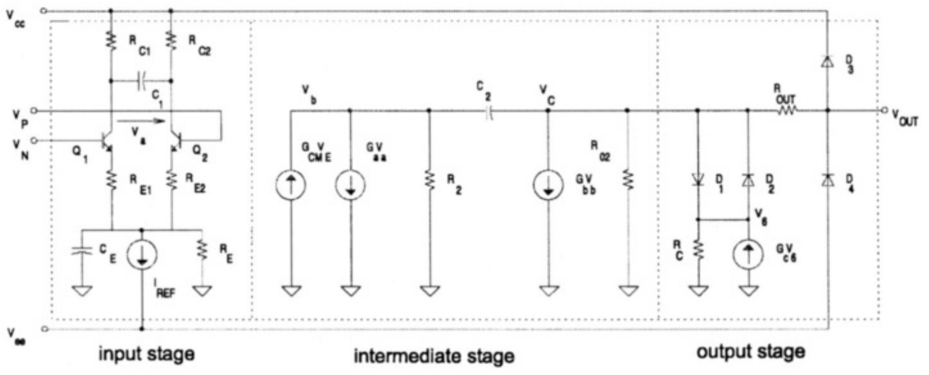


Figure 2.1. Macro level model of an operational amplifier

The generic macro level model of an operational amplifier — the fundamental building block of many analog and mixed-signal circuits — as depicted in Fig. 2.1 highlights the underlying basic modeling method. In addition to primitive components such as resistors, capacitors, and transistors there are *controlled voltage sources* present in this model. By appropriately defining the control functions, it is possible — to a certain extent — to represent a mapping of the input voltage waveform to the resulting output voltage in terms of accuracy. Note that macro models, in general, feature a galvanic separation of the input from the output signal on one hand as visible from Fig. 2.1 and a mapping of input to output signal values on the other hand by means of some algebraic specification introduced as polynomials. In contrast to Eq. (2.3), the calculation of companion voltages or currents at input and output ports, respectively, becomes much easier now.

Circuit Level. At this abstraction level a purely structural description is exploited, which consists of instances of passive and active primitive components and their interconnection. This description is also known as a *netlist,* or as a *schematic* when represented in a graphic-symbolical form. The signals to be considered here are again voltage and current waveforms in DC and in time domain. Figs. 2.2 and 2.3 depict a nonlinear CMOS circuit schematic and the resulting input and output waveforms generated from a transient simulation run.

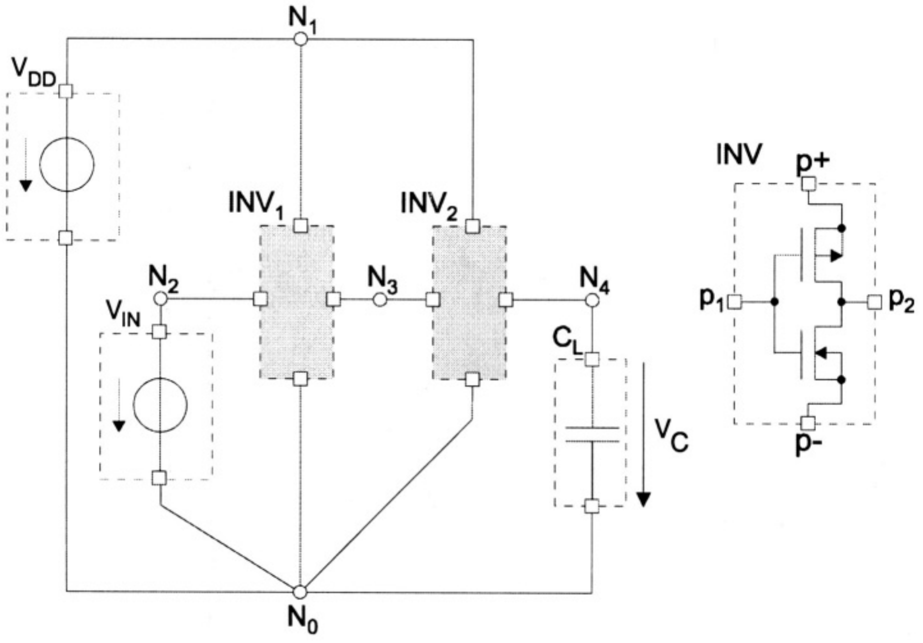


Figure 2.2. Schematic of a nonlinear CMOS circuit

At this point the question arises how a behavioral description is introduced to a circuit level simulator when using just a netlist representation of the block to be analyzed. The basic approach is known as the *functional structure* of the

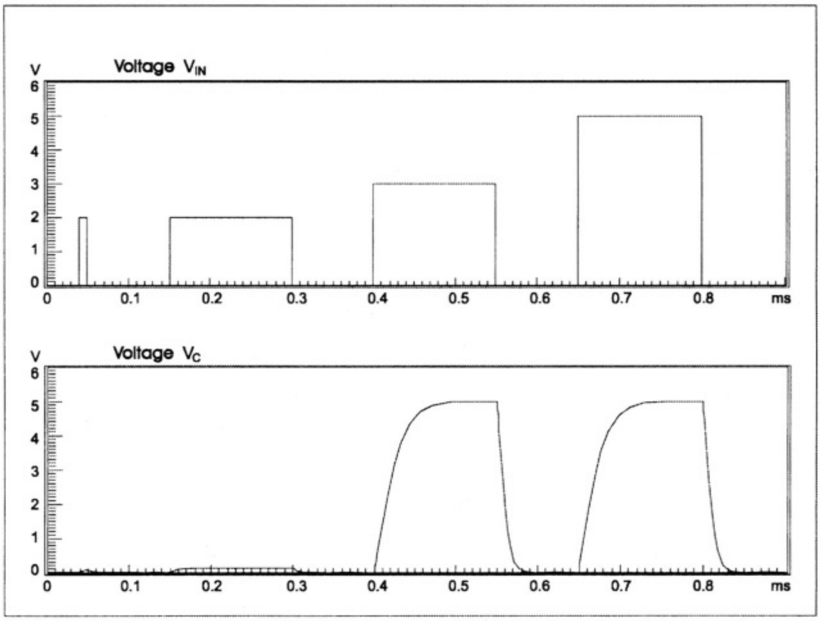


Figure 2.3. Results of transient simulation

circuit which consists of both an interconnection of primitive entities (i.e., the topology) and their behavior, denoted either as part of an external component library or directly in the simulator program code as found in the early versions of SPICE. The topological structure and the component behavior denoted as sets of equations are then used by the simulator to set up internal data structures according to an underlying analysis method such as the Modified Nodal Analysis [HRB73].

A netlist N_L, therefore, is a bipartite graph consisting of a set of vertices denoted as V_P (the *pins*), a set of vertices denoted as V_N (the *nodes*), and a set of edges denoted as E (the connecting *wires*) resulting in

$$\begin{aligned} N_L \quad &:= \quad \{V_P, V_N, E\} \\ & \qquad v_{Ni} \xrightarrow{e} v_{Pi} \,. \end{aligned} \tag{2.5}$$

The connectivity relation C is given by

$$C \subseteq (V_N \times V_P)\,, \quad e \in C \tag{2.6}$$

such that any vertex $v \in V_N$ may be connected to many v_{Pi}, $i = 1, \ldots, I$, but each v_{Pi} is to be connected to just one v_{N_i} only. This restriction follows from the semantic of $v \in V_N$ — it denotes the carrier of a unique value of the electrical potential (the *node voltage*) and it is not feasible, therefore, to assign different values to one and the same node. As an example, Fig. 2.4 depicts the specification of behavior for a primitive component. A *voltage/current characteristic function* (i.e., the constitutive element equation) denoted as g is introduced for this purpose, which is defined with respect to the vertex definitions given above. Note that g may either be a linear function as for an ohmic resistor or a nonlinear one as required to describe a general resistance element.

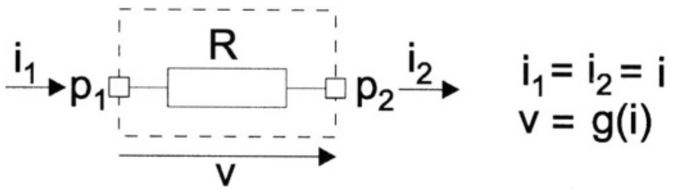


Figure 2.4. Specification of component behavior

From the definitions of signal properties given above it becomes clear that models at any of the levels *circuit, macro,* and *behavioral* as denoted in Table 2.1 fulfill the requirements of a behavioral model. Their functionality is described by equations or algorithms (explicitly or implicitly) and their interface signals are subjected to conservation laws.

A problem rises from the situation of denoting both an abstraction level and a fundamental property of models at different abstraction levels by one and the same term: ”behavioral”. In the author’s view this is one of the main reasons for the considerable confusion one notices in the circuit design and modeling communities when it comes to discussions on ”behavioral” models.

2.2 Digital Modules

Abstraction hierarchies for digital function blocks are introduced as found in literature and discussed in more detail because they are the formal basis for model representation by means of hardware description languages.

In a long ongoing development process a meanwhile widely accepted order of abstraction levels for digital design techniques was developed (e.g. [Arm89, GV94]). Levels of abstraction are thus well defined in the digital domain. They are addressed in the following, taking the summarized representation of Table 2.2 as a basis. From the viewpoint of the hardware modeler, the levels of abstraction form the foundation for design object representation.

Circuit Level. The circuit consists of interconnected active and passive components. Circuit behavior within time domain can be represented by means of a system of nonlinear differential equations. This level does not really belong to the digital abstraction hierarchy, it acts as a link to physical representations of digital circuits as discussed in the previous section.

Gate Level. On the gate level combinatorial and sequential blocks are the primitives. This level is also known as the *logic* level. The Boolean circuit algebra is represented by gates with additional simple memory elements such as Flip Flops. Specification is achieved with Boolean equations and state transition tables. However, the Boolean circuit algebra does not directly address the important delay time of the components. Therefore, time values must be attributed by additional models.

Register Transfer Level. On this level complex functional blocks are provided, such as register, counter, multiplexor, and storage modules. Specification takes place similar to the gate level, however, with enhanced Boolean equations, truth- or state transition tables. In addition, microoperations are addressed.

Chip Level. The chip level is located above the register transfer level. This level addresses the abstract modeling of microprocessors, memory, and bus controllers as primitive elements. Specification results from concurrently denoted descriptions of the input/output functionality. Therefore, this level is also known as the *algorithmic* level. Software designers determine the behavior on this level with programs by means of instruction sets of components. Programming on this level is addressed by the term *microprogramming.*

PMS (Processor, Memory, Switch) Level. The topmost level features processors, memory banks, and bus systems. The design of complete processor and system architectures is the main objective at this level. At the same time this level represents the connection to the functional levels of information processing systems addressing software architectures. Spatial and temporal ordering of activities (i.e., *allocation* and *scheduling*) as well as communication issues are of main interest on this abstraction level, which is sometimes donoted as *system* level.

Views, abstraction levels, modeling methods, structural primitives, and time models as summarized in Table 2.2 are related to the observable values produced by models of digital systems. These models may be represented in a hardware description language such as Verilog or VHDL. We will focus on the latter in the following for the reasons already mentioned in Chapter 1.

2.3 Mixed-signal Systems

Mixed-signal circuits and systems constructed thereof are characterized by the fact that they operate in time domain, but they generate and consume both value-continuous and value-discrete signals. Fig. 2.5 depicts the typical data exchange within a mixed-signal system. The analog partition derives value- and time-continuous signals from numerically solving a generally nonlinear DAE system, whereas the digital partition works on value- and time-discrete

Table 2.2. Abstraction hierarchy for digital modules and systems

View	*Level*	*Modeling Method*	*Structural Primitive*	*Time Model*	*Observable Values*
imperative	*PMS*	cooperating processing units	CPU, memory, bus	causality	freely definable
	chip	parallel algorithms	controller, RAM, ROM, UART	discrete (fine/coarse granularity)	interpreted words (freely definable)
reactive	*register transfer*	guarded commands	register, counter, ALU, multiplexor	discrete (coarse granularity)	bit fields (not interpreted, multi-valued)
	gate	Boolean logic equations	gate, Flip Flop	discrete (fine granularity)	multi-valued logic
	circuit	differential equations	transistor, R, L, C	continuous	continuous (voltage, current)

signals determined from an event processing algorithm. The signal values have to be converted from one domain to the other in order to accomplish the communication as depicted in Fig. 2.5.

An even more difficult operation consists of combining the analog and digital time base of these partitions, respectively, as to accommodate their interaction. Merging of time bases yields the *universal* scale of the mixed-signal system as depicted in Fig. 2.6. The distinct time points on the analog time scale stem from the numerical integration algorithms exploited for the solution of the DAE system. The actual time point is thus determined from incrementing the past time by a chiefly nonconstant, but discrete amount: the *integration time step*. Note that Fig. 2.6 gives only a simplistic view on the analog-digital interaction. In general, roll-backs on the time scale have to be considered, which are a major challenge to simulation tool providers.

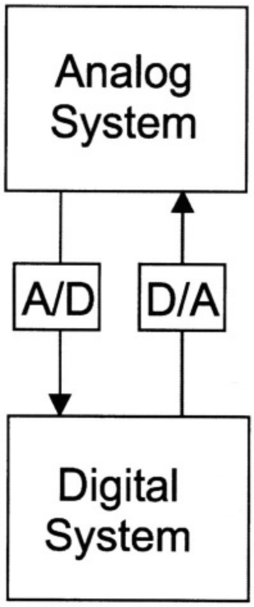


Figure 2.5. Internal communication within a mixed-signal system

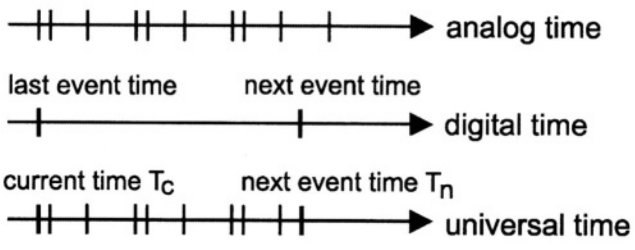
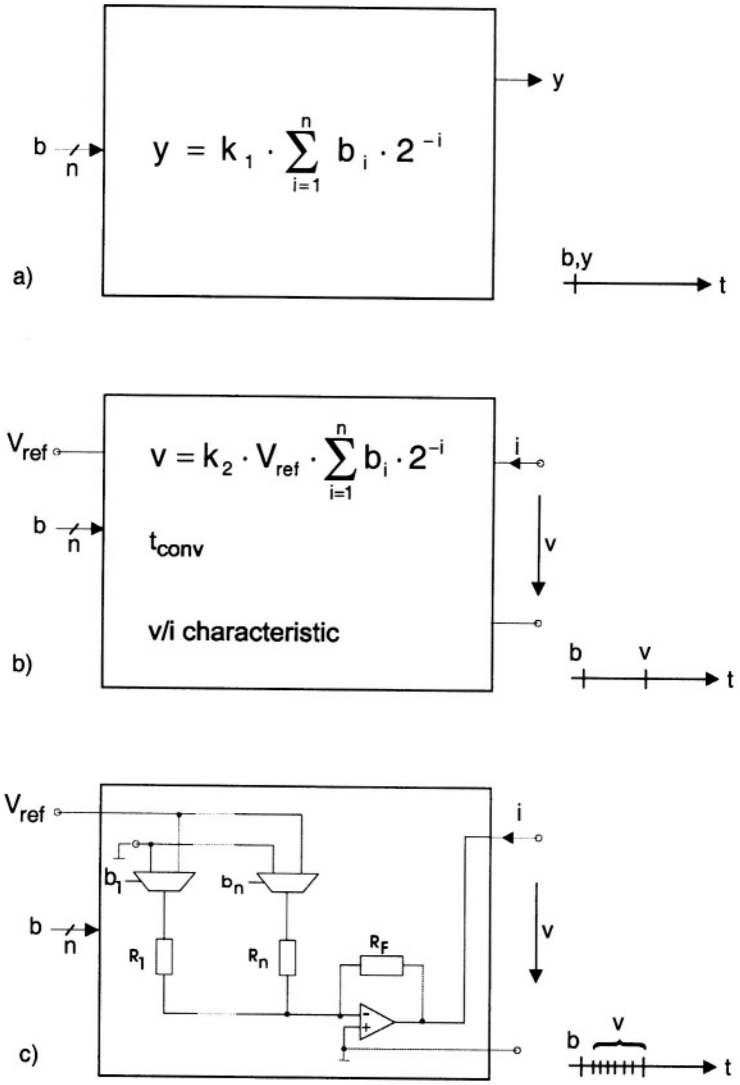


Figure 2.6. Time scales in mixed-signal systems

Each of the partitions may be denoted on either abstraction level as long as the conversion of the value domains and the time bases are addressed. This requirement is demonstrated for an n bit D/A converter as depicted in Fig. 2.7. The digital systems partition has to provide a bit-level representation of its computation no matter how the calculation has been performed internally. The delivery time point of the result is implicitely given by an event. This situation is depicted in Fig. 2.7a).

An abstract functional model of the converter simply maps the binary data word denoted as b to the time- and value-continuous signal y according to the detailed conversion algorithm. Its result becomes available almost at the same



$y = k_1 \cdot \sum_{i=1}^{n} b_i \cdot 2^{-i}$
b
n
y
a)
b,y
t
V_{ref}
$v = k_2 \cdot V_{ref} \cdot \sum_{i=1}^{n} b_i \cdot 2^{-i}$
t_{conv}
v/i characteristic
i
v
b)
b
v
t
V_{ref}
b_1
b_n
R_1
R_n
R_F
i
v
c)
b
v
t

Figure 2.7. D/A conversion a) Functional model b) Behavioral model c) Circuit model

point in time as the input data arrives. This is highlighted on the time scale next to the block output signal. Fig. 2.7 b) depicts a behavioral model of the converter according to the terminology of Table 2.1, which is a refinement of the functional model as easily visible from Fig. 2.7. Its output now consists of a voltage and current waveform, respectively, according to the generic definition of behavioral models. This is reflected in both the conversion algorithm and in the required *v/i* characteristic at the output pin of the block. Moreover, the conversion time t_{conv} is now part of the block specification, and an additional connector (V_{ref}) is introduced. However, the output waveform still unveils a digital-like behavior as highlighted on the associated time scale. In contrast, Fig. 2.7 c) depicts the schematic of a very simple, but feasible circuit implementation. Because of the practically analog structure of the circuit — the multiplexors in the schematic may be viewed as switches which, in turn are implemented by transistors — the output waveform is now much more analog-like as visible from the time scale next to the block output. It becomes clear that the output waveforms of the behavioral and the circuit model, respectively, are not identical, at least not for those time intervals in which transitions of the output signal take place. Despite of a definition of pin signals as electrical quantities, behavioral models of converters, in general, do not reflect the detailed circuit behavior (i.e., the signal waveforms) of a block as long as the underlying modeling concept relies on an abstract view of their functionality only. *Pin compatibility,* a frequently stated requirement when it comes to model hierarchies, cannot be provided in an easy way. This is because both the amount of connectors and their properties differ across the abstraction hierarchy as illustrated in Fig. 2.8.

When considering different implementation styles of one and the same functionality, one is faced with a design space which, in many cases, covers the whole range of signal and operation regions. This is highlighted for some possible implementation styles of a filter block — a rather simple and, at a first glance, a purely analog functional block operating in frequency domain. Although aimed to derive the output signal amplitude and phase in this domain from a manipulation of the input signal, the real block operation takes place in time domain. The basic specification of the filter behavior consequently is denoted by differential or by finite difference equations. The resulting filter characteristic in frequency domain is then derived from these representations by means of well-known transform operations. However, depending on the chosen implementation style of the abstract block functionality, the resulting module may be viewed as either an analog, a mixed-signal, or a digital circuit, respectively, as detailed in Table 2.3.

A similar situation arises with the representation of behavior of at least A/D and D/A converters in mind because these functional blocks provide the direct link between the analog and the digital subsystems as depicted in Fig. 2.5 — they are in some sense part of both domains. Rather different representations (i.e., modeling styles) may be applied, therefore, for these blocks especially at the behavioral abstraction level. Although the port signals of such a model have to consider conservation laws, one easily may denote the intrinsic conversion approach — such as flash, successive approximation, Σ/Δ, etc. — directly by an algorithmic (i.e., more digital-like representation) or by some other, more analog-like modeling style. The effects of different modeling styles as high-

Table 2.3. Different implementation styles of a filter block

Analog	*Mixed-Signal*	*Digital*
passive RLC		
Surface Accoustic Wave	active Switched Capacitator	digital filter architecture
active RC		

lighted in Fig. 2.7 are addressed and discussed in more detail in Chapter 5 and 6, respectively.

A redefinition of abstraction levels seems to be appropriate for mixed-signal circuits at least because of the multiple meanings of the term "behavioral". In addition, terms for the denomination of abstraction levels as given in Table 2.1, which stem from views and concepts coined with SPICE-like simulators in mind, should be updated in order to reflect the new and fundamental model representation features supported by modern HDL.

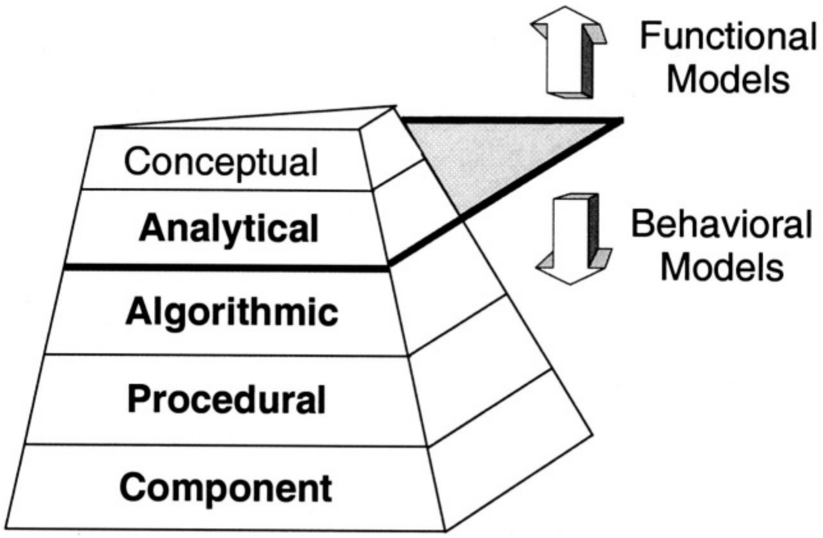


Figure 2.8. Abstraction hierarchy for mixed-signal systems

Fig. 2.8 visualizes the proposed abstraction hierarchy, which is subdivided into *functional* and *behavioral* model classes. The functional model class may consist of several abstraction levels. A *conceptual* level may be present in this class, which possibly exploits temporal logic and some flow calculus for functionality specification. However, the impact on mixed-signal systems modeling of such an abstract representation is not yet clear. The *analytical* level of Fig. 2.8, therefore, is assumed to be the highest abstraction level of practical interest for the time being.

Analytical Level. The topmost level of the functional model class addresses a specification of functionality by idealized analytical equations for continuous signals and by abstract processing for discrete signals. The fundamental concept on this level is causal modeling.

Algorithmic Level. Behavioral models usually exploit the acausal modeling concept for continuous signals, which are subjected to conservation laws, and causal/acausal representations of event processing algorithms. Both the control and the data flow within the model have to be denoted at algorithmic level. The specification of event processing, which addresses different timing resolutions, is thus straight forward. Continuous behavior is refined into disjoint operation regions of the model, which are to be represented accordingly by sets of analytical equations (i.e., data flow). Selections on the explicit equations to be executed at a given point in time take place according to parameter values, which define the active operation region (i.e., control flow).

Procedural Level. This level addresses refinement operations of an algorithmic model into modules. The resulting modules (i.e., *procedures* of the model code) represent either model entities of available building blocks or processing units in a similar way to structured software code. The procedural abstraction level relates to the macro level of Table 2.1. However, the latter is a subset of this level because a macro is just a primitive instance of a procedure in terms of software engineering.

Component Level. The component level is aimed at a direct representation of stuctural netlists of circuits, which are composed from primitive elements, such as transistors or logic gates. Their behavior has to be defined as part of the associated component library models.

The interrelation between conceptual models, their executable representations, and the executing simulator is highly complex. Fig. 2.9 details this relationship. An appropriate support of model instances is thus mandatory. This is covered by highly expressive modeling languages such as VHDL-AMS, which originated recently from the well-known hardware description language VHDL for digital systems. This HDL is discussed in the following Chapter.

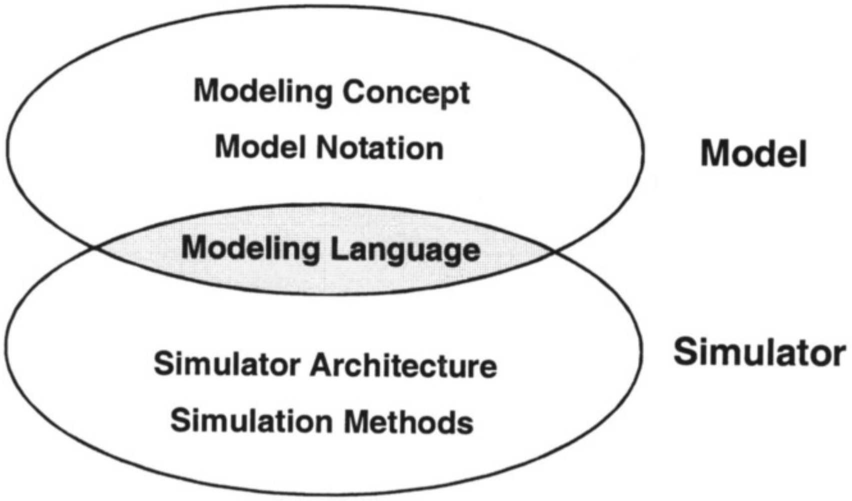


Figure 2.9. Interrelationship of model and simulator

Chapter 3

MODEL REPRESENTATION

Plenty of good literature is available on digital hardware description languages, especially on VHDL, and on synthesis algorithms, which exploit such languages as a means for specifying the functionality of digital components and systems. Continuous time and mixed-signal or multi-nature modeling, however, is not yet covered sufficiently. In [Cel91] the reader will find a detailed discussion on modeling of time-continuous systems. An introduction to fundamental numerical algorithms for solving differential-algebraic equations is given in [BCP89]. Modeling issues with an emphasis on analog circuits is addressed in [MF95] and [BLR95].

VHDL-AMS has been proposed as a highly expressive description language for executable representations of mixed time-continuous and event-discrete operating circuits and systems. The language reference of VHDL-AMS is given in [IEE99], the document of IEEE Standard 1076.1. [Men99] is referenced as an example for a user's manual for a commercial simulator. Finally, in [VB97] the modeling of multi-nature systems using VHDL-AMS is discussed by several authors, thus giving a first insight into this emerging area.

3.1 Fundamentals of VHDL

VHDL stands for VHSIC Hardware Description Language, whereby VHSIC is the abbreviation for Very High Speed Integrated Circuits. In 1980 the VHSIC project was launched by the US Department of Defense. It is regarded as an attempt to standardize the hardware description language. To that date, several basic approaches for modeling existed on the register transfer level in addition to the Boolean equations and SPICE circuit net lists as well as standard programming languages such as Basic, Pascal, or Lisp. These languages were used mainly to implement procedures for Petri networks or decision trees. In 1983 Texas Instruments, IBM, and INTERMETRICS were assigned to develop an implementation after predefining the requirements for VHDL. Language specifications were completed in 1987, and the first simulator was made public. In the same year IEEE began the implementation of their own VHDL version, which became the industrial Standard IEEE 1076 in the year of 1988.

As an example for the modeling capabilities of VHDL a register with linear feedback (LFSR) will be discussed. With an initial value of unequal zero, a cyclic sequence over all bit vectors with 3 bits unequal to zero is to be generated. Fig. 3.2 (left side) shows the block circuit diagram.

In VHDL the entity LFSR is defined in Fig. 3.1 as a black box with one input and three outputs — as to be seen in Fig. 3.2 on the right side.

```
entity LFSR is
  port(
    clock: in Bit;
    Q1, Q2, Q3 : out Bit);
end entity LFSR;
```

Figure 3.1. Entity declaration of LFSR

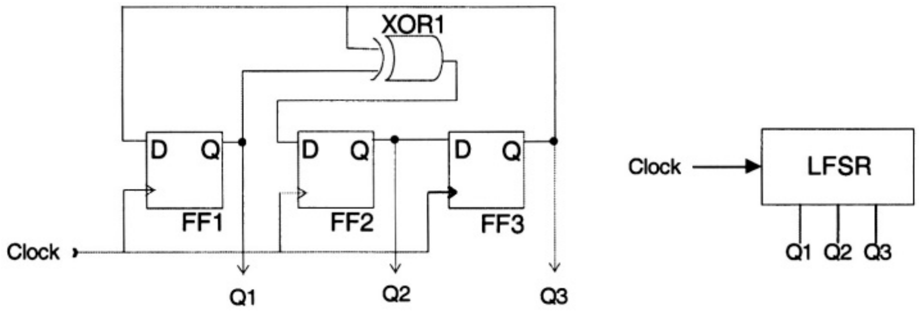


Figure 3.2. Block and black box schematic of LFSR

Several architectures can now be defined and exploited for this entity in VHDL, whereby the designer may choose the best suitable architecture for her or his purposes without having to face restrictions or changes in the functionality.

3.1.1 Behavior

Behavior specification corresponds with an algorithm that produces quasi-random sequences by calculating a primitive polynomial with coefficients of modulo 2 within an infinite loop as defined in Fig. 3.3.

```
architecture behavior of LFSR is
begin
  process
    variable state: Bit_vector (3 downto 0) := "0111";
  begin
    -- wait for clock
    wait until clock = '1';
    -- shift
    state := state (2 downto 0) & '0';
    if state(3)='1' then
      state := state xor "1011";
    end if;
    Q3 <= state(2) after 5 ns;
    Q2 <= state(1) after 5 ns;
    Q1 <= state(0) after 5 ns;
  end process;
end behavior;
```

Figure 3.3. Behavioral model of LFSR

3.1.2 Data flow

If the specification is to be orientated towards the data flow in the block, a feedback is constructed with three guarded signal assignments as given in Fig. 3.4:

```
architecture dataflow of LFSR is
  signal FF1,FF2,FF3 : Bit := '1';
begin
  -- wait for clock
  b1 : block(clock = '1' and not clock'stable)
  begin
    FF3 <= guarded FF2 after 5 ns;
    FF2 <= guarded FF1 xor FF3 after 5 ns;
    FF1 <= guarded FF3 after 5 ns;
  end block;
  -- external connection
  Q3 <= FF3;
  Q2 <= FF2;
  Q1 <= FF1;
end dataflow;
```

Figure 3.4. Data flow model of LFSR

3.1.3 Structure

The architecture `structure` is the direct result of the schematic as shown in Fig. 3.2. Structure specifications can be mapped directly to VHDL as detailed in Fig. 3.5.

```
architecture structure of LFSR is
  signal xor_out: Bit;
  signal SR1, SR2, SR3: Bit := '1';

  component FF
    port (clock, data: in Bit;
          Q: out Bit);
  end component;

  component XORgate
    port (a,b : in Bit;
          x: out Bit);
  end component;

begin
  FF1: FF port map (clock, SR3, SR1);
  FF2: FF port map (clock, xor_out, SR2);
  FF3: FF port map (clock, SR2, SR3);
  XOR1: XORgate port map (SR1, SR3, xor_out);
  Q3 <= SR3; Q2 <= SR2; Q1 <= SR1;
end structure;
```

Figure 3.5. Structural model of LFSR

3.1.4 Relations of models

All three models (i.e., architectures of the LFSR entity) accomplish the same tasks and may be interchanged at will. For instance, a testbench, which contains a stimulus for the LFSR, can invoke all three models without changing the overall behavior. The VHDL code of Fig. 3.6 shows how the data flow-oriented specification is instantiated in a testbench and is stimulated by a clock signal.

3.2 Introduction to VHDL-AMS

The basic concepts of VHDL form the foundation of the extension of this hardware description language to the modeling domain of Analog Mixed Signal (i.e., AMS) resulting in the new IEEE Standard 1076.1-1999. These concepts of VHDL are summarized in the following:

```
-- Stimulus for the LFSR
entity LFSRstim is
end LFSRstim;

architecture test of LFSRstim is
  component LFSR
    port (clock: in Bit;
          Q1, Q2, Q3: out Bit);
  end component;
  signal clock : Bit := '0';
  signal val: Bit_vector(3 downto 1);
begin
  -- instantiation of LFSR
  L1: LFSR port map(clock,val(1), val(2), val(3));
  --  clock generator
  ClkGen: process
  begin
    for I in 1 to 20 loop
      wait for 1 ns;
      clock <= not clock;
    end loop;
    wait;
  end process ClkGen;
end test;

-- configuration of LFSR: use dataflow architecture
configuration DATAconf of LFSRstim is
  -- architecture of LFSRstim
  for test
    for L1: LFSR use entity work.LFSR(dataflow);
    end for;
  end for;
end DATAconf;
```

Figure 3.6. Testbench of LFSR

- Model composition from communicating, concurrently active design entities.
- Strict separation of the unique interface description and the functional description(s) of an entity.
- Execution of the model description based on the Stimulus/Answer paradigm. Events are the only stimuli considered, thus resulting in an emphasis on digital circuits and systems.
- Definition of the simulation process on top of an event-driven simulation cycle.

The simulation process of a model encoded in VHDL is thus built around the event-driven simulation cycle as outlined in the following.

Simulation process:

```
while transactions remain to be processed
   increase time until a transaction has to be processed
   update the state from the previous transactions
   determine events caused by updates
   perform a simulation cycle
   based on new state and events
end while
```

3.2.1 Design objects

The description of a design object denoted as an `entity` consists of its interface declaration, of one or more architectural bodies, and of an optional configuration declaration. Their syntactical details are as follows:

Entity Syntax:

```
EntityDeclaration ::=
  entity Identifier is
     EntityHeader
     EntityDeclarativePart
   [begin
     EntityStatementPart ]
  end [entity] [EntitySimpleName];
EntityHeader ::=
   [FormalGenericClause]
   [FormalPortClause]
GenericClause ::=
  generic (GenericList);
PortClause ::=
  port (PortList);
```

Architecture Syntax:

```
ArchitectureBody ::=
  architecture Identifier of EntityName is
     ArchitectureDeclarativePart
  begin
     ArchitectureStatementPart
  end  [architecture] [ArchitectureSimpleName];
ArchitectureStatement ::= ConcurrentStatement
```

The IEEE Standard 1076 of VHDL has been extended considerably to cope with modeling requirements in the time-continuous domain and especially with mixed-signal, multi-nature systems (i.e., event-discrete and time-continuous behaviors of heterogeneous models). This objective is depicted in Fig. 3.7.

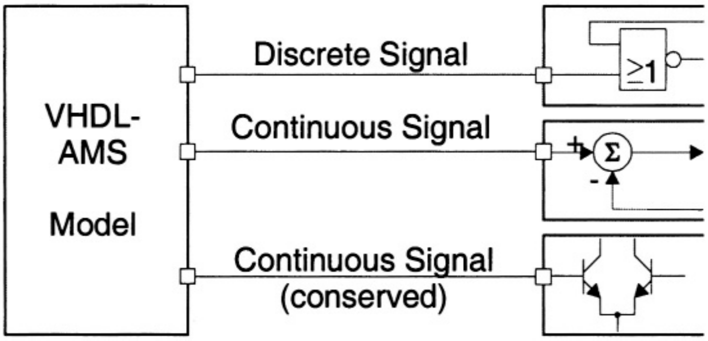


Figure 3.7. Combination of signal classes

A fundamental problem arises when combining time-discrete and time-continuous signals in mixed-signal circuits: The semantic of connections must be determined and implemented by appropriate methods as detailed in the previous chapters. This is highlighted in Fig. 3.8.

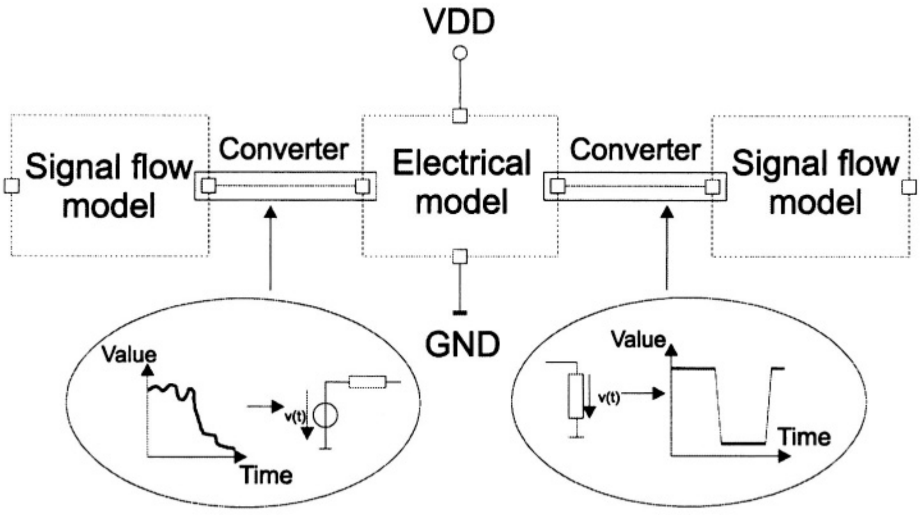


Figure 3.8. Connecting functional and behavioral class models

3.2.2 Extensions to VHDL

Obviously, the range of VHDL was by far not sufficient to suit the additional requirements of mixed-signal systems. Therefore, extensions to design objects and declarations, to assignments, to attributes, and especially to the simulation cycle were necessary. These extensions to the original VHDL language are depicted in Fig. 3.9.

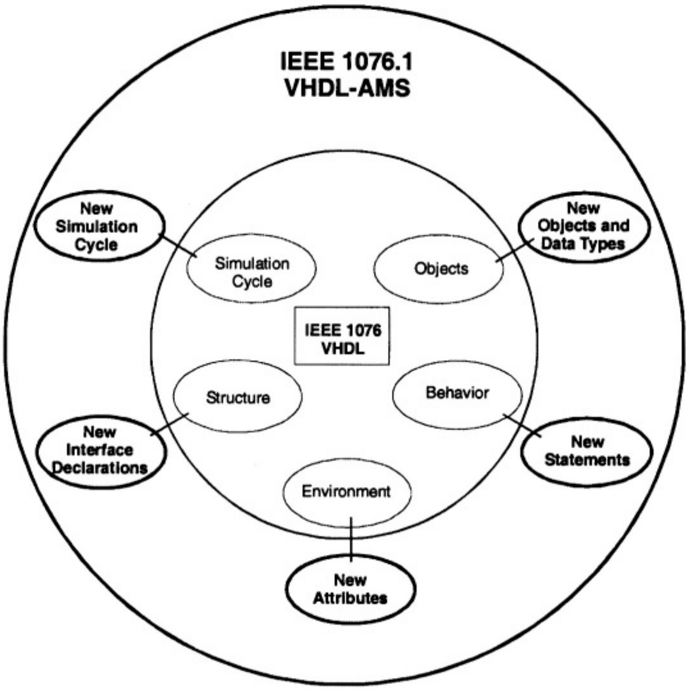


Figure 3.9. Extension to IEEE Standard 1076-1993

3.2.2.1 Quantities and Terminals

Descriptions of value- and time-continuous variables and the enforcement of conservation in different physical domains, such as Kirchhoff's Rules in the electrical domain, are possible when introducing the basic concepts of `quantities` and `terminals`.

The object class `quantity` has the following properties: Representation of time-dependent functions, partially continuous with a finite number of discontinuities in one time interval, assignment of physical dimension and existence of derivatives and integrals, which may be viewed as *membership functions.* Quantities may be defined as *free, branch,* or *source* quantities, Fig. 3.10 shows their classification. Note that mixed descriptions in terms of domains, modes, and technologies are achievable with this single basic concept.

The enforcement of conservation is of considerable interest when modeling physical systems at all, except for the functional modeling level. It has been

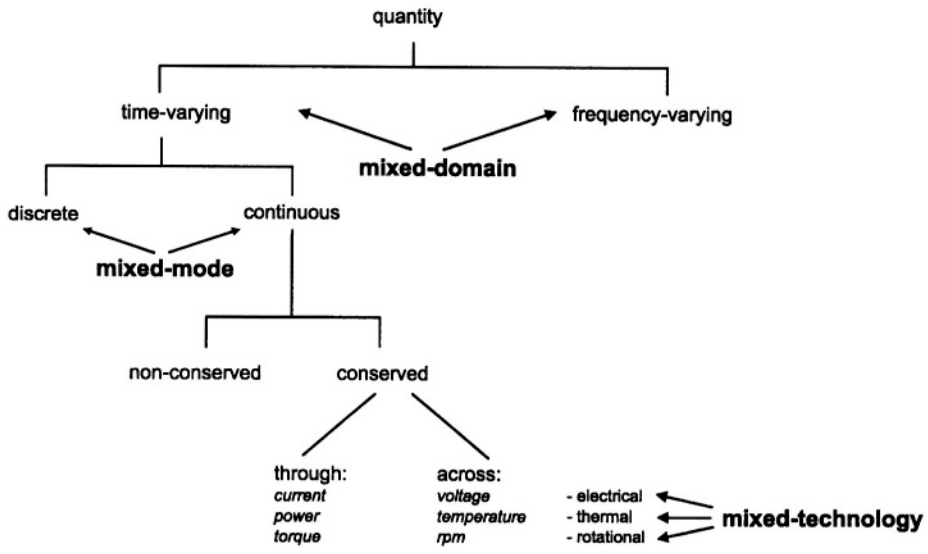


Figure 3.10. Classification of `quantities`

tackled in VHDL-AMS by a graph-based concept. Branch quantities with reference to `terminals` and `natures` are used for this purpose as detailed in the following.

- Quantities between two `terminals` represent the unknown variables (branch quantities) in a conservative system of equations:
  - `across` quantity: difference of potentials
  - `through` quantity: flow in a branch.
- Terminals are assigned to a physical domain known as `nature`, which has to be defined appropriately.

Fig. 3.11 shows how two branches (I1, I2) between two `terminals` (T1, T2) related to the `nature` Electrical as shown in Fig. 3.12 may be defined, thus resulting in `V` as the unique voltage and in `I1, I2` as the branch currents between T1 and T2, respectively.

3.2.2.2 Conservation Laws

Conservation laws have to be enforced independently of a user-defined partitioning of the entire model into design entities. This implies that the interface description of entities requires an enhancement. The connection pins of an entity, denoted as `port` in VHDL terminology, may now carry waveforms of

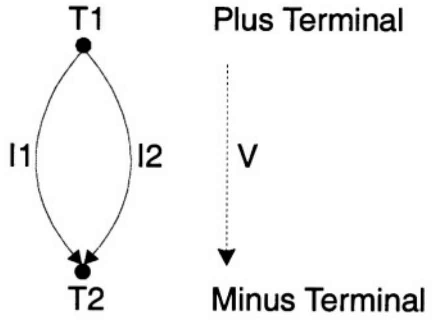


Figure 3.11. Simple example for across and through quantities

```
subtype Voltage is Real;
subtype Current is Real;
nature Electrical is
  Voltage across
  Current through
  ElectricalGround reference;

terminal T1, T2: Electrical;
quantity V across I1, I2 through T1 to T2;
```

Figure 3.12. Definition of nature Electrical and some branch quantities

different qualities: digital signals and continuous quantities of some physical domain with or without conservation requirements as given in Table 3.1 and depicted in Fig. 3.7. The new interface description syntax of VHDL-AMS is given in the following. Note that the compatibility for VHDL 1076 is accomplished by a blank `PortAttribute`, which is interpreted as equivalent to the `signal` attribute.

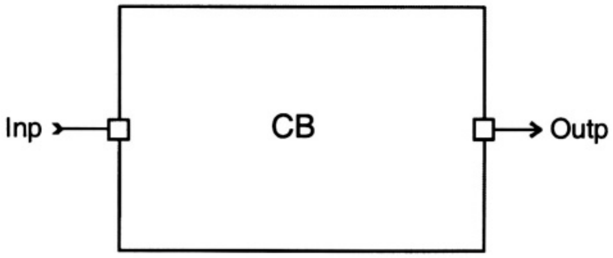
```
port (PortAttribute NameList: [Mode] Type)
     PortAttribute ::= signal | quantity | terminal
```

Definitions of port attributes, internal nodes and branches as well as external branches are first addressed for nonconservative ports. Fig. 3.13 a) depicts a functional block featuring one input and output each. The port signals are time and value continuous but not subjected to conservation laws. The block may represent the functionality of some component being part of a control loop.

The port quantities are thus part of the class non-conserved as depicted in Fig. 3.10. They are denoted as free quantities in VHDL-AMS and are derived either from the port declarations as shown in Fig. 3.13 b) or from a direct

Table 3.1. Values of mode subject to `PortAttribute`

PortAttribute	*Mode*	*Remarks*
`signal`	`in | out | inout`	uni- or bidirectional
`quantity`	`in | out`	unidiectional
`terminal`	———	No mode allowed because a `terminal` is always bidirectional.



a)

```
entity CB is
  generic ( ... );
  port (quantity Inp: in Real;
        quantity Outp: out Real)
end entity CB;

architecture A of CB is
begin
  -- representation of behavior
  ...
end architecture A;
```

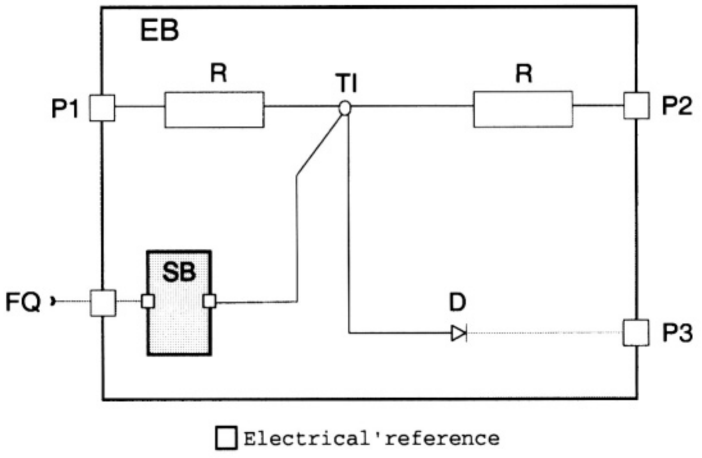
b)

Figure 3.13. Nonconservative block a) Connectors b) Code template for causal modeling

definition for quantities internal to an architecture according to the general definition syntax:

```
quantity identifier_list : [in | out] subtype_indication
                                       [ := static_expression]
```

Conservative signals are defined according to Fig. 3.11 by terminals acting as nodes for branches. This is highlighted for block EB of Fig. 3.14, which features both conserved and non-conserved connectors.



a)

```
entity EB is
  generic ( ... );
  port (quantity FQ : in Real;
        terminal P1, P2, P3 : Electrical);
end entity EB;

architecture B of EB is
  terminal TI : Electrical;
  quantity V1 across I1 through P1;
  quantity V2 across I2 through P2;
  quantity V3 across I3 through P3;
begin
  ...
end architecture B;
```

b)

Figure 3.14. Mixed causal/acausal block model a) Mixed connector attributes b) Code template

In addition to the free input quantity *FQ* there are three `port terminals` *P1, P2,* and *P3* as well as an internal `terminal` *TI* acting as electrical nodes. Terminals are formally defined by

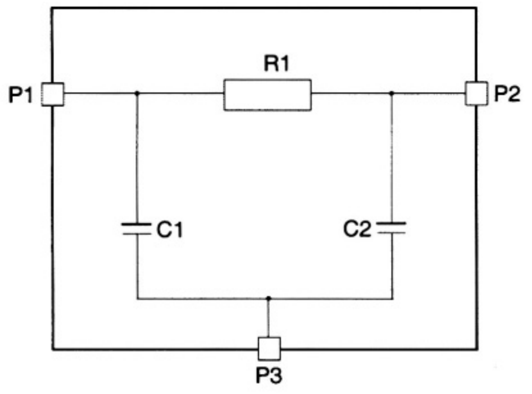
```
terminal identifier_list : nature_identifier;
```

They do not contain any data. Quantities related to terminals are to be defined as part of the architecture declaration as given in Fig. 3.14 b). Note that these branch quantities address implicitly the reference of the `nature` Electrical (i.e., the electrical ground).

Eq. (3.1) denotes the DAE of a block featuring free input and output quantities $\vec{q}_{in}$ and $\vec{q}_{out}$, respectively, in addition to branch quantities. This forms the basis for mixed causal/acausal specifications. Eq. (2.4) as introduced for the description of a behavioral model is then just a subset of Eq. (3.1). Consequently, both functional and behavioral models at any abstraction level of Table 2.1 or Fig. 2.8 may be combined into one `entity` according to Eq. (3.1).

$$\vec{f}\left(\vec{v},\vec{i},\vec{x},\dot{\vec{x}},\vec{q}_{in},\dot{\vec{q}}_{in},\vec{q}_{out}\right)=\vec{0} \tag{3.1}$$

Modeling of analog behavior is supported in quite different ways and is demonstrated by the simple circuit example depicted in Fig. 3.15. Fig. 3.15b) denotes its description in SPICE syntax.



a)

```
.SUBCKT RC P1 P2 P3

R1  P1  P2      1.0
C1  P1  P3      1.0E-0.9
C2  P2  P3      1.0E-0.9
.ENDS RC
```

b)

Figure 3.15. Simple analog circuit a) Schematic b) SPICE code

The equations defining component behavior are not directly visible from Fig. 3.15, they are implicitly assigned to key characters for component classes in SPICE syntax (e.g., 'R': Resistor, 'C': capacitor, 'X': subcircuit). In contrast, behavior definition is explicitly denoted in VHDL-AMS. Fig. 3.16 details entity declarations for resistor and capacitor components, respectively. Inital conditions regarding the charge of the capacitors are omitted from Fig. 3.16 for simplicity reasons. The simulation model of the circuit of Fig. 3.15 may now be stated in two different ways: either as a structural or as an equation-based description at component level as detailed in Fig. 3.17. Note that both models are behavioral descriptions of the RC circuit of Fig. 3.15.

These simulation models compare nicely to the following digital models at gate level: Fig. 3.17 a) to a gate netlist and Fig. 3.17 b) to a data flow description of a set of Boolean equations, respectively.

```
entity R is
  generic (R_val : Real := 1.0);
  port (terminal P1, P2 : Electrical);
end entity R;

architecture Ideal of R is
  quantity V across I through P1 to P2;
begin
  V == R_val * I;
end architecture Ideal;
```

a)

```
entity C is
  generic (C_val : Real := 1.0E-12);
  port (terminal P1, P2 : Electrical);
end entity C;

architecture Ideal of C is
  quantity V across I through P1 to P2;
begin
  I == C_val * V'dot;
end architecture Ideal;
```

b)

Figure 3.16. Coding of component behavior a) Resistor model b) Capacitor model

3.2.2.3 Representation of Behavior

The description of the functionality of a model is captured in the architectural body. In addition to the concurrent statements of VHDL used for digital circuits and systems (i.e., component instances, signal assignments, and process statements), there now are *simultaneous statements* available for the specification of continuous behavior by means of mathematical equations as follows:

```
architectureStatement ::=
  concurrentStatement | simultaneousStatement
simultaneousStatement ::=
  simpleSimStatement | compoundSimStatement
 [ label:] [ pure | impure ] simpleExpression
  == simpleExpression [toleranceAspect];
```

Simultaneous statements can be either simple or compound. The latter form may consist of a `procedural` or a conditional `use` statement. The simple statement as defined above unveils two interesting properties. First, it is a noncausal formulation of an equation. This means that the == symbol denotes equality

```
entity RC_2P is
  generic (Res  : Real  := 1.0;
           Cap  : Real  := 1.0E-09);
  port (terminal P1, P2, P3 : Electrical);
end entity RC_2P;

architecture Struct of RC_2P is
begin
  C1 : entity C(Ideal)
    generic map (C_val => Cap)
    port map (P1 => P1, P2 => P3);
  C2 : entity C(Ideal)
    generic map (C_val => Cap)
    port map (P1 => P2, P2 => P3);
  R1 : entity R(Ideal)
    generic map (R_val => Res)
    port map (P1 => P1, P2 => P2);
end architecture Struct;
```

a)

```
entity RC_2P is
  generic (Res  : Real  := 1.0;
           Cap  : Real  := 1.0E-09);
  port (terminal P1, P2, P3 : Electrical);
end entity RC_2P;

architecture DAE of RC_2P is
  quantity V1 across I1 through P1 to P3;
  quantity V2 across I2 through P2 to P3;
  quantity IC1, IC2, IR1 : Real;
begin
  IC1 == Cap * V1'dot;
  IC2 == Cap * V2'dot;
  V1 - V2 == Res * IR1;
  I1 - IR1 - IC1 == 0.0;
  I2 - IR1 + IC2 == 0.0;
end architecture DAE;
```

b)

Figure 3.17. Code of RC circuit a) Netlist b) Network equations

in the mathematical sense, not an assignment such as the <= symbol in digital data flow descriptions. As a consequence, the modeler has no need to formulate explicit equations. The more general implicit differential algebraic equation form (DAE) may be used directly instead. The second interesting property is related to the optional tolerance aspect of the statement. This is intended to specify the numerical accuracy of the equality, a very general way to deal with numerical properties. Usually, a modeler is interested only in accuracy aspects but not in selecting solution methods for numerical integration, for instance. Appropriate solvers are best provided by the implementors of a VHDL-AMS simulation engine, whereas a modeler specifies application specific accuracy levels only by means of tolerance aspects.

Statements within the architectural body of a design entity may consist of both concurrent and simultaneous statements — as already detailed. This is of special interest to a modeler because it supports a consistent description of design objects operating in mixed-mode without an artificial separation into event-discrete and time-continuous partitions. Mixing up event-oriented and time-continuous behavior in many cases results in dealing with model discontinuities. This problem is addressed by the `break` statement used for both initialization purposes and for discontinuity handling.

```
entity DAC is
  generic ( Vhigh: real := 5.0;
            Vlow: real := 0.0;
            ConvDly: time := 10 ns);
  port( signal S: in bit;
        terminal AnPin: Electrical)
end entity DAC;

architecture Gen of DAC is
  signal Sint: bit; -- internal copy of input signal S
  -- output branch quantities with implicit reference
  -- to Ground
  quantity Vout across Iout through AnPin;
begin
  Sint <= S'delayed(ConvDly); -- assign the DAC
  -- conversion delay by shifting the event time
  if Sint = '0' use
  -- calculate branch quantity values
    Vout == Vlow;
  else
    Vout == Vhigh;
  end use;
  break when Sint'event; -- an external event on S
  -- forces the recalculation of branch quantity
  -- values after ConvDly time units
end architecture Gen;
```

Figure 3.18. Generic model code of a D/A converter

A simple example of an 1 bit D/A converter as shown in Fig. 3.18 illustrates its usage. The mixed-mode property of the `entity DAC` is easily visible from its interface description: There is both a `signal` and a `terminal` definition at ports. Obviously, the model addresses the behavioral modeling at algorithmic abstraction level. The branch quantities related to the `terminal` *A* linked to the `nature` Electrical are defined in the architectural body. There are four statements in the executable part of the architectural body: A signal assignment, a compound simultaneous statement (`if ... use ... else ... end use`), the `break` statement which is sensitive to an event on signal *S* and a simple simultaneous statement. The output voltage of the D/A converter (i.e., the value of *V*) is recalculated each time the value of *S* changes. Note that additional equations or definitions are not required in order to enforce conservation laws at `port terminal` *A*.

3.2.2.4 Model Flow Using VHDL-AMS

VHDL-AMS offers a variety of advantages for mixed-signal and for general systems modeling. This is due to a consistent and easy way to integrate time-continuous and event-discrete computation, and last but not least, due to the presence of a comprehensive set of design tools for the automation of digital signal processing. The refinement of functional blocks and of components as outlined in Fig. 1.2 is thus well supported. This situation is detailed in Fig. 3.19 for the flow of models in mixed-signal, multi-nature systems.

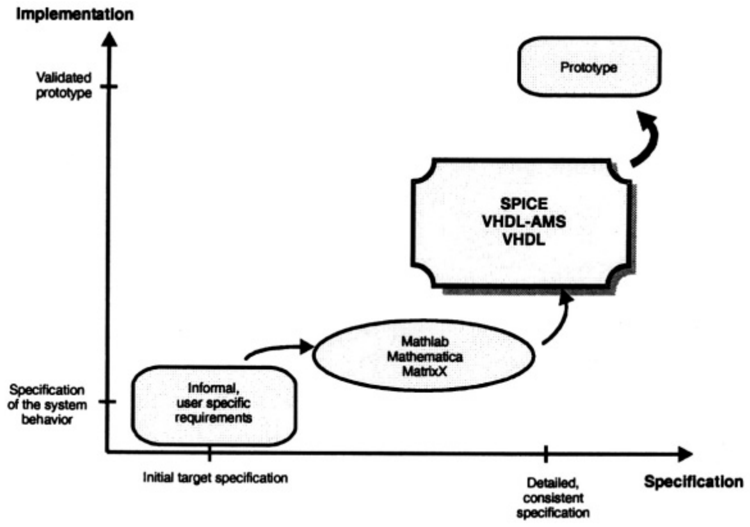

Implementation
Validated prototype
Specification of the system behavior
Prototype
SPICE
VHDL-AMS
VHDL
Mathlab
Mathematica
MatrixX
Informal, user specific requirements
Initial target specification
Detailed, consistent specification
Specification

Figure 3.19. Flow of model instances in systems design

However, two major drawbacks of VHDL-AMS should be mentioned. First, *generic modeling* across the borders of physical domains and engineering disciplines is not supported. A simple example demonstrates this drawback.

Fig. 3.20 depicts a generic harmonic oscillator in time domain. This block computes the output $y(t)$ from the inputs p_1, p_2, and p_3 by an ordinary differential equation (ODE).

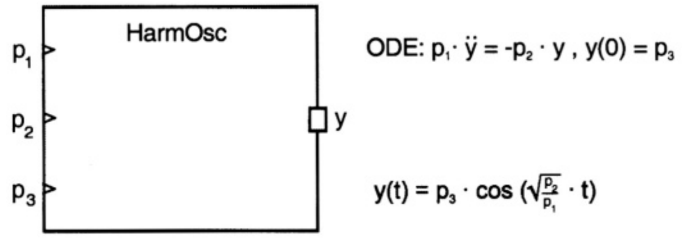

HarmOsc
p1
p2
p3
y
ODE: $p_1 \cdot \ddot{y} = -p_2 \cdot y$, $y(0) = p_3$
$y(t) = p_3 \cdot \cos(\sqrt{\frac{p_2}{p_1}} \cdot t)$

Figure 3.20. Generic harmonic oscillator

When trying to refine the generic model of Fig. 3.20 into an electrical and a mechanical version of a behavioral model one is faced with a major problem. Although the `architecture` sections of these models are almost identical, the `entity` declarations are not as obvious from Fig. 3.21.

```
entity HarmOsc_El is
 generic (P1, P2, P3 : Real);
 port (terminal Y : Electric);
end entity HarmOsc_El;
```

a)

```
entity HarmOsc_Mech is
 generic (P1, P2, P3 : Real);
 port (terminal Y : Mechanic);
end entity HarmOsc_Mech;
```

b)

Figure 3.21. Entity declarations for implementation variants a) Electrical b) Mechanical

This situation arises from the fact that `terminals` (i.e., the objects acting as anchors for branch quantity definitions) are to be linked to an explicit `nature`: in our case either Electrical or Mechanical. Object-oriented modeling principles such as *inheritance* and *implementation* are not present in VHDL-AMS. This restriction does not affect modeling of mixed-signal circuits too much because circuit design usually takes place in the electrical domain only. Modeling of multi-nature systems, however, is constrained by this lack of generic modeling features.

The second restriction is noticed when trying to refine functional models into behavioral ones. Free port quantities of functional models and branch quantities present at the ports of behavioral models are defined in VHDL-AMS in a completely different way. The latter have to be declared as part of the `architecture` section of the model based on `port terminal` definitions — completely in contrast to the input and output quantities of functional models. Thus, there is no automatic and universal way to model refinement across the functional-behavioral barrier. Fig. 3.22 visualizes this situation. The quality of ports differs in the functional and behavioral model classes, respectively, excluding `port signal` connectors, however. In addition, behavioral models sometimes feature connectors which are not present at higher abstraction levels. This specific property as highlighted in Fig. 3.22 was detailed in Fig. 2.7.

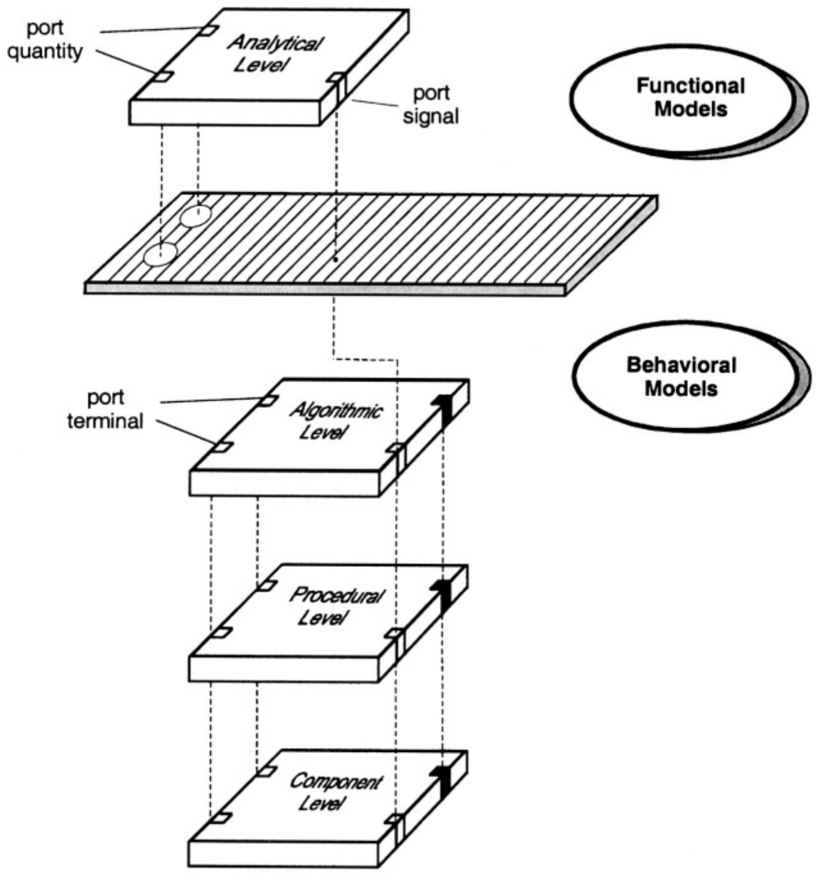


Figure 3.22. Pin compatibility of functional and behavioral models

3.2.2.5 Multi-Nature Systems

The modeling features of VDHL-AMS have been highlighted so far for electrical circuits only. Now, a simple multi-nature modeling example will demonstrate that all required concepts and syntax elements are already present for this purpose. First, we need to extend our set of `natures` (i.e., physical domains). The following package example summarizes the definitions for two additional physical domains. They are given as part of a `package` identified by the name `NaturePkg` as illustrated in Fig. 3.23.

```
package NaturePkg is
  -- physical constants
  constant Eps0 : Real := 8.8542e-12;
  -- vacuum permitivity [F/m]
  constant Mu0 : Real := 1.256e-6;
  -- vacuum permeability [H/m]
  constant Boltzmann : Real := 1.381e-23; -- [J/K]
  constant ElemCharge : Real := 1.602e-19; -- [C]
  constant AmbTemp : Real := 300.0; -- [K]
  -- electrical systems
  subtype Voltage is Real; -- subtypes for voltage
  subtype Current is Real; -- and current
  nature Electrical is Voltage across Current through;
  alias Ground is Electrical'reference;
  -- fluidic systems
  subtype Pressure is Real; -- subtypes for pressure
  subtype FlowRate is Real; -- and flow rate
  nature Fluid is Pressure across FlowRate through;
  alias FluidGround is Fluid'reference;
  -- thermal systems
  subtype Temperature is Real; -- subtypes for temperature
  subtype PowerTh is Real; -- and power
  nature Thermal is Temperature across PowerTh through;
  alias ThermalGround is Thermal'reference;
end package NaturePkg;
```

Figure 3.23. Subset of a multi-nature package [LAR+97]

A combined electrical and thermal model of a diode demonstrates how the effects of self-heating on its electrical characteristics may be modeled for a multi-nature application in an easy and consistent way [CB97]. This model is outlined in Fig. 3.24 and 3.25, respectively.

Passing of parameter values is accomplished by the `generic` mechanism of the entity declaration. There is one `terminal port` for the `nature` Thermal and two `terminal ports` for the `nature` Electrical. The associated branch quantities are defined in the architectural body. In addition, the two free quantities denoted as *Q* and *VT*, respectively, are defined as well. The first three simultaneous equations within the executable part of the `architecture` denote the electrical behavior of the nonlinear diode. Note the differential equation relating the charge *Q* to the current *IC*. The thermal voltage is now determined by means of the constants *Boltzmann* and *ElemCharge* taken from `NaturePkg` as a function of the temperature *Temp,* which in turn is one of the thermal branch quantities. The second one, *Power,* is calculated from the electrical branch quantities *V* and *ID* in the last equation. The diode current *ID* thus depends on the junction temperature via the thermal voltage *VT*.

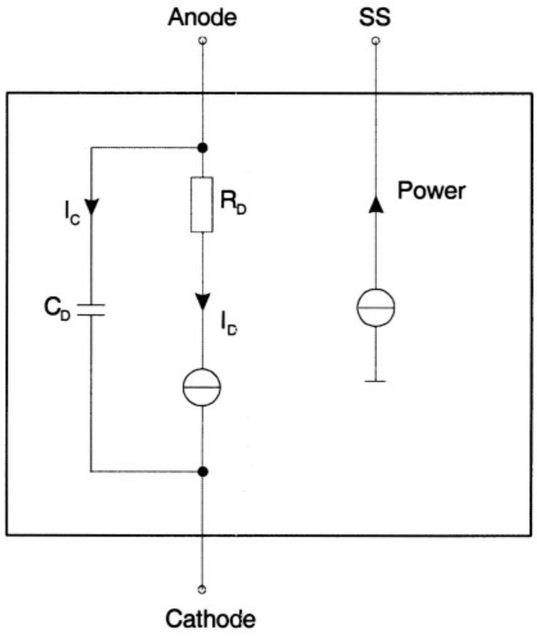


Figure 3.24. Structural multi-nature model of a diode

```
entity DiodeTh is
  generic(ISO : Real := 1.0E-14;
          N,Area : Real := 1.0;
          Tau,CJO,Phi,RD : Real := 0.0 );
  port(terminal Anode, Cathode : Electrical;
       terminal SS : Thermal ):
end entity DiodeTh;

architecture BNRealTH of DiodeTh is
  quantity V across ID,IC through Anode to Cathode;
  quantity Temp across Power through ThermalGround to SS;
  quantity Q : Charge;
  quantity VT : Voltage;
begin
  ID == Area*ISO*(exp((V-RD*ID)/(N*VT))-1.0);
  Q == Tau*ID-2.0*CJO*sqrt(Phi*(Phi-V));
  IC == Q'dot;
  Temp*Boltzmann/ElemCharge == VT;
  V*ID == Power;
end architecture BNRealTh;
```

Figure 3.25. Multi-nature model of a diode

3.3 Mixed-Signal, Multi-Nature Modeling Example

The increasing complexity of information processing products and the pressure of design to market requirements are the main reasons for a change in the design process of these products, resulting in an emphasis of timely design validation of *virtual prototypes* instead of breadboarding. This reveals possible concept errors long before an implementation of real prototypes takes place. On the other hand, such products generally are too complex to be completely modeled at lower (i.e., more accurate) abstraction levels. A need of a sensible mix of models on different abstraction levels thus arises resulting in the necessity to represent and to execute models both in different domains and at several abstraction levels. The purpose of such a system model is to support an assessment of the conceptualization and the refinement according to the design flow outlined in Fig. 1.2 in terms of meeting initial target specifications. Modeling of multi-nature systems in context with mechatronic and microsystems is an evolving application area of model engineering [Rom98, Kas00].

The process of model definition, refinement, and assessment is demonstrated in the following by means of a depth gauge — a safety critical device for deep water divers. The demonstrator was built with system simulation of multi-nature design objects in mind. It is a fairly complex, but still easily comprehensive example on how to construct a simulation model in three physical domains featuring both time-continuous and event-discrete (i.e., mixed-signal) behavior. In addition, synthesizable models of digital signal processing units are incorporated at register and algorithmic levels as defined in Table 2.2.

All VHDL-AMS models are operational and were thoroughly tested by means of the ADVanceMS simulator from Mentor Graphics Corp. Unfortunately, the complete set cannot be presented here for space restriction reasons. However, all models of this depth gauge and additional information on other complex multi-nature, mixed-signal modeling projects are available from the author´s web site.

3.3.1 Partitioning the System

The models are partitioned in two ways. The whole system is subdivided into smaller entities, which are each modeled as a unit. Furthermore, there are several models (i.e., `architectures`) present for each unit. They differ in terms of abstraction levels and accuracy or represent different implementation styles. The architectures suited for system simulation are emphasized in the following according to Fig. 3.26.

The functional partitioning of the model of a depth gauge denoted *Virtual Diving Computer (VDC) [Klu01]* is based on considerations of

- functional decomposition

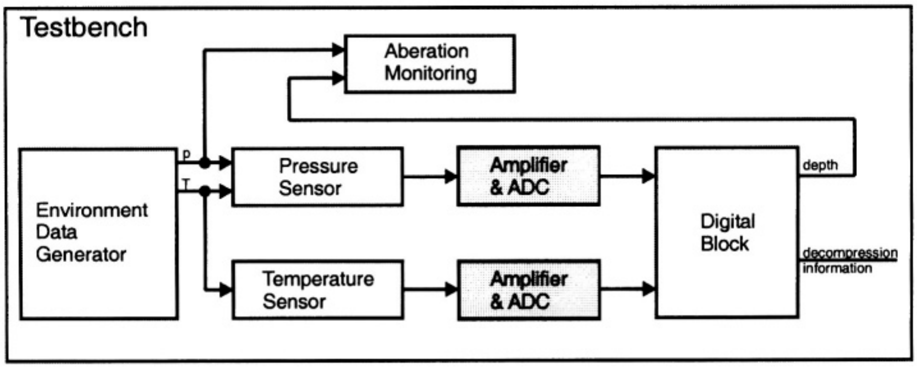


Figure 3.26. Composition of the depth gauge

- entity reuse
- complexity
- testability.

The different architectures can be classified by means of the architecture names. Most of them are self-explanatory.

- The **simple** architecture is limited to fundamental features only, it is modeled in the functional domain. It thus yields a low simulation cost and idealized results.
- The **digital_algorithmic** architecture is a high-level description using a synthesizable subset of VHDL. These models have to be passed to a scheduling and resource allocation tool prior to mapping to a target library. For the demonstrator this was accomplished with the Behavioral Compiler[1].
- The **digital_RTL** architecture is a register-transfer level description, again exploiting a synthesizable subset of VHDL. These models may reflect multi-clock or even asynchronous operating designs and are easily mapped to target libraries. The Design Compiler[2] was exercised for this purpose.
- The **beh** architecture is a behavioral model of analog or mixed- signal components. This model reproduces timing and signal waveforms of the entity including major hazards and glitches.
- The **eldo** architecture is used to incorporate SPICE (i.e., macro and circuit level) analog models.

[1]Behavioral Compiler is a trademark of Synopsys, Inc.
[2]Design Compiler is a trademark of Synopsys, Inc.

3.3.2 Models of VDC functional units

3.3.2.1 Testbench

The entity denoted Testbench is the top level module. It contains the environmental description and the device under test. The Testbench is self-contained and is used to define the interconnections of design entities. Moreover, it selects the architectures to be used. Its code is given in Fig. 3.27.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;
use work.all;

entity vdc_testbench is
end vdc_testbench;

architecture simple of vdc_testbench is
  constant T_ADC_Width : natural := 6;
  constant U_T_min : emf := 0.0;
  constant U_T_max : emf := 2.72;
  constant P_ADC_Width : natural := 12;
  constant U_P_min : emf := -0.03;
  constant U_P_max : emf := 0.188;
  constant Depth_Width : natural := 10;
  -- Derived constants:
  constant DSP_P_Min : integer := integer(1000.0 * U_P_Min);
  constant DSP_P_Max : integer := integer(1000.0 * U_P_Max);
  constant DSP_T_Min : integer := integer(1000.0 * U_T_Min);
  constant DSP_T_Max : integer := integer(1000.0 * U_T_Max);
  -- signals and terminals (internal to testbench)
  terminal Pressure_In : fluidic;
  terminal Temp_In : thermal;
  terminal U_Temp, U_Pressure : electrical;
  terminal a_Depth, a_Error : electrical;
  signal Clk : std_logic := '0';
  signal nClk : std_logic;      -- not(Clk)
  signal Rst : std_logic;       -- Reset
  signal T_Data : std_logic_vector((T_ADC_Width-1) downto 0);
  signal P_Data : std_logic_vector((P_ADC_Width-1) downto 0);
  signal D_Data : std_logic_vector((Depth_Width-1) downto 0);
```

Figure 3.27. Code of vdc_testbench

```
begin
  Rst <= '1', '0' after 5 ns;      -- Initialization
  Clk <= not(Clk) after 1 ms;      -- Clock Generator
  nClk <= not(Clk);                -- Inverted Clocksignal
  -- structural description
  ENVIRONMENT : entity vdc_sources(beh_LakeDive)
    port map (Pressure_In, Temp_In);
  TSENSOR : entity vdc_tsens
    port map (Temp_In, U_Temp);
  PSENSOR : entity vdc_psens
    port map (Pressure_In, Temp_In, U_Pressure_P,
              U_Pressure_N);
  T_ADC : entity vdc_ADC
    generic map (U_T_Min, U_T_Max, T_ADC_Width)
    port map (U_Temp, Electrical_Ground, Clk, Rst, T_Data);
  P_ADC : entity vdc_ADC
    generic map (U_P_Min, U_P_Max, P_ADC_Width)
    port map (U_Pressure_P, U_Pressure_N, Clk, Rst, P_Data);
  DSP : entity vdc_dsp
    generic map (Depth_Width, DSP_P_Min, DSP_P_Max,
                 DSP_T_Min, DSP_T_Max)
    port map (nClk, Rst, P_Data, T_Data, D_Data);
  TESTER : entity vdc_tester
    port map (Rst, D_Data, a_Depth, a_Error, Pressure_In);
end simple;
```

Figure 3.27 (continued). Code of vdc_testbench

3.3.2.2 Environmental Model

The environment is modeled such as to produce all the input data needed to test the depth gauge by model execution. As a consequence, only observable effects for the device under test are reproduced.

The generic `timescale` of Fig. 3.28 can be used as a bypass for simulators that do not allow a change in the resolution limit for the type `time`. Simulations based on femtoseconds reduce the guaranteed simulation time to 2.14 microseconds only. However, typical application scenarios cover several hours or even days. This requires range extensions beyond 2^{31} or changing the resolution limit to milliseconds. If both are not feasible then a `timescale` as to shrink the simulation duration may be used instead.

The port terminals `Pressure_sensor` and `Temp_sensor` are modeled as signal sources, which provide pressure and temperature potentials without side effects. These potentials may be derived from the diver's position. To simplify the model, only the resulting potentials are addressed in the following two architectures. The first one is characteristic for *lake diving*, the second one for *ocean diving*.

```
library disciplines;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;
entity vdc_sources is
  generic(timescale : real := 1.0);
  port(terminal Pressure_sensor : fluidic;
       terminal Temp_sensor : thermal);
end vdc_sources;
```

Figure 3.28. Entity declaration of vdc_sources

The model for the lake dive outlined in the following reproduces the strong coherence between diving depth and water temperature. Such a temperature profile is characteristical for deep lakes in summertime. The water surface is heated by the sun, whereas wind and day/night changes form a several meter deep warm surface layer. Below this layer the temperature drops rapidly to values less than 10 degree Celsius.

3.3.2.3 Temperature Sensor

The temperature sensor to be used is a platinum resistor made in thin layer technology on Al_2O_3 substrate — an off-the-shelf product. We use a $1k\,\Omega$ sensor with a temperature sensitivity (Tk) of $3,85 \cdot 10^3 / {}^\circ C$ as specified in IEC 751. These sensors are quite popular in automotive applications, thus resulting in low costs and good availability of subcomponents. The sensor is instantiated in a voltage divider circuit together with the constant resistor `Rfix`. The temperature information is represented by the output voltage `Uout`. The multi-nature property is thus obvious.

The `simple` architecture is based on the sensor *Nite PT1000* from Heraeus, Inc. as depicted in Fig. 3.30. Thus, the modeling process is constrained to an exploitation of commercially available components.

3.3.2.4 Analog/Digital Converter

The conversion is accomplished in a rather generic way. The underlying abstraction level is the algorithmic level. This is reflected in the basic converter function denoted as `Convert_AnalogToDigital`. Note that its input data is a continuous signal. The conversion time as well as its resolution are passed via the generic mechanism emphasizing the rather abstract view of this fundamental mixed-signal functional unit. The model compares directly to the modeling approach depicted in Fig. 2.7 b) without detailing the *v/i* characteristic at the converter's input terminals `U_in_N` and `U_in_P`.

```
library disciplines;
use disciplines.Fluidic_system.all;
use disciplines.Thermal_system.all;
library IEEE;
use IEEE.math_real.all;

architecture beh_LakeDive of vdc_sources is
  constant surfacePressure : pressure := 0.0;
  constant metalimnion : pressure := 1.2;
  constant surfaceTemp : temperature := 21.0;

  function LakeTemp (
    constant Tsurface : real;
    constant Psurface : real;
    constant Pmetalimnion : real;
    p : real)
    return real is

    constant Tc : real := 5.0 / (Pmetalimnion - Psurface);
    constant Tb : real := (Tsurface - 10.0) /
                          (exp(Tc * Psurface)
                           - exp(Tc * Pmetalimnion));
    constant Ta : real := Tsurface - Tb * exp(Tc * Psurface);
    constant Td : real := 4.0;
    constant Tf : real := (Tb * Tc * exp(Tc * metalimnion)) /
                          (10.0 - Td);
    constant Te : real := (10.0 - Td) / exp(Tf * metalimnion);
-- the 6 constants are calculated based on:
-- Temp(p=Psurface) == surfaceTemp
-- Temp(p=Pmetalimnion) == 10.0
-- Temp(p -> infty) == 4.0
-- Temp'dot(p=Psurface) * exp(5.0) == Temp'dot(p=Pmetalimnion)
-- Temp(metalimnion - delta ) == Temp(metalimnion + delta)
-- Temp'dot(metalimnion-delta ) == Temp'dot(metalimnion+delta)
    variable res : real;
  begin -- LakeTemp
    if (p < Pmetalimnion) then
      res := Ta + Tb * exp(Tc * p);
    else
      res := Td + Te * exp(Tf * p);
    end if;
    return res;
  end LakeTemp;
```

Figure 3.29. Architecture declaration of LakeDive

```
  quantity
    p across
    P_flow through
    Pressure_sensor;
  quantity
    Temp across
    T_P through
    Temp_sensor;
begin
  if (now < 0.1 * timescale) use        -- diving starts at the
    p == surfacePressure;               -- surface
  elsif (now < 3.0 * timescale) use     -- descending along the
    p'dot == 0.3 / timescale;           -- lake's profile
  elsif (now < 16.0 * timescale ) use   -- descending along the
    p'dot == 0.1 / timescale;           -- lake's profile
  elsif (now < 18.0 * timescale ) use   -- ground time: 2 min
    p'dot == 0.0;                   -- (cold, dark, few fish)
  elsif (p > 0.4) use               -- ascending up to 4m
    p'dot == - 0.75 / timescale;
  elsif (p > 0.3) use               -- diving in the epilimnion
    p'dot == - 0.005 / timescale;   -- (much fish, crayfish and
  elsif (p > surfacePressure) use   -- insects)
    p'dot == - 0.04 / timescale;    -- ascending to the surface
  else
    p == surfacePressure;           -- diving ends at the surface
  end use;
  Temp == LakeTemp(surfaceTemp, surfacePressure,
                   Metalimnion, p);
end beh_LakeDive;
```

Figure 3.29 (continued). Architecture declaration of LakeDive

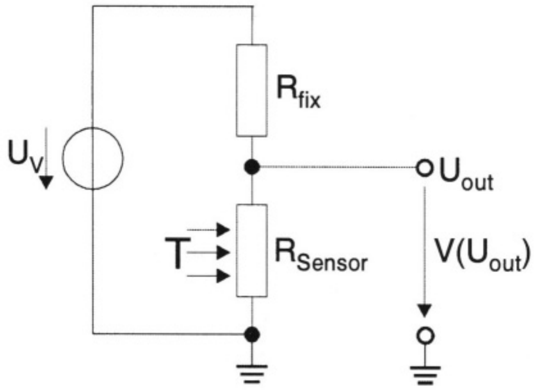


Figure 3.30. Equivalent circuit for a temperature sensor

```
library IEEE;
use ieee.math_real.all;
library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Thermal_system.all;

entity vdc_tsens is
  generic (
    R0 : real := 1000.0;
    Rfix : real := 1000.0;
    UV : emf := 5.0;
    Tk : real := 3.85e-3);
  port(terminal T_In : thermal;
       terminal Uout : electrical);
end vdc_tsens;

architecture simple of vdc_tsens is
  quantity R1 : real;
  quantity
    Temp across
    T_In;
  quantity
    Uout_V across
    Uout_I through
    Uout;
begin
  R1 == R0 * (1.0 + Tk * Temp);
  Uout_V == UV * R1 / (R1 + Rfix);
end simple;
```

Figure 3.31. Model of temperature sensor

3.3.2.5 Digital Block

The depth value is calculated in the digital part from data obtained through pressure and temperature measurements. It is then represented as electrical quantities. In the first intermediate step the actual temperature is determined using the temperature data represented in 6 bit accuracy. This is followed by the calculation of the actual pressure from the pressure data given in 12 bit accuracy. The conversion of the specified pressure to a depth value below the water surface is obtained by Eq. (3.2).

$$depth = \frac{1m}{0,1bar} \cdot (pressure - 1bar) \tag{3.2}$$

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library disciplines;
use disciplines.electromagnetic_system.all;

entity vdc_ADC is
  generic (
    MinVoltage : emf := -0.03
    MaxVoltage : emf := 0.188
    typDelay : time := 30 ns;
    Res : integer := 12);
  port(terminal U_in_P : Electrical;
       terminal U_in_N : Electrical;
       signal Clock, Reset : in std_logic;
       signal Data : out std_logic_vector((Res-1) downto 0));
end vdc_ADC;

architecture Analog_to_Digital of vdc_ADC is
  quantity U_In across U_In_P to U_In_N;

  function Convert_AnalogToDigital
    ( MinVoltage: real; MaxVoltage: real; Res: integer;
      U_in : real ) return Std_logic_vector is
    variable r : real;
    variable i : integer;
    variable u : unsigned(Res-1 downto 0);
    variable slv1 :std_logic_vector((Res-1) downto 0);
  begin
    r := (U_in - MinVoltage) / (MaxVoltage - MinVoltage);
    i := integer(r * (2.0**(Res-1)));
    u := conv_unsigned(i,u'length);
    slv1 := std_logic_vector(u);
    return slv1;
  end Convert_AnalogToDigital;
begin
  P1 : process (Clock, reset)
    variable r : real;
  begin -- process P1
    if reset = '1' then
      Data <= (others => '0') after typDelay;
    elsif Clk'event and Clk = '1' then
      Data <= Convert_AnalogToDigital(MinVoltage, MaxVoltage,
                                      Res, U_in) after typDelay;
    end if;
  end process P1;
end Analog_to_Digital;
```

Figure 3.32. Model of vdc_ADC converter

The model outlined in this section implements digital signal transformations aimed to compensate for the nonlinearities of the sensors. The depth value is calculated at a guaranteed accuracy of 0.1 m.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library DW02;
use DW02.DW02_components.all;

entity vdc_dsp is
  generic (
    D_Res : integer := 10;
    P_min : integer := -30;   -- in mV
    P_max : integer := 188;   -- in mV
    T_min : integer := 0;     -- in mV
    T_max : integer := 2720;  -- in mV
    Tk0  : integer := 260;    -- (Rfix)/(R0*Tk)
    Tk1  : integer := 5000;   -- UV in mV
    Tk2  : integer := -260;   -- -1/Tk
    Tp0  : integer := -10;    -- UOffset in mV
    Tp1  : integer := 10);    -- UV *k0 in mV
  port(
    Clk : in std_logic;
    Reset : in std_logic;
    Pdata : in std_logic_Vector;
    Tdata : in std_logic_Vector;
    Depth : out std_logic_Vector((D_Res-1) downto 0));
end vdc_dsp;
```

Figure 3.33. Entity declaration of vdc_dsp

The `digital_algorithmic` architecture related to the entity declaration in Fig. 3.33 makes use of resource sharing and scheduling features of state-of-the-art high-level synthesis tools, which results in a relatively small circuit complexity. The synthesized VHDL code yields one multiplication unit, one division unit, and some glue logic. The code given in Fig. 3.34, an algorithmic-level description, is fully synthesizable by the Behavioral Compiler.

This example highlights the features of VHDL as a subset of VHDL-AMS in denoting high-level, but still synthesizable models for the digital domain.

3.3.3 Simulation Results

The aberation monitor depicted in Fig. 3.36 and denoted as `vdc_tester` in Fig. 3.35 takes the environmental pressure data and the depth output of the digital block for the calculation of both an absolute error and an analog value

```
architecture digital_algorithmic of vdc_dsp is
begin -- behaviourDescription
  process
    variable k1 : integer;
    variable k2 : integer;
    variable Temp, Pressure : integer;
    variable i : integer;
    variable i32_1, i32_2, i32_3 : integer;
    variable b : Bit;
    variable s32_1 : signed (31 downto 0);
    variable u32_1 : unsigned (31 downto 0);
  begin
    Depth <= (others => '0');
    main_loop: loop
      wait until clk'event and clk='1';
      if reset='1' then
        exit main_loop;
      end if;
      Temp := conv_integer(unsigned(TData));
      i32_1 := Temp * (T_Max - T_Min);
      i32_2 := T_Min * (2 ** TData'length);
      Temp := i32_1 + i32_2;
      Pressure := conv_integer(unsigned(PData));
      i32_1 := Pressure * (P_Max - P_Min);
      i32_2 := P_Min * (2 ** PData'length);
      Pressure := i32_1 + i32_2;
      -- correction of sensor error:
      -- 28 bit resolution needed
      i32_1 := Tk0 * Temp;
      i32_2 := Tk1 * (2 ** TData'length);
      i32_2 := i32_2 - Temp;
      Temp := conv_integer(conv_signed(i32_1,32) /
                           conv_signed(i32_2,32));
      Temp := Temp + Tk2;
      Temp := Temp + 10;
      -- accuracy of Temp is 1 deg C
      -- Temp is T' = T + 10 deg C (because of T in (-10;50))
      -- k2 by linear approx.
      s32_1 := conv_signed(3 * Temp,32);
      u32_1 := conv_unsigned(5,32);
      i32_1 := conv_integer(shr(s32_1,u32_1));
      k2 := 253 + i32_1;
      -- k2 is k2' = k2 * 256
      -- correction of sensor error:
      -- first guess: k1'=256:
      k1 := 256;
```

Figure 3.34. Architecture declaration of vdc_dsp

```
      i32_1  := (Pressure - Tp0 * (2 ** PData'length));
      i32_1  := i32_1 * (25 * (2 ** (18 - PData'length)));
      i32_2  := Tp1 * k1;
      i32_2  := i32_2 * k2;
      i32_3  := conv_integer(conv_signed(i32_1,32) /
                             conv_signed(i32_2,32));
      -- calculation of k1' using the approximation:
      if (i32_3 < 320) then
        k1  := 256;
      else
        s32_1  := conv_signed(i32_3,32);
        u32_1  := conv_unsigned(14,32);
        i32_1  := conv_integer(shr(s32_1,u32_1));
        k1  :=i32_1 + 251;
      end if;
      i32_1  := (Pressure - Tp0 * (2 ** PData'length));
      i32_1  := i32_1 * (25 * (2 ** (18 - PData'length)));
      i32_2  := Tp1 * k1;
      i32_2  := i32_2 * k2;
      i32_3  := conv_integer( conv_signed(i32_1,32) /
                              conv_signed(i32_2,32));
      -- i32_3 = depth in factors of 10cm
      Depth <= std_logic_vector(
                            conv_unsigned(i32_3,Depth'length));
    end loop main_loop;
  end process;
end digital_algorithmic;
```

Figure 3.34 (continued). Architecture declaration of vdc_dsp

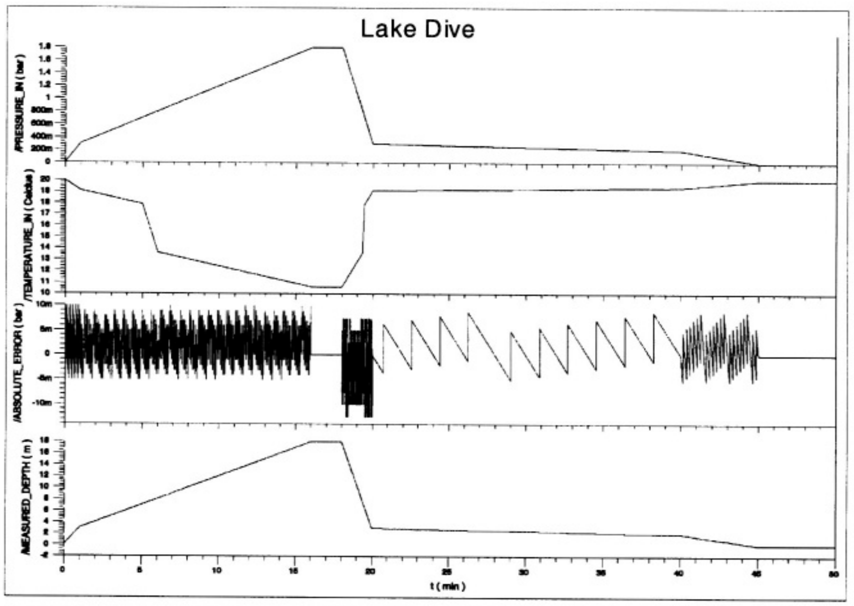
corresponding to the digital depth information. The results of the monitoring are depicted in each third trace as part of the overall simulation results shown in Fig. 3.36.

The first plot in Fig. 3.36 visualizes the results of system simulation for a lake dive. Note the temperature profile of the lake (second trace).

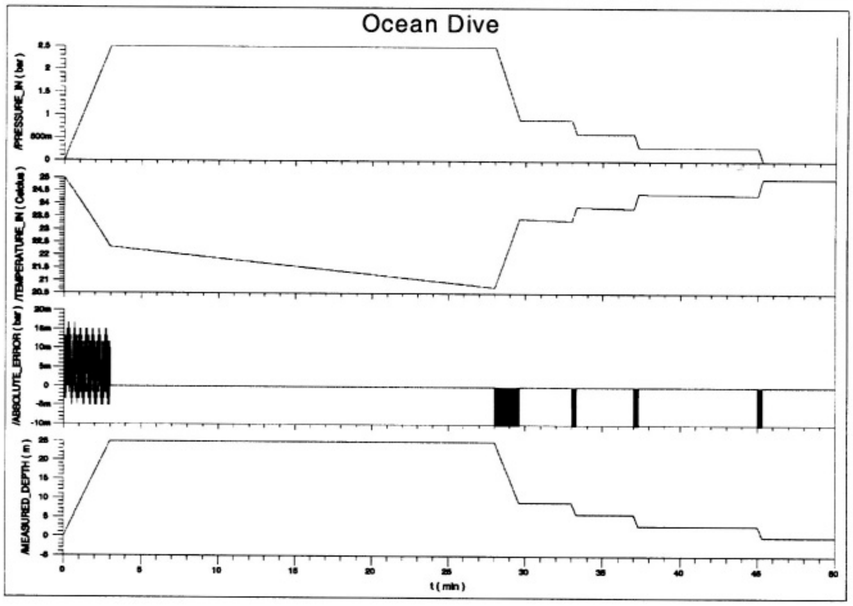
In contrast, the second plot in Fig. 3.36 depicts the simulation results for an ocean dive scenario. The system's model of the gauge is as before, but the environment has changed. Thus, the gauge model acts as a virtual prototype to be exercised in different contexts. The accuracy of the model is visible from the third trace in each plot. The absolute error (i.e., the difference between the actual and the measured depth) is in the range of the least significant bit of the discretized depth value as visible from the third trace in Fig. 3.36 a) and b), respectively.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library disciplines;
use disciplines.electromagnetic_system.all;
use disciplines.Fluidic_system.all;

entity vdc_tester is
  port (signal Reset : in std_logic;
        signal Depth : in std_logic_vector;
        terminal aDepth, aError : electrical;
        terminal act_Pressure : fluidic);
end vdc_tester;

architecture simple of vdc_tester is
  quantity
    act_P_V across
    act_Pressure;
  quantity theDepth across Depth_I through aDepth;
  quantity theError across Error_I through aError;
  quantity myDepth : real;
begin
  if reset = '1' use
    myDepth == 0.0;
    theDepth == 0.0;
    theError == 0.0;
  else
    myDepth == 0.1 * emf(conv_integer(unsigned(Depth)));
    theDepth == myDepth;
    theError == act_P_V - 0.1 * myDepth;
  end use;
end simple;
```

Figure 3.35. Model of aberation monitor vdc_tester



a)



b)

Figure 3.36. Results of system simulation a) Lake dive scenario b) Ocean dive scenario

Chapter 4

CHARACTERIZATION OF CIRCUIT PROPERTIES

4.1 Role and Principles of Circuit Property Extraction

Models of functional blocks or of entire integrated circuits may be derived in different ways, depending on the scope and even on the modeler's view on the design object. Independently of the method of model construction, there is always a need to assign physical values to at least some of the model parameters. The process of determining such values is commonly addressed as *characterization.* There are two distinct application areas present in the domain of electronic circuit design: *device* and *circuit* characterization. The first application aims to parameterize models for active, in general, nonlinear devices such as diodes and transistors. We will not detail on this application because we are mainly interested in model engineering of functional blocks and complex systems. In contrast, circuit characterization addresses the extraction of raw data from either measurements of real circuits or from simulation of models of such circuits in terms of different complexity and calculation of circuit properties from this raw data. In principle, there are three basic methods available for the determination of circuits properties.

1 **Analysis:** Some properties may be calculated directly from more or less simplified mathematical models of functional blocks. A subset of the block behavior has thus to be denoted by means of analytical equations which are largely valid in some subset of the entire operation domain of the block only.

2 **Measurement:** This is the traditional approach to an extraction of circuit properties, which are then arranged into data sheets thus yielding a major part of product documentation. However, a prerequisite to this method is the presence of a real piece of hardware. This method, therefore, is not feasible during the design phase of a functional block or a system. In addition,

some properties cannot easily be determined from measurements only, such as the open loop gain of an operational amplifier, for example.

3 **Simulation:** A description of the functional block to be characterized is represented in an executable hardware description language such as the SPICE netlist format or, more appropriate in these days, VHDL-AMS. The model of the block is then connected to a testbench (i.e., the *measurement setup*) for the properties to be determined as demonstrated in Chapter 3. This augmented model is then subjected to an excitation by input signals and operating conditions and is executed by a simulation program yielding the waveforms of the branch quantities within the block (i.e., voltages and currents) — at least at the input and output ports of the functional block. Then, the properties sought may be calculated from this raw data. Although quite general and of practical impact, this method is not always applicable because of many problems arising especially from solving nonlinear model equations or numerical integration.

Some definitions will be given in the following for the terms introduced above in order to highlight the role of circuit characterization within the design flow of mixed-signal integrated circuits. Without loss of generality we will consider only the simulation-based characterization in the sequel.

DEFINITION 4.1 *A circuit property of a functional block denotes a certain behavior of the block. In general, the behavior depends on a set of independent variables.*

DEFINITION 4.2 *Characterization is the process of determining values of circuit properties from raw data generated from simulation results of appropriate models of the functional block to be analyzed.*

A simple example is introduced to clarify the term *circuit property.* Consider a functional block which implements the Boolean function of inversion (i.e., an inverter) as depicted in Fig. 2.2. An interesting circuit property of an inverter is probably its propagation delay. According to a widely accepted definition — for CMOS circuits at least — the delay value is determined from the time points associated to the 50% values of the voltage at the input and output ports, respectively. Obviously, we have to run the simulation for the structural description of the inverter circuit first, and then we need to determine the mid swing time points from this raw data for both the input and output waveforms. We subsequently execute a difference operation to finally get the time value of the circuit property known as propagation delay. But is this delay value independent of the *operating conditions* of the functional block such as its fanout load or die temperature? Of course not, these independent variables affect the propagation delay of our inverter as illustrated in Fig. 4.1 for some example load values. In addition, the resulting waveforms of the inverter acting as a signal source affect the delay behavior of its sinks, a well-known phenomenon of digital circuits.

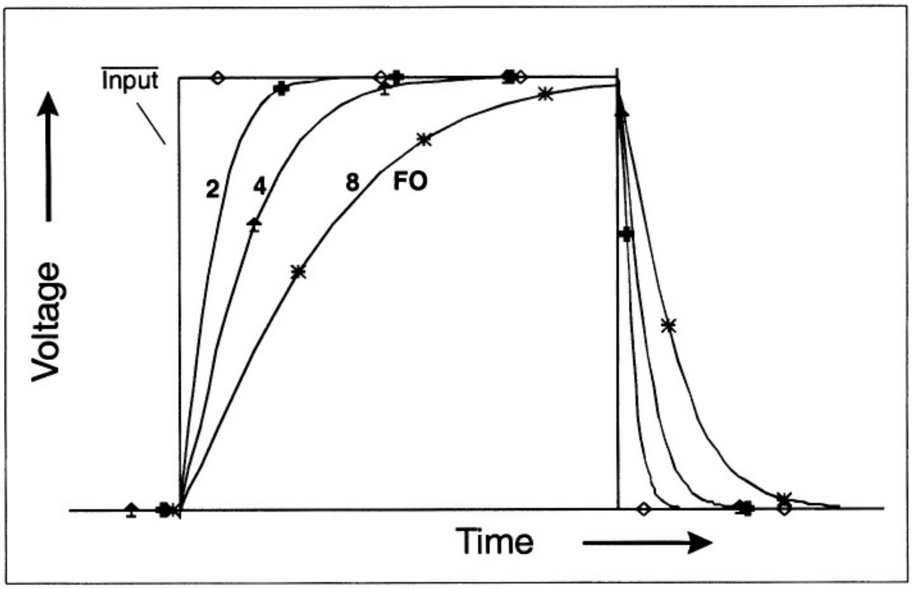


Figure 4.1. Output waveforms of an inverter resulting from fanout load conditions

From this simple example we can derive a more general and formal representation of circuit properties and characterization plans. Denoting a circuit property by f, the set of branch quantities of the structural representation of the functional block by Q_b, and the set of its port branch quantities by Q'_b, whereas $Q'_b \subset Q_b$, then a mapping operation F exists such that

$$F : Q_b \longrightarrow f. \tag{4.1}$$

Note that the mapping of just Q'_b onto f is a special case as demonstrated above for the inverter example. In general, at least some internal branch quantities have to be considered for the mapping as given in Eq. (4.1) because of an internal *state* of the functional block. Finite states are obvious for some digital blocks, but the energy present in analog circuits may be viewed as a generalized state and has to be considered accordingly for property calculations. We will detail on this in the next chapter. Denoting the sets of

Circuit parameters: s

Device model parameters: m

Operating conditions: oc

Statistical parameters: $\boldsymbol{\sigma}$

then we can define the following mapping G for the set Q_b of a functional block

$$G : s \times m \times oc \times \sigma \longrightarrow Q_b(s, m, oc, \sigma). \tag{4.2}$$

Based on Eq. (4.1) and Eq. (4.2) we can now introduce and discuss a new term: the *characterization plan.*

DEFINITION 4.3 *A characterization plan defines the process of extraction of circuit properties and the organization of its results into a data sheet or into a data base.*

The complexity of a characterization plan depends on both the class of the functional block to be characterized and on its domains of operation. For example, digital standard cells operate in time domain and need some simple DC characteristics only. In contrast, analog blocks are to be analyzed in AC, DC, and time domain, whereas the testbenches for the extraction of properties vary considerably compared to digital circuits. Characterization plans, especially for analog and mixed-signal circuits, are highly complex. To allow for a reuse of plans, there is a need to provide a modular way to specify the following tasks.

Definition of simulation runs: The sequence of simulation runs and the analysis domains are to specified, whereas the result of a run is possibly to be used as a parameter for a subsequent simulation. This situation is quite common in analyzing operational amplifiers, for example. The value of the offset voltage needs to be determined from an analysis in DC domain prior to a transient simulation in order to compensate for its effects.

Definition of conditions: Testbenches, operating conditions, sweep intervals for the values of some circuit elements, statistical distributions of model parameters as well as optional runtime values to be passed to the simulation engine — such as accuracy requirements — have to be captured and denoted accordingly.

Definition of algorithms: Many circuit properties (e.g., settling times of dynamic systems) need to be calculated from the raw simulation data by means of more or less complex algorithms. Functions and procedures, therefore, are first to be specified in appropriate formal and executable languages. They are then to be invoked as part of the characterization plan.

Definition of data storage: The extracted values of circuit properties are to be stored and represented in a user definable way in order to produce standard or animated data sheets, for instance.

It is obvious from these first considerations that the elaboration of characterization plans for mixed-signal circuits is a highly complex task. The results obtained from executing characterization plans, however, are central to the design flow of all integrated circuits. Fig. 4.2 depicts the role of characterization with respect to major activities which are part of the design flow in integrated circuit engineering.

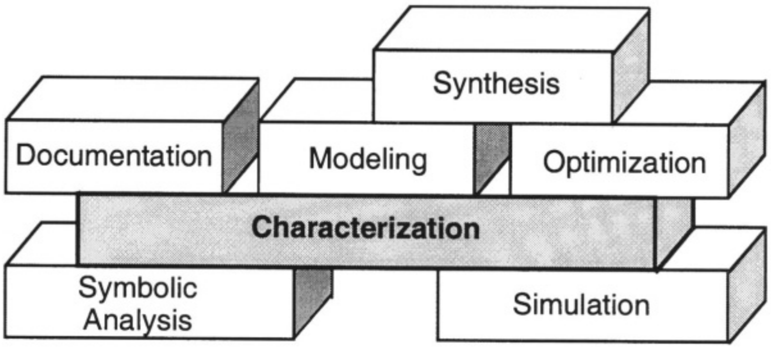


Figure 4.2. Role of funcional block characterization

Characterization produces a comprehensive description of circuit properties in relation to both internal and external elements of a functional block. In addition, it provides a representation of properties as functions of operating conditions. It is thus a prerequisite to both fundamental tasks in circuit design (i.e., *generation* and *validation*) of candidate solutions to an engineering problem. Generation of candidate solutions from synthesis methods (e.g., a specific operational amplifier according to a set of specified characteristics) relies upon parametric models and element sizing procedures which, in turn may be based on general optimization strategies. Its result, however, needs a validation by an intermediate characterization step unless the correctness of the synthesis output is taken for granted because it is supposed to be "correct by construction" — sometimes rather risky in engineering reality.

Validation of models for functional blocks at different abstraction and accuracy levels depends completely on the comparison of output waveforms and circuit properties of the reference device to the model to be analyzed. For meaningful results this comparison requires that the same testbenches and operating conditions are applied to both — model and reference device — regardless of the way how the data is generated, either by measurement or by simulation. Testbenches, operating conditions, data generation methods, and extraction procedures for circuit properties are at best combined into a modular characterization plan for the assessment of model quality.

The *representation of results* may either be in textual form, by using viewers, or directly as data sheets. In general, all these representation forms are required in practice for maximum flexibility. Finally, it makes a significant difference whether the *application domain* is restricted to some circuit class such as digital cells. Mixed-signal circuits as considered here are nonlinear functional blocks operating in DC and in large signal time (i.e., transient) domain. The characterization tool set has to consider this peculiarity accordingly.

From these general objectives we now introduce a set of requirements to be used as a means to assess possible solution methods to the characterization problem:

R1 Intuitive specification of characterization plans

R2 Transparent representation of control and data flow

R3 Simulator independent characterization kernel

R4 Scalable runtime system.

Most of these requirements are addressed in some way by available tools and characterization environments. These tools or tool sets may be classified either into *restricted* or into *general* approaches. Simulator add-ons and functionally enhanced viewers of simulation results belong to the first category. In general, they are by far not fulfilling all the requirements summarized above. The more interesting approaches belonging to the second class are based on either *script languages* or on *procedural simulation* as the fundamental method for performing circuit characterization.

There are plenty of tool suites available which exploit script languages for characterization purposes. Some of them directly use Unix scripts as in autochar [SSS97], or they exploit general purpose script languages such as Perl, or Tcl/Tk just to name a few. These script languages, however, were not intended and designed to support this specific purpose. Therefore, many tool sets define their own script language such as in AIM [Ana97], APLAC[1] [VHK$^+$91], ASIS [SG94], Nutmeg [Nut97], SCL [DJS95], or SimPilot[2] [Ana95]. Especially the script language of SimPilot enhances considerably the capabilities of the circuit simulator Eldo[3] — in context with a comprehensive library of extraction procedures. This simulation engine was initially developed by Anacad GmbH and is now part of the Mentor Graphics Corp. tool suite. As an example for a specific script language, Fig. 4.3 shows a section taken from a characterization plan coded for SimPilot. It is aimed to analyze the temperature dependency of an opamp's slew rate in time domain.

[1] APLAC is a trademark of APLAC Solutions Corp.
[2] SimPilot is a trademark of Mentor Graphics Corp.
[3] Eldo is a trademark of Mentor Graphics Corp.

```
set circuit = "opamp.cir"
#Array cte contains operating temperature values in deg C
cpvector cte[0] -20 0 27 50 100
let elem = length (cte)
let SRUp = vector(elem)
#Run transient analysis to dtermine rising slewrate value
let i = 0
cat $circuit opamp_closedloop.env > tran.cir
tran.cir
while i lt elem
   let @temp[t] = cte[i]
   #Stop simulation when following conditions hold
   stop when sec gt 10u v(out) lt -0.4
   tran 50u 100u
   reset
   let SRUp[i] = 0.4 / risetime(v(out),-0.2,0.2 /1.0e6
   let i = i + 1
end write tran.cou SRUp
#Generate ASCII text file ...
```

Figure 4.3. Part of a characterization plan coded for SimPilot

Using script languages as outlined above often results in somewhat cryptic characterization plans as visible from Fig. 4.3. In addition, parameterizing simulation runs and especially their input data (i.e., testbenches and circuit elements being part of the description of the functional block to be characterized) may give rise to major problems for average circuit design engineers.

Because of the drawbacks stemming from script languages, a different method has been proposed: procedural simulation. One of the first commercially available characterization environments based on this method is known as SimBoy [NHM94], which combines DSS-Spread-Sheets with the versatility of the CAE framework of Mentor Graphics Corp. In contrast, a completely tool and vendor independent approach to procedural simulation is recalled in the following. An application specific programming language named CLANG [HGT91], a runtime system for it, as well as a comprehensive library of procedures for circuit property extraction and for the control of circuit simulator engines form the charcterization environment as summarized in Fig. 4.4.

The structural or behavioral description of the block to be analyzed is strictly separated from both the testbench and the control section of the simulation engine. Thus, block description, testbench, as well as simulation control may be parameterized or even updated according to results gained from previous simulation runs.

Due to the simple syntax of CLANG — a synopsis from Pascal and C programming languages — and the functionality encapsulated into a comprehen-

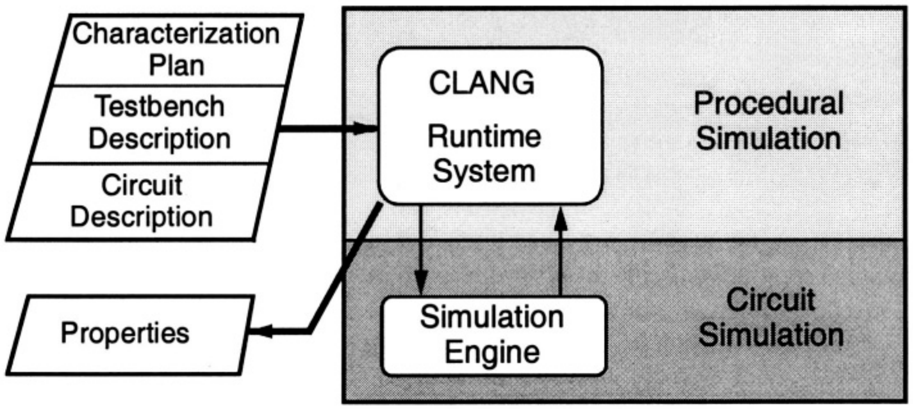


Figure 4.4. Characterization on top of procedural simulation

sive library of functions and procedures coded in C, the task of composing characterization plans for complex applications is now much more straight forward to solve. This becomes obvious when comparing the code of the characterization task denoted in SimPilot as given in Fig. 4.3 to the procedural simulation subset coded in CLANG as outlined in Fig. 4.5.

```
i = 0;
foreach v in Vin do
   foreach t in Temp do
      signal = SigGen ("slewrate.sig", step, v);
  *** Parametrize and start simulation
      Simulate(circuit, "slewrate.tb", signal.t, "", NULL,
                         ".TRAN 10us 1ns", "PRINT TRAN V(OUT)");
  *** Extract raw data from output file
      GetFuncs("", "TAL|T/TIME TAL|T/V(OUT)", time, vout);
  *** Calculate slewrate value
      SlewRate(time, vout, v*0.1, v*0.9, sr);
      i = i + 1;
  *** Store results into arrays
      SRVin[i] = v; SRTemp[i] = t; SRRes[i] = sr;
   end for
endfor
```

Figure 4.5. Extraction of slew rate values coded in CLANG

When coming back to the basic requirements introduced above we now can assess the figures of merit of the two fundamental methods for capturing and representing characterization plans. Table 4.1 summarizes these assessments.

Table 4.1. Coverage of requirements of characterization plans

Requirement	*Script Language*	*Procedural Simulation*
R1	–	∘
R2	∘	+
R3	–	∘
R4	–	–

Although procedural simulation provides a clear improvement to specifying plans in a more intuitive way, it seems not to be sufficient yet. Procedural simulation is definitely not the ultimative means for plan specification, mainly because of a purely textual representation of characterization plans. But, due to its flexibility, it should be exploited adequately. In addition, none of these two basic methods offers an intrinsic support for an efficient runtime execution of characterization tasks.

4.3 Visually specified Characterization Plans

The simulation-based characterization of mixed-signal circuits addresses two layers of knowledge and expertise from a user's point of view. First, paradigms for capturing both control and data flow are required. Secondly, the generation of data by exercising simulation engines and the extraction of circuit properties are the main purpose of characterization, but the way of how this is achieved at the implementation level of the tool is not of primary interest to a user. Therefore, a multi-paradigm approach has to be envisaged:

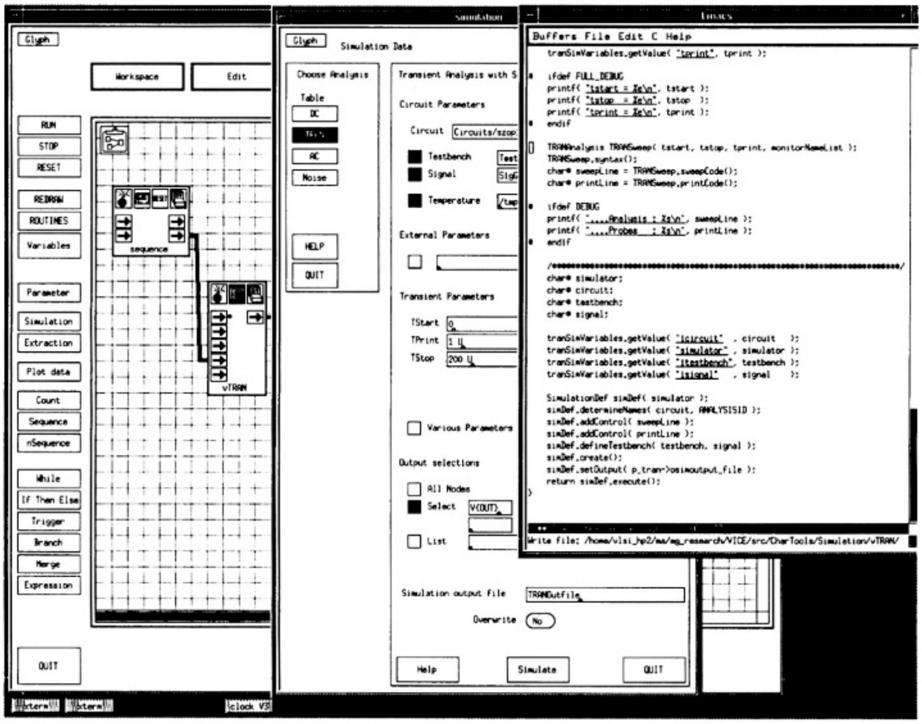
1 A *front end* for an appropriate communication with circuit designers acting as users of the characterization system

2 A *back end* which translates the user-specified characterization plan into a sequence of data generation, extraction, and storage tasks in an intuitive and transparent way.

Thus, a general approach to mixed-signal circuit characterization should consist of

- Visually specified control and data flow with variable granularity
- Textually specified operators.

The Visual Characterization Environment *ViCE* [GH93, GH94, HG95, Goe01] outlined in the following exploits visual programming techniques for the specification of characterization plans (i.e., the front end) and the procedural simulation method for the implementation of the back end. Fig. 4.6 depicts its user interface, which consists of three basic access modes: *visual specification* of data flow, *definition* of dialogues, and *textual specification* of operator functionality by means of C programming language.



Circuit: Circuits/szop
Testbench: Test
Signal: SigG
Temperature: /tmp
TStart: 0
TPrint: 1 U
TStop: 200 U
Various Parameters
All Nodes
Select: V(OUT)
List
Simulation output file: TRANOutfile
Overwrite: No

```
    tranSimVariables.getValue( "tprint", tprint );

#   ifdef FULL_DEBUG
    printf( "tstart = %e\n", tstart );
    printf( "tstop  = %e\n", tstop  );
    printf( "tprint = %e\n", tprint );
#   endif

    TRANAnalysis TRANSweep( tstart, tstop, tprint, monitorNameList );
    TRANSweep.syntax();
    char* sweepLine = TRANSweep.sweepCode();
    char* printLine = TRANSweep.printCode();

#   ifdef DEBUG
    printf( "....Analysis : %s\n", sweepLine );
    printf( "....Probes   : %s\n", printLine );
#   endif

    /****************************************************************************/
    char* simulator;
    char* circuit;
    char* testbench;
    char* signal;

    tranSimVariables.getValue( "icircuit"  , circuit   );
    tranSimVariables.getValue( "simulator" , simulator );
    tranSimVariables.getValue( "itestbench", testbench );
    tranSimVariables.getValue( "isignal"   , signal    );

    SimulationDef simDef( simulator );
    simDef.determineNames( circuit, ANALYSISID );
    simDef.addControl( sweepLine );
    simDef.addControl( printLine );
    simDef.defineTestbench( testbench, signal );
    simDef.create();
    simDef.setOutput( p_tran->osimoutput_file );
    return simDef.execute();
}
```

```
Write file: /home/vlsi_hp2/ma/mg_research/VICE/src/CharTools/Simulation/vTRAN/
```

Figure 4.6. Access windows to the ViCE system

The leftmost window depicts the coarse-grain data flow specification of a characterization plan. The associated control flow is represented by specific *glyphs*, such as the `sequence` block, and by global variables. The window in the center of Fig. 4.6 visualizes the parameterization of `glyphs` by means of dialog forms as frequently needed for the instantiation of simulation services as parts of a characterization plan. Finally, the rightmost window shows the access by the ViCE systems administrator, who implements the `glyph` functionality by means of programs — in C in this case.

ViCE is based on the Cantata visual language in Khoros [RW91], a specification and simulation system for digital signal processing applications. A plan is captured in a straightforward way by a data flow graph-based approach (leftmost window in Fig. 4.6): visually represented vertices are connected by data links. However, application domain related functionality and means of controlling the characterization task are a prerequisite to this approach. This requirement is met by providing suited operator elements visible as vertices in the data flow graph.

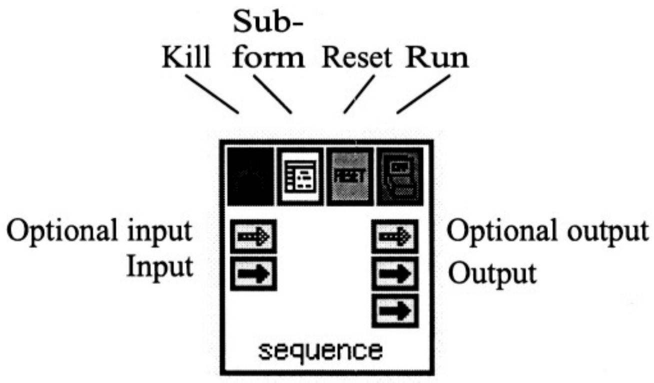


Figure 4.7. Representation of a graph vertex as a glyph

The visual representation of basic operator elements of Cantata denoted as `glyphs` is depicted in Fig. 4.7. The operators may be classified into the following groups.

- **Operator vertices:**
  These elements calculate output values from input data by means of an associated algorithm. They are the main data manipulation operators within the system.

- **Control vertices:**
  Control structures such as loops and conditionals are represented by control operators.

- **Procedure vertices:**
  These elements contain Cantata subgraphs and are thus dedicated to a hierarchical modularization of characterization plans.

The representation of a characterization plan is now much more adequate for circuit designers who are very familiar with visual representations of structural descriptions of functionality such as circuit schematics or block diagrams. In fact, the plan looks much like a block diagram of a transmission system rather than a specification of characterization tasks.

The ViCE system is conceived for a direct support of the following three fundamental *roles* when it comes to circuit characterization.

User: Circuit designers usually take the role of a user of characterization services. They specify characterization plans and testbenches for the circuits to be analyzed.

System administrator: CAD engineers find themselves in the role of setting up a characterization environment, of enhancing functionality upon user request, and of integrating third party software products into the system.

Vendor: Providers of software products such as simulators, library packages, or computer algebra suites deliver the basic building blocks of a characterization system.

In practice, some of these roles may be assigned somewhat differently. In small companies, for example, user and system administrator may be the same person. In contrast, large companies frequently let their in-house CAD groups act as a vendor of product specific design and analysis software packages to the own design departments.

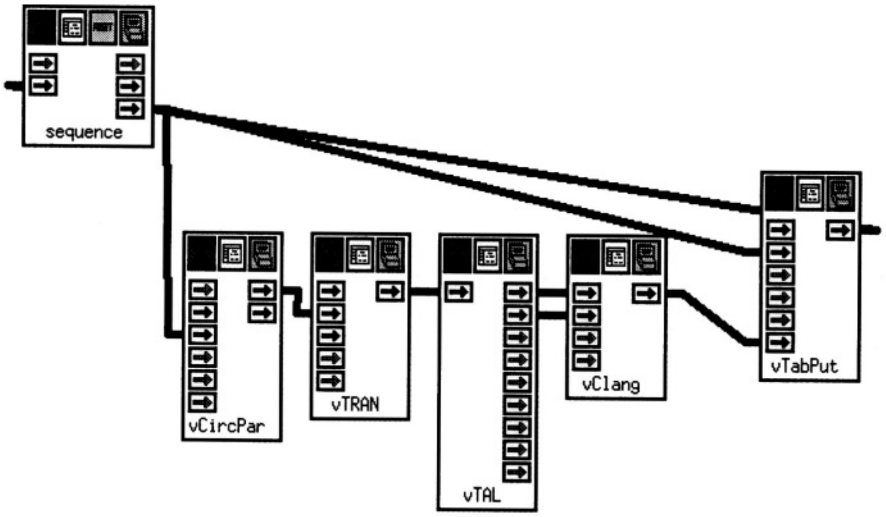


Figure 4.8. Visually represented characterization plan for slew rate extraction

For demonstration purposes, Fig. 4.8 depicts the visual plan of the inner loop for slew rate extraction as in Fig. 4.5. The leftmost vertex generates the sequence of temperature values taken from a predefined set. Then, the vertex `vCircPar` inserts the actual temperature value into the input file of the simulator. Now, `vTRAN` runs the transient simulation. The resulting output file is parsed by `vTAL`, thus extracting the raw transient voltage data waveform to be used as an input to `vClang`. By means of procedures and functions from a program library this vertex finally calculates the slew rate value for each element of the temperature set which, in turn is stored accordingly by `vTabPut`. Note that the actual temperature value is forwarded to this operator by the control vertex denoted in Fig. 4.8 as `sequence`, which implicitly provides the loop construct for stepping through the temperature range. Compared to the representation given in Fig. 4.5, this characterization plan plainly depicts what is to be done during characterization. Visually represented plans are, therefore, best suited for the specification of complex characterization tasks. The visual language Cantata is more appropriate for this kind of application than other existing visual languages or systems such as LabView[4] or VEE[5], for instance. This is because of its data generation mechanism which is aimed more to program execution rather than to event and message processing during real-time measurements.

Cantata provides a variety of control operators as visually represented vertices. However, some characterization-specific extensions and enhancements to Cantata had to be considered. Especially global variables as used originally in Cantata for communication purposes between control vertices had to be replaced because they did not follow the data flow principle. Some additional control operators were introduced such as `cond` and `do`, which implement the `if` conditional and the `while` loop constructs without referring to global variables as in the original version of Cantata.

Table 4.2 summarizes the complete set of control operators available in the ViCE system. Details on the various enhancements to the ViCE version of Cantata can be found in [Goe01].

The control flow within a characterization plan is thus embedded into specific operators resulting in a data flow-based overall computational model of the plan. The correctness of the underlying data flow graph in terms of safeness and liveliness is validated by animating the plan (i.e., by interactive execution).

Table 4.2. Available control operators

Cantata	*ViCE*
`for`, `if`, `merge`,	`cond`, `do`,
`trigger`, `switch`, `while`	`sequence`, `nsequence`

[4]LabView is a trademark of National Instruments, Inc.
[5]VEE is a trademark of Agilent Technologies, Inc.

4.4 Architecture of the ViCE System

The foundation of the ViCE system is the *client-server* architecture. The characterization plan defined visually by a user acts as a client and requests different services offered by specific servers of the system. Therefore, servers are provided for the services of simulation, data extraction, data analysis, data storage, documentation, and result visualization. Each of these services is embodied by one or more servers. A server is implemented as an operator vertex to be instantiated by the visual plan entry system.

As an example, consider the operator vertex denoted `vTRAN` in Fig. 4.8. This operator provides a simulation service and thus represents one of the simulation servers for transient analysis, which invokes a properly interfaced SPICE3 simulation engine in a completely transparent way to a user. In addition, this approach supports an easy integration of a variety of external tools and characterization plans coded in different script or programming languages. For example, the Mathematica package provided by a vendor as well as third party procedures or complete libraries coded in CLANG and/or in SimPilot may be introduced to the ViCE system as servers. Once the interfacing, the execution scripts of the tool, and the visual representation of the operator vertex are defined by the system administrator as outlined in Fig. 4.6, then the new server is just another vertex in the underlying data flow graph which, in turn represents the complete characterization plan. Fig. 4.9 depicts an overview of the ViCE system.

As mentioned in the previous section, the basic concepts of the characterization system ViCE are visually specified control and data flow as well as textually specified operators. Taking the coarse granularity of operators and the message passing paradigm into consideration for the communication between the operators, it seems quite straight forward then to attempt an automatic parallelization of characterization plans. An inherent parallelism is almost always there when characterizing circuit blocks over operating condition ranges or intervals of element values. This common situation is depicted in Fig. 4.10 for the parallelization of an intrinsic loop within a visually represented data flow graph. Here, two operator vertices are merged (i.e., *clustered*) into one vertex in order to reduce communication overhead.

Following another objective of the ViCE project (i.e., to exploit as many available methods and subsystems as possible) results in an integration of the well-known PVM[6] program suite [SJN94], thus yielding both a reliable and a general runtime system for circuit characterization [Ant99]. Fig. 4.11 highlights the phases of parallelization on top of PVM. First, the data flow graph extracted from the visually specified plan in the ViCE version of Cantata is

[6]PVM is public domain software initially developed at Oak Ridge National Labs.

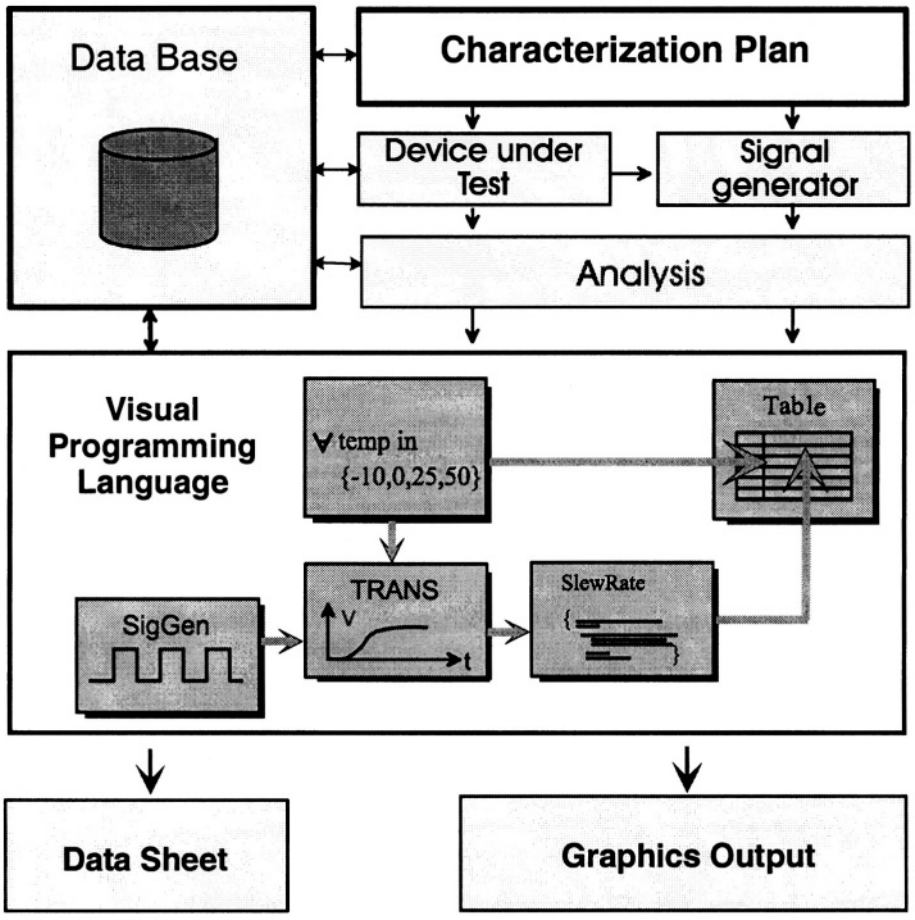


Figure 4.9. Overview of ViCE

mapped to an intermediate, object-oriented graph representation [Bau94]. Secondly, an optional clustering of the intermediate graph and a generation of C code takes place. This code is then translated on top of possibly different hardware platforms for an execution (e.g., HP-UX[7], Linux[8], and Solaris[9] workstations). Thirdly, scheduling of the executable code is performed by means of PVM. Servers are being activated upon request during the runtime of the programs generated from the initial data flow graph. The communication of

[7]HP-UX is a trademark of Hewlett Packard Corp.
[8]Linux is public domain software from Linus Torvalds
[9]Solaris is a trademark of Sun Microsystems, Inc

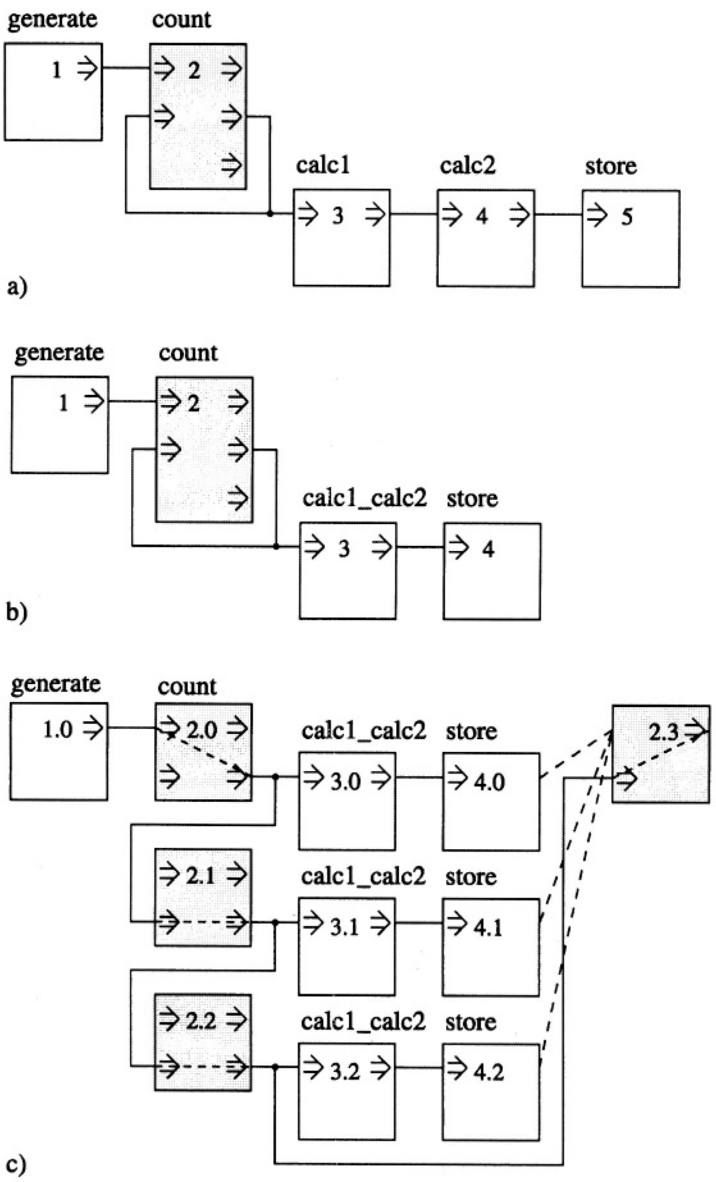
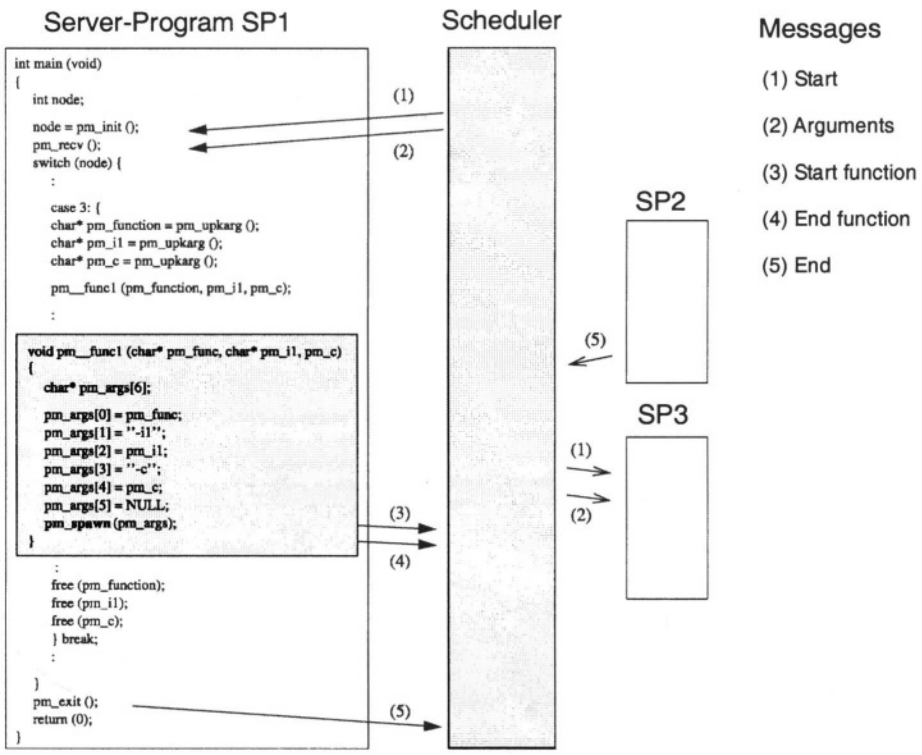


Figure 4.10. Loop parallelization a) Data flow graph b) Clustering c) Data flow graph with node replication

the PVM scheduler with the server programs denoted as SP1, SP2, and SP3 is detailed in Fig. 4.11. The mapping of the graph is summarized in Fig. 4.12 for the resulting system *xpViCE.*



```
int main (void)
{
   int node;

   node = pm_init ();
   pm_recv ();
   switch (node) {
      :

      case 3: {
      char* pm_function = pm_upkarg ();
      char* pm_i1 = pm_upkarg ();
      char* pm_c = pm_upkarg ();

      pm__func1 (pm_function, pm_i1, pm_c);

      :

void pm__func1 (char* pm_func, char* pm_i1, pm_c)
{
   char* pm_args[6];

   pm_args[0] = pm_func;
   pm_args[1] = "-i1";
   pm_args[2] = pm_i1;
   pm_args[3] = "-c";
   pm_args[4] = pm_c;
   pm_args[5] = NULL;
   pm_spawn (pm_args);
}
      :
      free (pm_function);
      free (pm_i1);
      free (pm_c);
      } break;
      :

   }
   pm_exit ();
   return (0);
}
```

Figure 4.11. Generic communication of scheduler and server programs within xpViCE

One has to take care in an adequate configuration and runtime control support when exploiting a parallel computer as the workhorse for simulation-based circuit and functional block characterization. This is especially true in the presence of a virtual and heterogenous parallel machine. Configuration and monitoring utilities for the workstation network used as the parallel machine, therefore, are part of the experimental, parallel operating characterization system xpViCE [BHW99].

The graphical user interface for runtime control is detailed in Fig. 4.13. Configuration and control services are accessible via the entry console (1), which is also visible from this figure. Available hosts and restrictions on the amount of allowed processes on each host may be defined (2) as well as details on the maximum number (3) of parallel program runs feasible for licensed products. Finally, by invoking the scheduler, (4) the parallel virtual machine may be started.

The outlined characterization environment xpViCE has demonstrated to be flexible and powerful enough for fulfilling the requirements of complex mixed-signal applications.

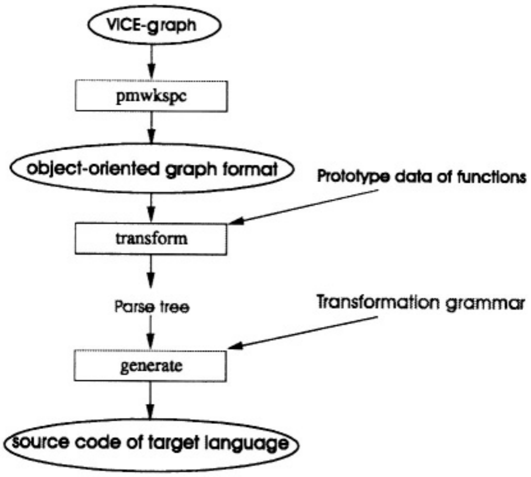
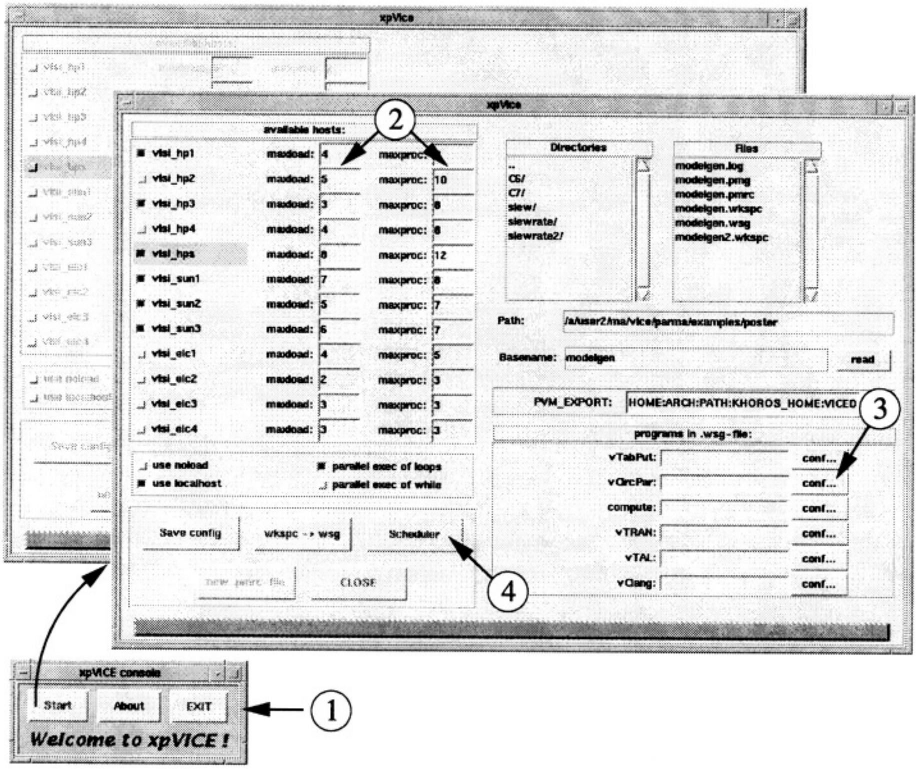

VICE-graph
pmwkspc
object-oriented graph format
Prototype data of functions
transform
Parse tree
Transformation grammar
generate
source code of target language

Figure 4.12. Outline of graph mapping



available hosts:

■ visi_hp1	maxload: 4	maxproc:
❑ visi_hp2	maxload: 5	maxproc: 10
■ visi_hp3	maxload: 3	maxproc: 8
❑ visi_hp4	maxload: 4	maxproc: 8
■ visi_hps	maxload: 8	maxproc: 12
■ visi_sun1	maxload: 7	maxproc: 8
■ visi_sun2	maxload: 5	maxproc: 7
■ visi_sun3	maxload: 6	maxproc: 7
❑ visi_eic1	maxload: 4	maxproc: 5
❑ visi_eic2	maxload: 2	maxproc: 3
❑ visi_eic3	maxload: 3	maxproc: 3
❑ visi_eic4	maxload: 3	maxproc: 3

❑ use noload
■ use localhost
■ parallel exec of loops
❑ parallel exec of while

Directories: .. C6/ C7/ slew/ slewrate/ slewrate2/

Files: modelgen.log modelgen.pmg modelgen.pmrc modelgen.wkspc modelgen.wsg modelgen2.wkspc

Path: /a/user2/ma/vice/parma/examples/poster

Basename: modelgen

PVM_EXPORT: HOME:ARCH:PATH:KHOROS_HOME:VICED

programs in .wsg-file:
vTabPut:
vCircPar:
compute:
vTRAN:
vTAL:
vClang:

Figure 4.13. User interface of xpViCE

Although data extraction by measurements or by simulation share many requirements of processing and compression of data, there are significant differences in controlling the generation process of raw data. In case of measurements there is a need control real measurement equipment and to collect their data within tight time frames resulting in stringent requirements of handling interrupts efficiently. In contrast, data generation from simulation relies upon mixed-signal models of basic elements and functional blocks, on algorithms for event processing and numerical integration, and especially on controlling the software which executes models by running algorithms for the purpose of raw data generation: an event-discrete/time-continuous simulator.

4.2 Requirements of Simulation-based Characterization

Because of its importance, circuit characterization has been addressed by many authors in the past (e.g., [Cir91, LSJ96, LMN95, MK90, NHM94] and [Ant98]). There are several CAE tools available for this purpose, either from academia or as commercial products, but many of them are aimed to produce the documentation of digital cell libraries only. In the following we will focus on the assessment of well-known basic methods in analog and mixed-signal circuit characterization because analyzing purely digital functional blocks requires the execution of just a subset of the characterization tasks needed for mixed-signal circuits. We then will introduce a set of requirements and — based on these criteria — discuss some of the available approaches to circuit characterization in more detail. The main objectives of mixed-signal circuit characterization, therefore, are

- Execution efficiency
- Runtime control
- Result representation
- Application domain.

Execution efficiency is related to how the simulation runs are executed for a given characterization plan (i.e., whether the simulations have to be run sequentially or may be executed in parallel in case of an intrinsic parallelism within the plan). From the user's point of view it is essential for the latter case that the runtime environment provides an automatic scheduling of simulation and extraction runs without major user interaction. *Runtime control,* again from the user's point of view, should be as convenient as possible. It is of utmost importance how the control information is captured: table-driven, by text files, or by means of visual techniques.

Chapter 5

ADVANCED MODELING METHODOLOGY

5.1 Motivation

The engineering process of mixed-signal integrated circuits and systems is rather complex and requires means to assess and evaluate concepts and properties of the design object well in advance to a first implementation in addition to skilled and experienced designers. Simulation is the premier method for coping with early evaluation and assessment tasks, but its prerequisite is the availability of appropriate models for the design object to be engineered. Model engineering, therefore, is a fundamental activity throughout the whole design process of integrated circuits.

Behavioral models as emphasized in this book are characterized by the mapping of input to output signals described completely by mathematical equations and event processing algorithms as discussed in Chapter 2. In addition, interface signals with a direct physical interpretation are subjected to conservation laws (i.e., Kirchhoff's Rules), which are to be met for currents and voltages in mixed-signal integrated circuits. These characteristics of behavioral high-level models are highlighted for a generic analog-to-digital converter as depicted in Fig. 5.1.

The converter takes a voltage value as delivered by a preceding sample-and-hold block at the input ports and converts it by a successive approximation method thus producing digital output data, which is the coded representation of the input voltage value. A behavioral model of this converter is outlined in the VHDL-AMS code of Fig. 5.2.

This representation meets all the characteristics given above: The input signal is defined as an `across` branch quantity derived at `port terminals` which, in turn are assigned to the `nature` Electrical (i.e, voltage), and the output signal is a parameterized array of digital bits. In addition, the mapping

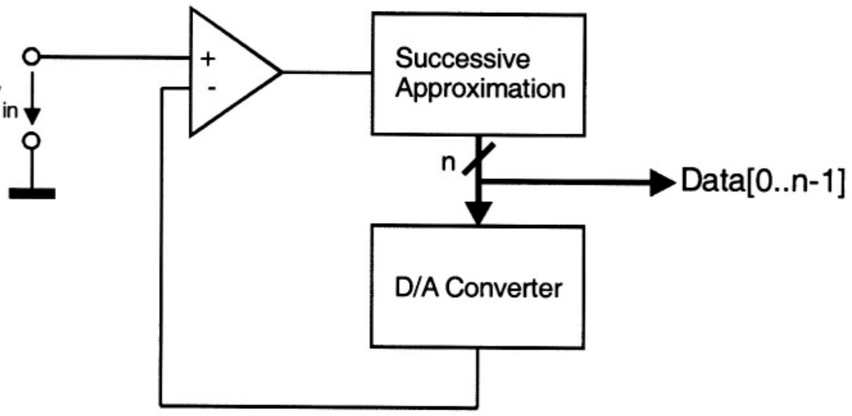


Figure 5.1. Block schematic of a successive approximation A/D converter

from input voltage to output data is denoted by an algorithm, which consists of both analytical equations and event processing. So, this seems to be a perfect behavioral model of an A/D converter. Because the representation of Fig. 5.2 is parameterizable, one may easily define a nicely operating — in simulation, though — instance of this model for a 24 bit ADC working at 2 GS/sec conversion rate simply by setting the generic resolution parameter `Res` to 24 and by defining the clock signal frequency outside the model accordingly. However, from practical experience it is well-known to a skilled design engineer that implementing in reality such a converter is more than hard. What went wrong?

When starting from a rather abstract view on the functionality of information processing blocks one should carefully consider the way how behavioral models are denoted. In our example, the modeler was biased to a 'top-down' view resulting in an algorithmic description for synchronously operating architectures (i.e., more to the view of a software programmer than of a mixed-signal circuit design engineer).

5.2 Classification of Modeling Approaches

Over the past decades plenty of approaches and detailed methods have been proposed as possible solutions for the complex task of modeling the behavior of analog and later on of mixed-signal circuits in an abstract way. Starting from macro modeling as pioneered by Boyle et al. [BCP74], the actual model bandwidth as documented in literature is impressive: From transistor netlists enhanced by analytical equations and coded in VDHL-AMS [Cam98] to functional, algorithmically denoted and essentially digital models wrapped into some behavioral shell as in Fig. 5.2, from activation semantics of abstract

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library disciplines;
use disciplines.electromagnetic_system.all;
entity ADC is
  generic (
    MinVoltage : emf := 0.0;
    MaxVoltage : emf := 1.0;
    Res        : integer := 8);
  port (terminal U_in_P : Electrical;
        terminal U_in_N : Electrical;
        signal Clock, Reset : in std_logic;
        signal Data : out std_logic_vector((Res-1) downto 0));
end ADC;

architecture beh_cascaded of ADC is
  quantity U_In_V across U_In_P to U_In_N;
begin
SeqConvAlg : process (Clock,Reset)
  variable i : integer;
  variable r, temp : real;
  begin  -- process
    if Reset = '1' then
      Data <= (others => '0');
    elsif Clock'event and Clock = '1' then
      -- normalisation: input mapping to [0 ... 1]
      r := (U_in_V - MinVoltage ) / (MaxVoltage - MinVoltage);
      -- A/D conversion
      for i in (Res-1) downto 0 loop
        temp := (2.0**(i-Data'length));
        if r > temp then
          r := r - temp;
          Data(i) <= '1';
        else
          Data(i) <= '0';
        end if;
      end loop;
    end if;
  end process SeqConvAlg;
end beh_cascaded;
```

Figure 5.2. Behavioral generic VHDL-AMS model of an A/D converter

functionality in time domain combined with structural information [GW98] to sophisticated, causal models [Jes01], from a mix of functional, behavioral, and

Table 5.1. Classes of modeling approaches

Concept	*Objective*	*Style*
causal	block property mock-up	mixed-mode algorithmic description
acausal	nonnumerical evaluation	mixed-level structural description
	signal waveform calculation	

architectural representations [EZJ+00] to complex compound DAE and event processing models [OHLR00].

Taking into consideration the abstraction hierarchy levels as summarized in Chapter 2 and the expressivity of modern HDL, which support quite different ways to represent one and the same behavior as discussed in Chapter 3, it seems that every possible variant of a model may be produced in one way or the other. Therefore, an attempt to classify available modeling approaches for both analog and mixed-signal circuits is presented in the following.

Models are structured into three main classes, which are related to their basic rationale. These classes are denoted as *concept, objective,* and *style,* respectively. The content of each of these classes is summarized in Table 5.1. The fundamental modeling concepts known as causal and acausal have already been introduced and discussed in Chapter 2.

Modeling objectives may be structured into three groups: *block property mock-up, nonnumerical evaluation,* and *signal waveform calculation.* Property mock-up is most popular when it comes to abstract behavioral modeling of analog and, to a certain extent, of mixed-signal circuits. The underlying idea is to identify key properties of a block, such as loop gain, transit frequency, S/N ratio, slew rate, settling time, linearity, conversion time, etc., and to reflect these properties directly in the model as it has been done for long years with macro modeling. Values of identified key properties are first extracted from a characterization process applied to a low-level description of the block. They are then represented either as analytical functions (possibly defined in disjoint operating domains of the block), which are parameterized from a fitting process exploiting regression models, or directly as tables in a higher-level HDL description of the block. Digital signal processing may be added to the model by instantiating the code of associated algorithms. The following list referring to methods of implementing this objective is by far not complete [VVGS99, FVG00, WVD+99, VVV+99, DG98, LDGS97, VDG00, EZJ+00, DP98].

Many commercial vendors of model libraries favor property mock-up because it yields high quality behavioral models even when compared to their real counterparts by means of measurements [Ant98] — but only if such empirical models were generated by experts and if the models were to be applied in a context exactly as specified by the model producer. However, two major drawbacks are to be mentioned. First, a model reflects only properties and property interactions anticipated by a human modeler. In other words: Properties or relations of a property to other ones not explicitly coded into the model are simply not there. Secondly, only ”if one fully understands the circuit, one is thus able to replace the complete circuit by a behavioral equivalent” [LDGS97] model generated in this way. In case that model producer and model user are disjoint persons, then problems may arise concerning the accuracy and the application ranges of such a model.

Nonnumerical evaluation, also known as *symbolic analysis,* an area pioneered by Gielen, Sansen, et al. [GS91, WGS98, DLG+98, GWS94], is the second basic objective. Nonnumerical evaluation is aimed to an analytic representation and symbolic manipulation of equation systems which denote the behavior of analog circuits featuring a weak nonlinearity. In addition, the circuits subjected to symbolic analysis are generally operated under small signal conditions. An application of symbolic analysis to modeling of mixed-signal circuits and systems is not straight forward.

The third objective is signal waveform calculation. By exploitation of numerical DAE solvers as well as event processing engines, a user of mixed-signal models may deduce all signal waveforms of interest from their execution — both internal to a block and at its interface. The accuracy of the resulting waveforms, however, depends primarily on the degree of detail present in the description chosen for the block behavior, but not on solvers or engines. This is usually because no explicit selection from a set of available solvers takes place prior to a simulation run of models being specified at different detail levels. Abstraction levels as summarized in Chapter 2 have thus to be considered carefully when denoting the block functionality or behavior. Signal waveform calculation produces explicitly all data required for determining electrical and abstract properties as well as their interaction within a block for model representations stated on any of the mentioned abstraction levels. The high computational effort required for signal waveform calculation especially for component level representations is the main drawback to be noticed. This major flaw was basically the reason for introducing macro models in the past which, however, are constructed according to the objective of property mock-up.

Modeling styles may be distinguished by the point of view a modeler takes. The first style is denoted as *mixed-mode algorithmic description.* The time-continuous and the event-discrete partitions of a model, respectively, are separately coded. They interact in an explicitly defined way resulting in well-structured algorithmic representations of mixed-signal behavior as mentioned in [OHLR00]. The second style is denoted as *mixed-level structural description.* This style is related to a successive decomposition of design objects into a structure of interacting modules, which may be represented by models denoted at different abstraction levels.

The methodology presented in the remainder of this chapter addresses model generation and representation at functional analytical and at behavioral algorithmic level according to Fig. 2.9, thus covering both the causal and the acausal modeling concepts. Its objective is signal waveform calculation, and the advocated model representation style is mixed-level structural description. A high accuracy of resulting signal waveforms is the main asset of this methodology. A considerable computational effort is indispensable for accurate waveforms, it cannot simply be abstracted away. This effort, however, may be shifted from the model execution to the model generation phase as illustrated in Fig. 5.3 and detailed in the sequel. The interaction of time-continuous and event-discrete signals within a mixed-signal circuit model is based on the systems theoretical DEV&DESS conceptual model, which is discussed in the next section.

Generality is not claimed for this methodology because it does not address each and every class of circuits in all operation domains. For instance, oscillator circuits or small signal modeling in frequency domain are not covered. However, the advocated methodology efficiently supports model generation for analog and mixed-signal circuits as found in many relevant engineering applications. In addition, the modeling process is transparent, scalable, and under full user control.

5.3 The DEV&DESS Model

Functional descriptions of time continuous (i.e., analog circuits) may be derived using DESS as a basis. Event or time discrete (i.e., digital systems behavior) may exploit DEVS or DTSS, respectively, as a theoretical foundation as discused in Chapter 1. Heterogenous behavior, a characteristic of mixed-signal circuits, needs an appropriate theoretical model for denoting the requirements of both analog and digital signal processing. Consequently, a combined model has been proposed for that purpose in systems theory [Pra91].

DEV&DESS, the combined differential equation and discrete event specified dynamic system model, is defined by

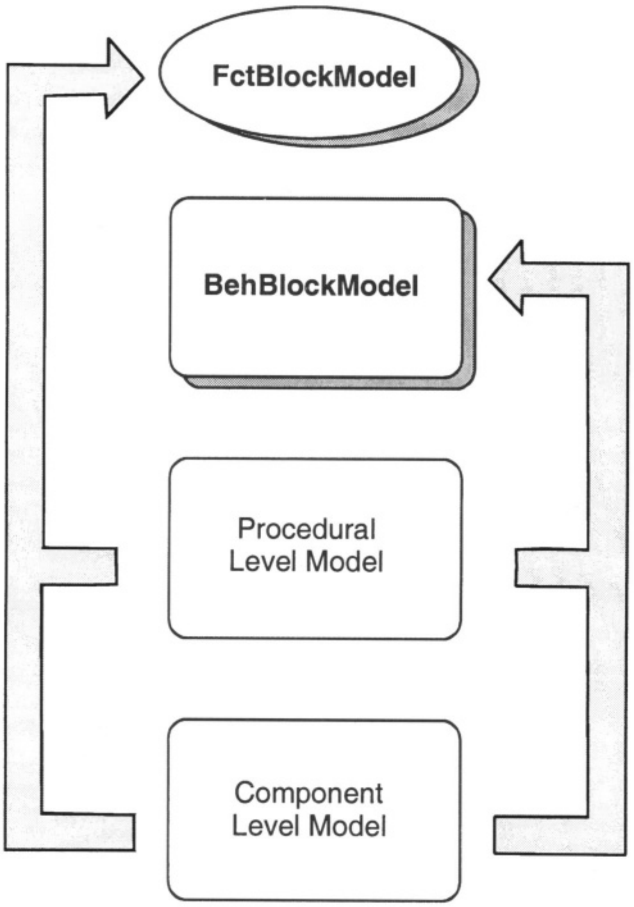


Figure 5.3. Information flows for model generation

$$DEV\&DESS = \{t, \vec{u}, \vec{y}, \vec{x}, \delta_{ext}, \delta_{int}, \lambda, ta, f\} \tag{5.1}$$

$\vec{u}$	Vector of input signals
$\vec{y}$	Vector of output signals
$\vec{x}$	Vector of state variables
δ_{ext}	Transition function for external and state events
δ_{int}	Transition function for internal events
λ	Output function
ta	Time advance function
f	Rate of change function

This intrinsically causal model class defines the cooperation of DEVS and DESS submodels, Fig. 5.4 illustrates the underlying formalism. The input $\vec{u}$ as well as the output $\vec{y}$ consist of both event and piecewise continuous segments. The DESS rate of change and the output functions, respectively, specify the continuous behavior of the composed model as long as no events originating from DEVS interfere.

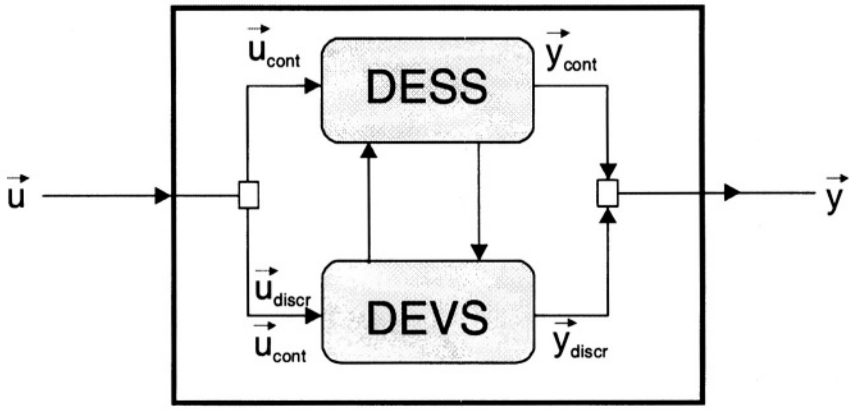


Figure 5.4. DEV&DESS model architecture

The DEVS part in Fig. 5.4 is affected by events produced by its DESS companion in addition to input signal changes. This kind of evens denoted as *state event* is generated when some DESS internal continuous state variable crosses predefined thresholds. Conditions specified on such quantities then become true, thus causing an event which is forwarded to DEVS for further processing. A short and informal description of the DEV&DESS operation reads as follows, formal and comprehensive representations are given in [Pra91, ZPK00].

In case that an internal event occurs first within a time interval, then the internal transition function of DEVS executes and establishes a new state s which is possibly reached $ta(s)$ time units later. The time advance function ta thus reflects a delay operation. The associated output function of DEVS is invoked for the calculation of the associated output event. At a certain point within a time interval an external or a state event may occur. Now, the external transition function is executed for establishing the next state, again possibly after a certain time delay. Continuous state waveforms are computed by DESS always from its actual input and state values at the beginning of each time interval plus the integral of the rate of change function along the time interval until the time point of an event. The continuous output waveforms are calculated accordingly.

5.4 Basic Methodology and Model Architecture

The DEV&DESS model is an appropriate theoretical foundation for functional and behavioral models as detailed in the following because it considers both event processing and the time derivatives of state variables within one entity at the same time, thus fulfilling the requirements of mixed-signal circuit modeling. However, the model definition of Eq. (5.1) does not denote explicitly algorithms for the calculation of the detailed output waveforms of the block in time domain resulting from a close interaction of events and time-continous signals. Again, the modeler is urged to return to structural and behavioral details of the block functionality of the circuit to be modeled, has to 'somehow' construct the model, and then map it to an executable model description which shows in its coded version more or less clearly the basic components of the DEV&DESS architecture. A closer look on how the output waveforms are determined from simulating executable model descriptions in time domain helps to establish a feasible and at the same time transparent model generation approach. For the sake of clarity we will consider only blocks featuring single inputs and outputs in the sequel.

The time-dependent output waveform of a dynamic system is generally determined by an integration operation which, in turn is implemented as a numerical algorithm. Thus, the values of the output variable are calculated at discrete time points only, which result directly from the time steps of the integration process. These time steps may or may not be equidistant, it depends on details of the time step calculation methods as implemented in the integration algorithm. A generic result from such an integration algorithm is depicted in Fig. 5.5 for the output signal y. A first consequence from this calculation at discrete points in time is that each new value is determined by incrementing the previous one by a certain amount. As a second consequence, the output values at time points in between integration time steps are not known. They only may be estimated, for example, by linear interpolation as depicted in Fig. 5.5.

When changing our way of looking at these results, then we may state that an event-discrete processing takes place: At points in time not known a priori — they result from the integration algorithm — events are produced internally to the model, which are processed in order to calculate the output value as visualized in Fig. 5.5. Up to now, we have considered internal events only.

Next, let us consider the response of a dynamic system to the change of a input value (i.e., its reaction to an external change of variable values). Fig. 5.6 shows a typical response of such a system to an input step function. First, the output variable reacts delayed to the change of the input value. Secondly, the rate of change of the output (i.e., its time derivative) is bounded resulting in a slewing characteristic as visible from Fig. 5.6. Thirdly, the value of the rate of change function is not a constant. This is especially true for nonlinear dynamic

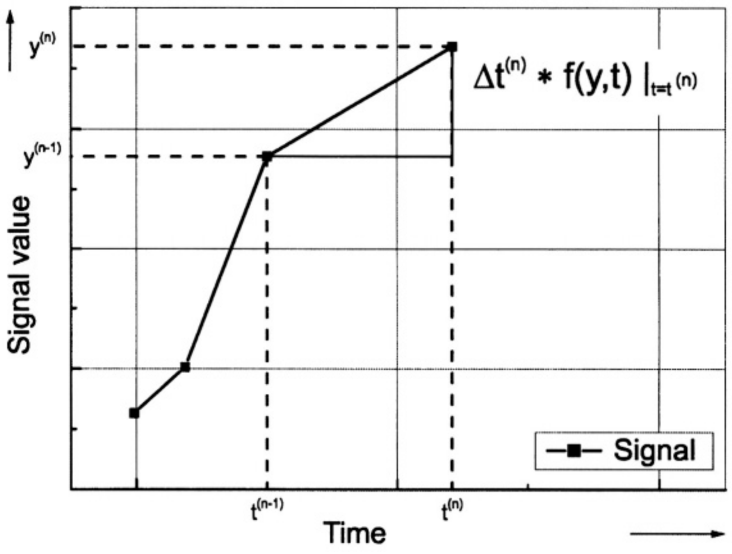


Figure 5.5. Output signal values at discrete points in time

systems, where the slewrate is a strongly nonlinear function in terms of input variables.

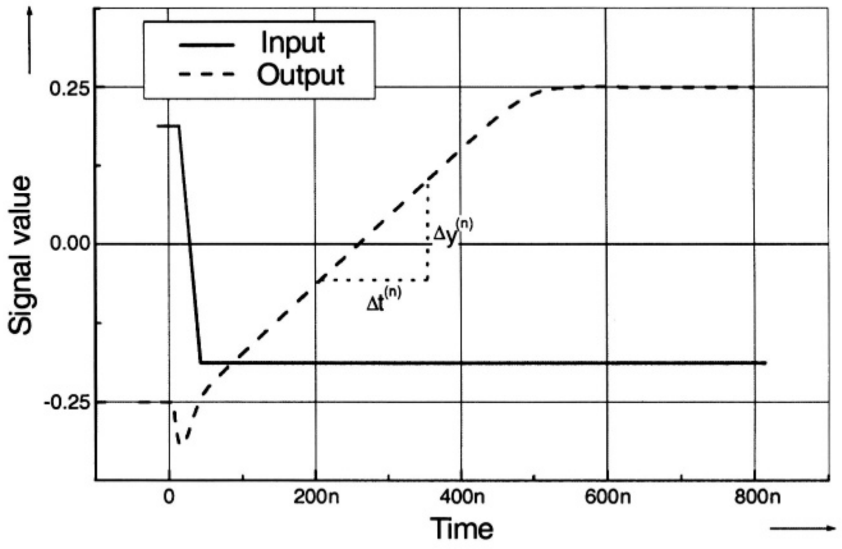


Figure 5.6. Response of a dynamic system to an input variable change

Value changes of a time-continuous input function do not immediately establish the systems response. They are evaluated at the time points of internal events and affect the output possibly at a later point in time and for a much longer time interval than the next integration time step takes. So, changes taking place to the model externally are handled by internal events under consideration of the internal state the system is in (i.e., the past of the system is also taken into account for the calculation of the resulting output waveform). This situation is summarized in Fig. 5.7. Starting at an internal event (i.e., an integration time point) an external input is first sampled and then the new values for state and output variables are scheduled for calculation by means of additional internal events.

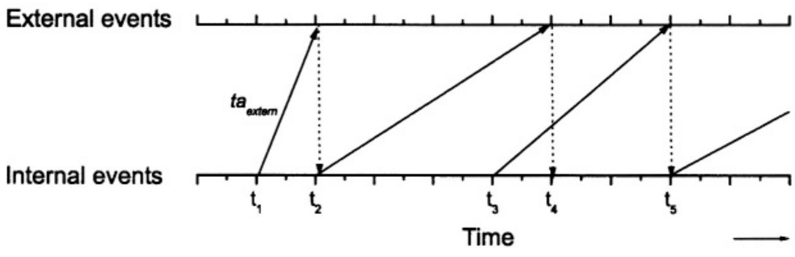


Figure 5.7. External and internal event queues

What happens in case of an external event (i.e., a value change of a digital input signal) to the model of a mixed-signal circuit? Consider the generic model code for a 1 bit D/A converter given in Fig. 5.8 to highlight its effect. Remember that in VHDL-AMS all statements placed in the executable section of an `architecture` operate concurrently.

An event set to S (i.e., the digital input to the converter) forces `ConvDly` time units to a recalculation of and a value assignment to the time continuous-branch quantities denoted as `Vout` and `Iout` after said event. Consequently, an event is scheduled accordingly by adding it to the event queue of the simulator, which contains all known future events. Thus, an interrupt of the ongoing calculation of time-continuous waveforms takes place at due time caused by the `break` statement in Fig. 5.8. This situation is depicted in Fig. 5.9, which may be viewed as another representation of the simulation cycle in mixed-signal hardware description languages.

The architecture of the functional block model detailed in the following is a refinement of the formal DEV&DESS model according to the observable behavior of mixed-signal circuits as summarized above. It is based on work published in [RH98, Ros01].

```
entity DAC is
  generic ( Vhigh: real := 5.0;
            Vlow: real := 0.0;
            ConvDly: time := 10 ns);
  port( signal S: in bit;
        terminal AnPin: Electrical)
end entity DAC;

architecture Gen of DAC is
  signal Sint: bit; -- internal copy of input signal S
  -- output branch quantities with implicit reference
  -- to Ground
  quantity Vout across Iout through AnPin;
begin
  Sint <= S'delayed(ConvDly); -- assign the DAC
  -- conversion delay by shifting the event time
  if Sint = '0' use
  -- calculate branch quantity values
    Vout == Vlow;
  else
    Vout == Vhigh;
  end use;
  break when Sint'event; -- an external event on S
  -- forces the recalculation of branch quantity
  -- values after ConvDly time units
end architecture Gen;
```

Figure 5.8. Generic VHDL-AMS model code for a 1 bit D/A converter

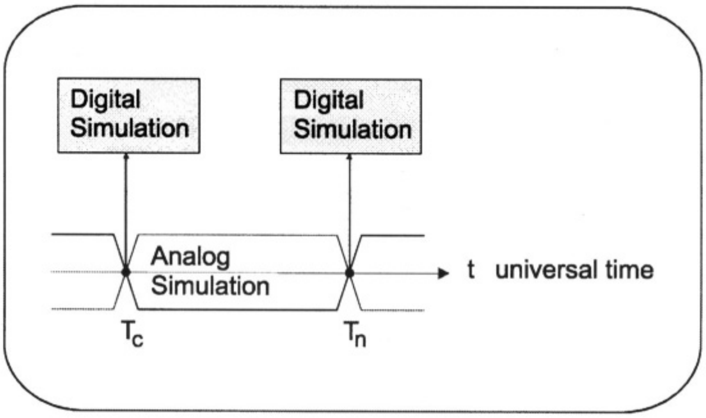


Figure 5.9. Simulation of mixed-signal circuits

$$BlockModel = \{\vec{u},\vec{x},\vec{y},\delta,\lambda,ta,f\} \tag{5.2}$$

$\vec{u}$	Vector of input variables
$\vec{y}$	Vector of output variables
$\vec{x}$	State variables
δ	Transition function for events
λ	Output function
ta	Time advance function
f	Rate of change function

Input, output and state variables are grouped into arrays and may be defined as*free* or *branch* quantities, respectively. Their sets are denoted by *U, Y,* and *X*, respectively. Without loss of generality we will restrict the subsequent discussion to $|U| = |Y| = 1$. As we will see later on, the dimension of $\vec{x}$ in Eq. (5.2) is independent of the number of state variables (i.e., branch quantities of energy storage elements present in the structural description of the circuit).

The state transition function δ denotes the changes of the array values in $\vec{x}$ in case of events and is composed from analytical functions as outlined in Eq. (5.4). Note that there is no distinction between internal and external events as in Eq. (5.1).

$$\delta : U \times X \times Y \longrightarrow X \tag{5.3}$$

$$\delta = \{h_i, i = 1,...,p\} \tag{5.4}$$

The new values to be assigned to the elements of x_i are determined from the functions in Eq. (5.4) such that

$$x_i^j = h_i\left(u^j,\vec{x}^{j-1},y^{j-1}\right). \tag{5.5}$$

They determine at time point j from the actual input u, from past values of states $\vec{x}$ and output y the new value of the state variable x_i. h_i are intrinsic functions of *BlockModel* in Eq. (5.2). The are denoted as *parameterfunctions* of the model because their actual relationships have to be established from a calibration process. Model calibration is detailed in the next section.

The output function λ determines the output value from the actual state of the system.

$$\lambda : X \longrightarrow Y \tag{5.6}$$

By adding an appropriate increment recursively to the past value of y, the new value at time point j is established from

$$y^j = y^{j-1} + \Delta y^j \tag{5.7}$$

$$\Delta y^j = \Delta t^j \cdot \frac{dy}{dt}\Big|_{t=t^j} \tag{5.8}$$

whereas,

$$y^o = DC\left(u^0\right). \tag{5.9}$$

The increment of the output value at each timestep j may be derived from a linear Taylor series expansion as given in Eq. (5.8). The calculation starts at time point 0 and, as an initial condition, exploits the DC operating point of the block as denoted in Eq. (5.9). This is a standard method in transient analysis of nonlinear circuits. The rate of change function f in Eq. (5.2) may be thus be viewed as *time derivative* of the output y

$$f(y,t) := \frac{dy}{dt} \tag{5.10}$$

and may be used to rewrite Eq. (5.7) as

$$y^j = y^{j-1} + \Delta t^j \cdot f(y,t)\,|_{t=t^j}. \tag{5.11}$$

The only problem with the algorithm given in Eqs. (5.7) to (5.11) is that it is not executable because a direct calculation of the increment of y at time point j in Eq. (5.8) requires the derivative value which, in turn needs the value of y at j. Therefore, an indirect method is proposed in the following, which is based on the analysis of the block behavior at its inputs and outputs. First, the time derivative of y using Eq. (5.10) may formally be stated as

$$f(y,t) = \frac{dy}{du} \cdot \frac{du}{dt}. \tag{5.12}$$

Thus, the increment of y in Eq. (5.8) is rewritten using Eq. (5.12) as

$$\Delta t^j \cdot f(y,t)\,|_{t=t^j} = \frac{dy}{du} \cdot \Delta u^j. \tag{5.13}$$

The derivative of y in Eq. (5.13) may easily be established from an input-output analysis of the system behavior resulting in the *slewrate function* of the block. However, the reaction of the dynamic system on changes at its inputs needs a more detailed discussion. First, as already shown in Fig. 5.6, a change of an input value is principally effective for a longer time period than the next integration timestep takes. Fig. 5.10 depicts a typical situation when applying an incremental increase of the input value (i.e., a step function) to a dynamic system.

The final steady state $\widehat{y}$ of the output can be calculated from the DC characteristic of the system, but it takes some time to reach this value. Note that in this case the primary input value does not change during the calculation of the

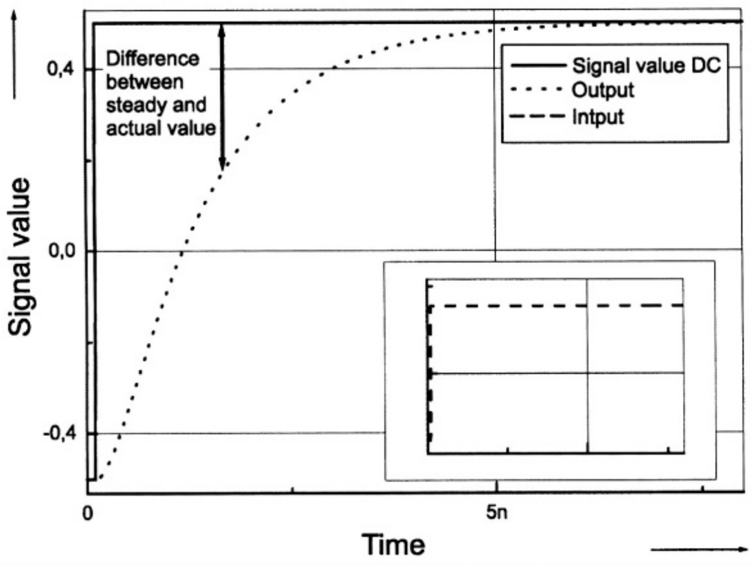


Figure 5.10. Step response of a dynamic system

output values by means of numerical integration. Therefore, an introduction of an *effective input value increment* $\Delta\widehat{u}$ for each timestep is required in order to cope with this situation.

$$\Delta\widehat{u}^j = g\left(u^j, u^{j-1}, y^{j-1}, \widehat{y}^j\right) \tag{5.14}$$

$$\widehat{y}^j = DC(u^j) \tag{5.15}$$

Secondly, as visible from Fig. 5.10, the system unveils a delayed reaction to the change of the primary input value. This is considered by introducing a delay function, which, in general, depends nonlinearly on the input signal of the block. Eq. (5.11) is now rewritten as

$$y^j = y^{j-1} + \Delta t^j \cdot \widehat{f}\left(\Delta\widehat{u}^j\right), \tag{5.16}$$

whereas the rate of change function $\widehat{f}$ is finally defined as

$$\widehat{f} := slewrate\left(delay\left(\Delta\widehat{u}^j\right)\right). \tag{5.17}$$

The time advance function *ta* of Eq. (5.2) consists, as already mentioned, of the combined ordered set of internal and external events and of a scheduling mechanism for introducing future events to this set according to actual values of the delay function. In contrast to the well-known processing of events in logic simulation, skipping directly to the time point of the next known event is not feasible here because events caused by the integration algorithm or state events have to be processed accordingly. This situation is depicted in Fig. 5.11. The calculation of the output value y should be done at time point $t + TD$ (i.e., *TD* time units after the event caused by the input change of u). This reflects the delayed reaction of the system. But the integration algorithm produces some events in the time interval $(t, t + TD]$ which cannot be simply skipped as in logic simulation.

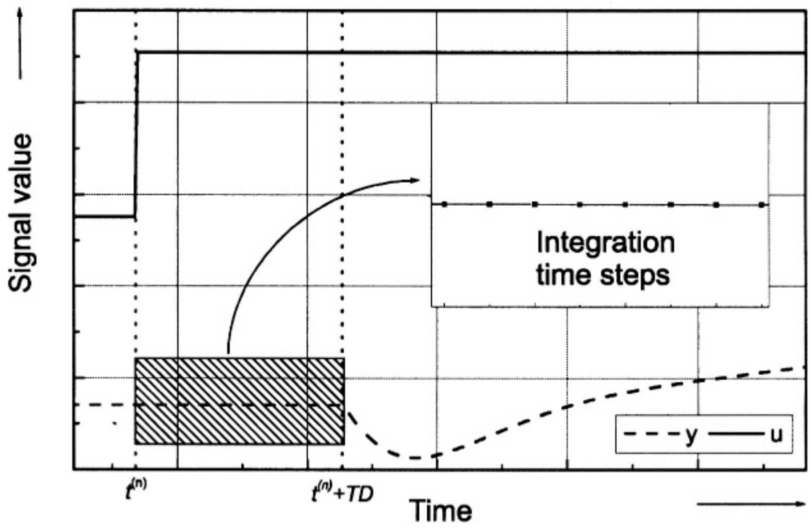


Figure 5.11. Integration steps prior to external event processing

There is a good chance that several events, which are adjacent to each other in time domain, have to be processed. First, this is because events at the input ports of a block affect the output waveforms for a considerably longer time than the next integration step takes, and secondly, the reaction time of the system depends primarily on the increment of the input values. Every event affects the resulting output behavior of the block in some way and should be considered appropriately. This situation is highlighted in Fig. 5.12.

At time point $t = 1$ there may be an external event at the input of the block. Depending on the amount of change of the input value, the delay of the system reaction is considered by setting a future event so that the output value is to be updated at $t = 5$ in this case. In between these two events an integration step may occur resulting in a new event at $t = 3$. As a result, the inputs are sampled at this time point. No change of input values is detected at $t = 3$, so the calculation of the new output value shall be done only 2.5 time units later. At $t = 5$ the output value calculation takes place — which, in general, is not yet

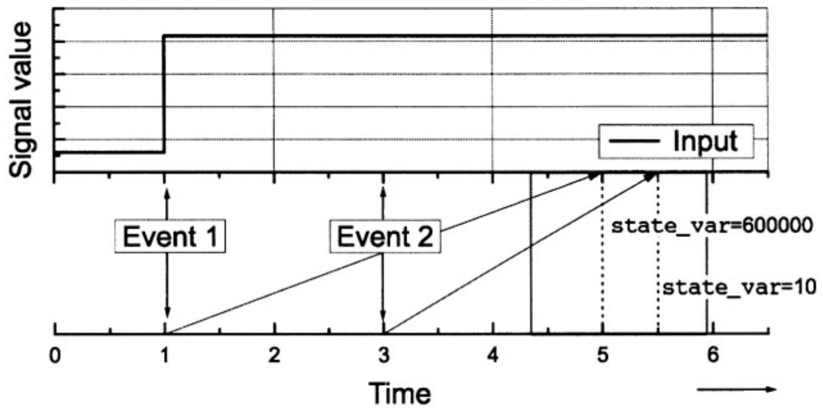


Figure 5.12. Effect of adjacent events

completed in one timestep only — as highlighted in Fig. 5.10. At $t = 5.5$ the effects of the two pending events have to be combined into one whole event. By comparing the actual value of the output to the steady state value to be reached, one can eventually assess the progress in completing the calculation for the event's effect. The actual implementation of the block model methodology exploits a maximum operation in order to determine the persistence of former events.

An informal description of the dynamic behavior of the outlined block model reads as follows:

At point in time t^j the state of the block model is at $\vec{x}^j$, input levels are at $\vec{u}^j$, and output levels are at $\vec{y}^j$. The next external event is set to time point $t^{j+1} = t^j + ta(\vec{x}^j)$ by means of the time function ta. Until then $\vec{x}^j$ holds for the elapsed time $e = ta(\vec{x}^j)$, i.e., the model is in the overall state $BlockModell^j = \{\vec{u}^j, \vec{y}^j, \vec{x}^j, \delta, \lambda, ta, f\}$.

At point in time t^{j+1} following actions are to be carried out:

1 Input vector $\vec{u}^{j+1}$ is updated.

2 The new assignment of the state variables $\vec{x}^{j+1} = \delta(\vec{u}^{j+1}, \vec{x}^j)$ is calculated by the transition function δ using the actual input signal and the previous state value.

3 The rate of change function calculates $f = \frac{d\vec{y}}{dt} \simeq \widehat{f}(\vec{u}, \vec{x}, \vec{y}, t)$. Thus, the slew rate of the output signal is known.

4 The time advance function ta adds an external event to the updated event list using the delay time (as calculated in Step 2 by δ) at which point in time the updated value of f and the state variables $\vec{x}^{j+1}$ are activated.

5 The output signals are set to $\vec{y}^{j+1} = \lambda(\vec{x}^j)$.

The transition function δ in Step 2 calculates at least the envisaged output value using DC analysis, the time displacement for the setting of events, and the slewrate of the output signal.

The case study to follow in this chapter shows how a specific functional block model is built and how the output signal waveform is calculated. Additional parameter functions may be added to scale the block model to different operating conditions such as environment temperature [Ham95].

The block model as detailed up to now is aimed at calculating the continuous time domain output waveforms for functional analytical level models. Such an instance of the *BlockModel* of Eq. (5.2) is denoted as *FctBlockModel.* In VHDL-AMS terminology this means that the `port quantity` values are determined by the *FctBlockModel,* but not the branch quantities as required at a `port terminal` to enforce conservation laws during behavioral model simulation of the block. At `port terminals` both the `across` and the `through` quantities have to be modeled such that an instatiation of a behavioral block model fits correctly — in terms of accuracy — to some extent into a structural component level description of a hierarchically specified module. In other words: The current-voltage characteristics of the block both at its input and at its output pins have to be provided within given accuracy ranges as a prerequisite to a correct mixed-level simulation of the module. Such a model is denoted as *BehBlockModel.*

The construction of a behavioral model can be viewed as a formal refinement or, by using the notation introduced in Chapter 2, as an *implementation* of a functional model. Consequently, a behavioral model at the algorithmic abstraction level is derived by an appropriate instantiation of a model specified previously on the analytical abstraction level. Signals at the connectors of the block have thus to be refined and their relations according to conservation laws require specification also, whereas the intrinsic functionality of the block model is still derived from the functional model. The generic architecture of a behavioral model resulting from this implementation step is depicted in Fig. 5.13.

The *v/i* characteristic at the input pins of the behavioral model is represented by a new parameter function denoted *Rin,* which results in a value for *RI.* The actual input voltage is forwarded as the input u to the functional block model. The Thevenin equivalent circuit depicted in Fig. 5.13 provides the electrical output behavior of the model. The output signal denoted as y of the internal functional block is assigned to the voltage source *VO* and an additional parameter function denoted *Rout* results in values for *RO* in order to model the *v/i*-characteristic at output pins. The functional model of the block is left unchanged, it is simply instantiated as the central part of the behavioral model.

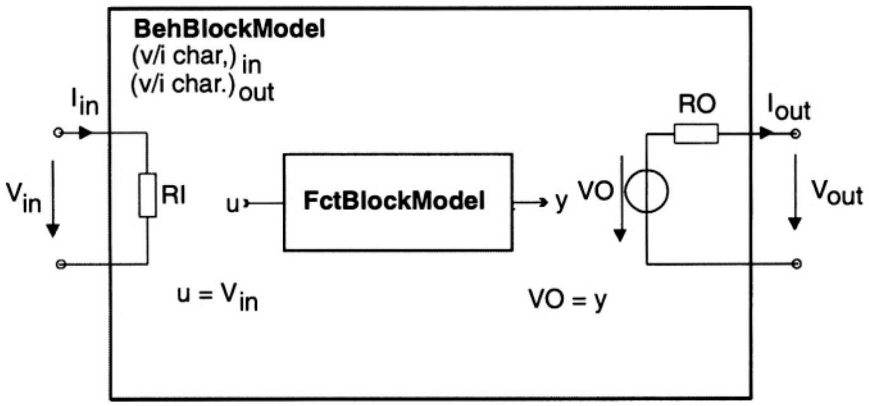


Figure 5.13. Generic architecture of a behavioral algorithmic level model

The parameter functions *Rin* and *Rout* are predominantly nonlinear. In addition, their results may depend on context properties of a behavioral model instantiation such as output load conditions. This is detailed in Chapter 6 for an opamp circuit example.

The sets of parameter functions needed for the representation of mixed-signal circuits as functional or behavioral block models, respectively, according to the proposed methodology are now complete. They are summarized in Table 5.2. The elements of $\vec{x}$ in Eq. (5.2) (i.e., the states of these models) contain the values of the parameter functions h_i as given in Eq. (5.5) or values calculated from these functions in each time step during model execution such as the effective input value as outlined in Eq. (5.14). The number of states within a *BehBlockModel* or *FctBlockModel,* respectively, is thus a constant (i.e., it does not depend on the number of real state variables as in Eq. (5.1)).

Table 5.2. Fundamental parameter functions of block models

Abstraction Level	*Model Architecture*	*Parameter Function*
Analytical	FctBlockModel	DCtf, SlewRate, TDly
Algorithmic	BehBlockModel	Rin, Rout, DCtf, SlewRate, TDly

Fig. 5.14 summarizes the process of generating models according to the proposed methodology. Starting from a representation of the block to be modeled at a lower abstraction level (i.e., component or procedural level as depicted in Fig. 2.8) and a characterization plan, the parameter function values are ex-

tracted from an overall rather complex characterization run, thus yielding an abstraction from lower to higher levels. The parameter function values determined in this way are available in a tabular form from this simulation-based characterization. The table representation is, however, not an efficient data structure for the modeling tasks to be accomplished in the next step. Therefore, an analytic representation of this raw data is derived next, which results in a compact and portable data base: A piecewise defined set of analytic relationships to be used as the parameter functions summarized in Table 5.2.

Additional block properties, such as its operating conditions dependent behavior, may easily be captured by means of introducing additional parameter functions to the fundamental set given in Table 5.2.

An intermediate representation of a functional or a behavioral model, respectively, may automatically be converted into an executable model description as detailed in Fig. 5.14. Upon user request the available code generators produce executable codes in either one of the description languages from the primary data base of a block model: Eldo-FAS, Mathematica, Saber-MAST, and VHDL-AMS. This flexibility is a unique property of the outlined approach to model generation.

5.5 Model Calibration

After completion of the simulation and extraction steps defined in the characterization plan as discussed in Chapter 4 all data required for the calibration of the functional block model according to Eq. (5.2) and for a possible refinement into a behavioral model as shown in Fig. 5.13 is available. At this point the parameter functions of the model are represented, as depicted in Fig. 5.14, in tabular form. Tables need, however, a considerably large amount of storage space. In addition, looking-up of data from and performing interpolation steps based on table entries as needed to be done frequently at run time of the simulation take much too long because large tables cannot be stored efficiently as part of the executable code of a model. Therefore, the model execution efficiency can be increased considerably by introducing mathematical models of the tabular representations of parameter functions.

Thus, the basic task consists of describing a set of ordered data by means of a mathematical model. Well-known fundamental approaches to solve this problem are *interpolation, response surface methods* [BD87, KC87, Mon91], and last but not least, *regression analysis.* For reasons detailed elsewhere [Ros01] none of these basic approaches is appropriate for a direct application to the task of model calibration. The approach presented in the following may be classified as being part of general regression analysis, but there are several significant additional features to be mentioned. First, there is a comprehensive set of linear and nonlinear regression models available. Secondly, distance metrics and error norms for the quantification of accuracy may be selected from a va-

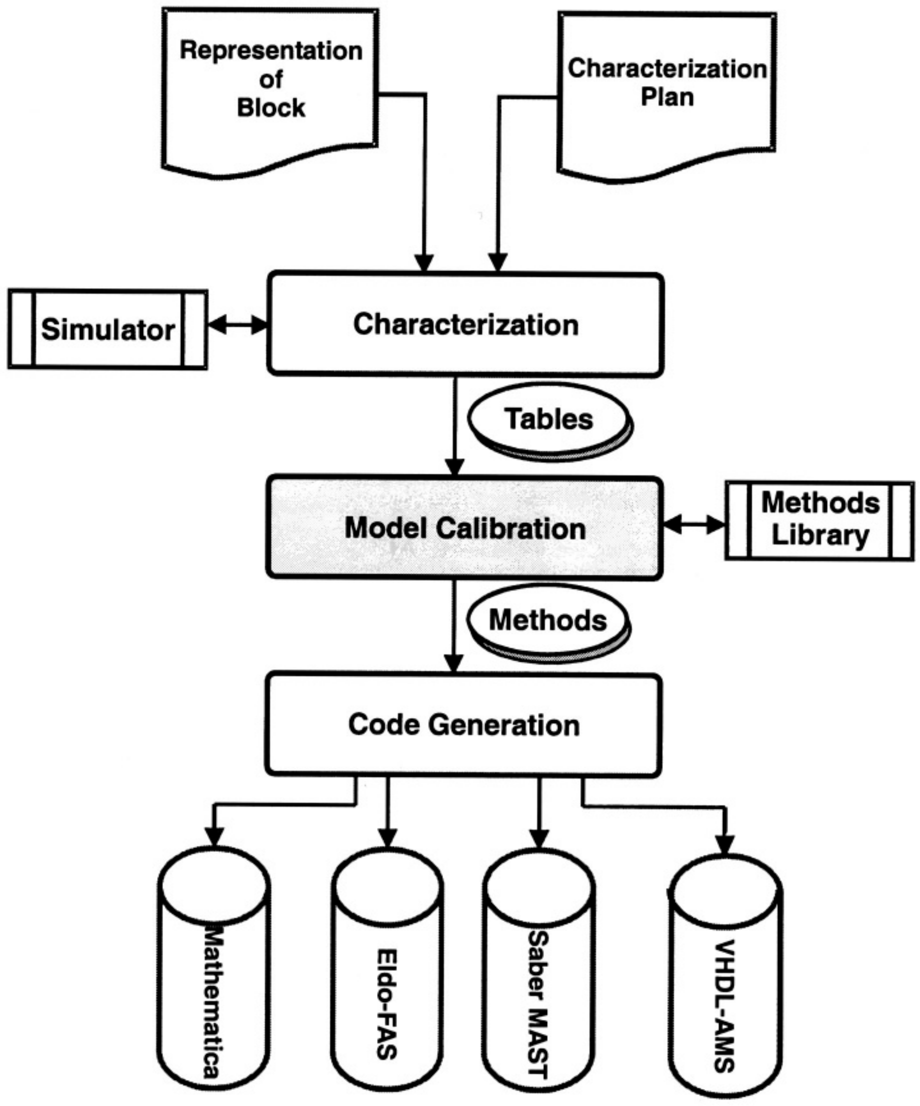


Figure 5.14. Model generation methodology

riety of distances and norms, and thirdly, the calibration process automatically subdivides value ranges into an ordered set of *segments* (i.e., intervals) and selects regression models such that predefined error bounds are not exceeded during model calibration.

A functional relation between an independent variable denoted as x in the following and the dependent variable denoted as y by means of a generally unknown mapping function f is denoted as

$$f : x \rightarrow y. \tag{5.18}$$

The functional relation in Eq. (5.18) is to be determined using regression analysis by applying a so-called *method* according to the definition given in the following.

DEFINITION 5.1 *Each segment is mapped by means of a regression model function φ to fit the dependent data set gained from characterization for the segment at hand*

$$\varphi_i : \omega_i \rightarrow f_i(x). \tag{5.19}$$

The range of the independent variable x is subdivided into segments such that

$$\omega_i : (x_{i1}, x_{i2}] = \{x | x \in \boldsymbol{R}, x_{i1} < x \leq x_{i2}\}. \tag{5.20}$$

A Method $M = \{\Omega, F\}$ consists of two sets — a set of segments $\Omega = \{\omega_j : 1 \leq j \leq n\}$ and a set of regression functions $F = \{\varphi_k : 1 \leq k \leq m\}$.

Following properties are provided by methods:

Completeness: Let $\boldsymbol{D}_f$ be the domain of the independent variable to be modeled. Then: $x \in \boldsymbol{D}_f \Rightarrow \exists i : x \in \omega_i$.

Unambiguousness: For each pair of segments (i, j) with $i \neq j$ it follows: $x \in \omega_i \Rightarrow x \notin \omega_j$.

C_0-Continuity: For two subsequent segments $\omega_1 : (x_i, x_j] \rightarrow f_1(x)$ and $\omega_2 : (x_j, x_k] \rightarrow f_2(x)$ it follows: $\lim_{x \rightarrow x_j} f_2(x) = f_1(x_j)$. This property is important to ensure the convergence of the simulation runs.

Preciseness: For each input value x_i, each corresponding output value y_i, and a maximal error of ε it follows that: $f(x_i) - y_i \leq \varepsilon$. Different norms standards (l_1, l_2 and l_∞) [Box78] and distances (e.g., absolute and Euclidean distance) [JRS91] may be selected for the computation of ε.

The results of the characterization of a block usually yields a tabular representation of results as (T_u, T_y), whereas T_u denotes the independent variable (i.e., the input signal) and T_y denotes the dependent variable (i.e., the output signal) produced by the block to be modeled in the notation of the *BlockModel*. The fitting of the dependent data to regression models is accomplished, as outlined in (e.g. [Rat83]), by a *methods library,* which combines both standard and user defined regression models, distance and error norm metrics, as well as algorithms for an automatic accuracy-driven segmentation into a set of methods. The methods library is implemented mainly on top of the Mathematica tool set [Ham95, Ros94a].

Fig. 5.15 visualizes the effects of model calibration by means of exercising the methods library. The tabular steady-state relation (T_u, T_y) of the magnitudes of input and output signals, respectively, produced by DC analysis of a functional block (i.e., the parameter function `DCtf` of the block model) is depicted as a graph in Fig. 5.15a). Taking (T_u, T_y) as input data, the methods library produces a method `M` for `DCtf`, which features a segmentation of u into five intervals, whereas a different regression model is parameterized in each segment in order to meet user defined accuracy requirements on the fitting results for y. Fig. 5.15b) depicts the effects of this approach to calibrating `DCtf`, which is produced by invoking

```
DCtf = BuildMethod [ DCtfTable, DistanceMetric -> Euclid,
                     Smooth -> Yes, Norm -> Infinity,
                     MaxDegree -> 8, MaxError -> 3 ]
```

for a maximum fitting error of 3%. The approximation is performed according to user specified distance and norm metrics and limits the degree of polynomials to be used as regression models to 8 as detailed above. The content of the large table (T_u, T_y) of raw data is thus compressed into a set of segments Ω for u and into a set F of parameterized regression models for y, which can be evaluated much more efficiently at run time of the model compared to a table look-up from (T_u, T_y).

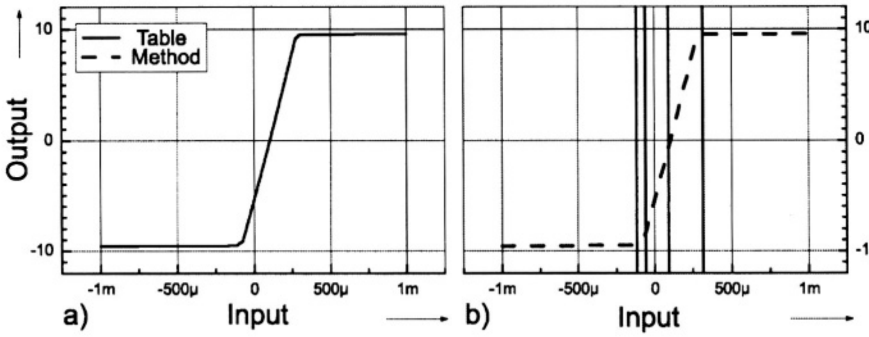


Figure 5.15. Model calibration by means of the methods library a) Table T_u, T_y from block characterization b) Method for the parameter function `DCtf`

Compressing raw tabular data into methods as proposed rises questions about the resulting accuracy of the calibration process. Let us return for convenience reasons to continuous time representations of model behavior for the discussion of these issues. By denoting $y(t)$ the reference output waveform of the block in $t \in [a, b]$ and $\tilde{y}(t)$ the waveform produced by the functional or by the behavioral block model in the same time interval, we define d_p^m as the distance metric from $y(t)$ to $\tilde{y}(t)$ as

$$d_p^m(y, \tilde{y}) = \|z_m\|_p \, . \tag{5.21}$$

Here, we denote different distances by z_m and different error norms by $||z_m||_p$. The latter establish how differences (i.e., distances of y and $\tilde{y}$) are accumulated into the error, which, in turn acts as a metric for the accuracy of the block model. Following norms are available as part of the methods libray:

1 l_1 norm. Differences $z(t) = y(t) - \tilde{y}(t)$ are accumulated in z_1 according to

$$\|z\|_1 = \int_{t=a}^{b} |z(t)| dt \, . \tag{5.22}$$

2 l_2 norm. This norm is the basis for many least squares fitting algorithms.

$$\|z\|_2 = \sqrt{\int_{t=a}^{b} z(t)^2 dt} \tag{5.23}$$

1 l_∞ norm. The maximum distance in vertical direction is established from

$$\|z\|_\infty = \max_{t \in [a,b]} |z(t)| \tag{5.24}$$

and is exploited by many *MinMax* algorithms. However, when specifying the maximum procentual error of the calibration process, a modification of the norm definition is required such that

$$\|\tilde{z}\|_\infty = \max_{t \in [a,b]} \left| \frac{z(t)}{y(t)} \right| \tag{5.25}$$

holds. The selection of appropriate norms (i.e., assigning to p the appropriate value from $(1, 2, \infty)$) is essential to the quality of model calibration. This well-known requirement is highlighted in Fig. 5.16. The effects of errors stemming from z_1 and z_2 norms, respectively, are obvious from this figure.

The consideration of the vertical distance

$$z(t) = y(t) - \tilde{y}(t) \tag{5.26}$$

as used before is, in general, not sufficient to decide whether a regression model is suited to fit a reference waveform $y(t)$ within a given interval [YA91].

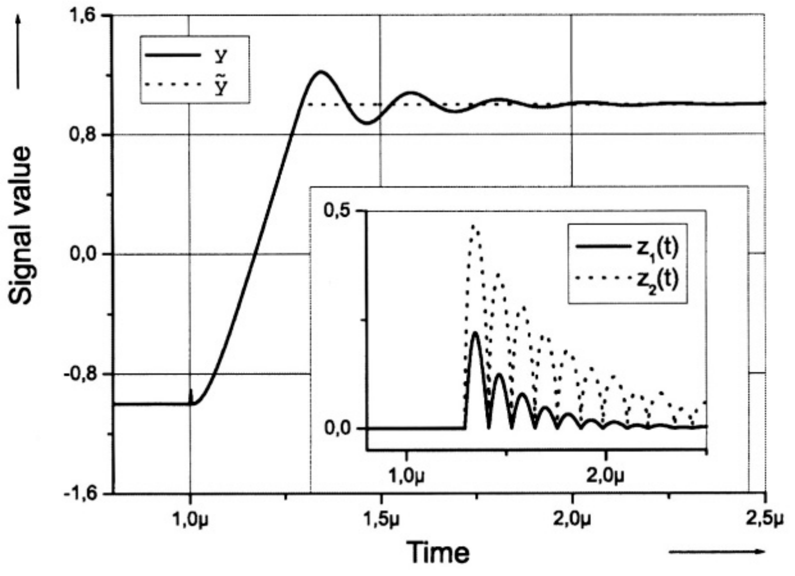


Figure 5.16. Comparison of reference and model behavior by different error norms

A better approach is based on the introduction of *Euclidean distances* z_E defined by

$$z_E(t_i) = \min_{t \in [a,b]} \sqrt{z^2(t) + s^2(t - t_i)^2} \,. \tag{5.27}$$

Its geometrical interpretation is depicted in Fig. 5.17. The minimum dis-

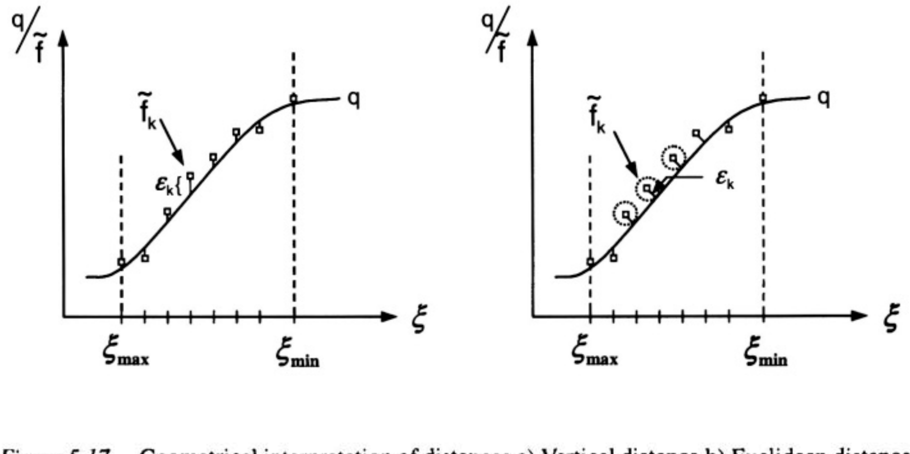


Figure 5.17. Geometrical interpretation of distances a) Vertical distance b) Euclidean distance

tance from each point of the reference to the model waveform is the radius of a circle around the point, which is intersected by the model curve in exactly one point. When combining the Euclidean distance with the error norms summarized above, one yields

$$d_p^E(y,\tilde{y}) = \left\|E_{y-\tilde{y}}(t)\right\|_p = \|E_z\|_p \,. \tag{5.28}$$

The set of distance metrics available in the methods libray is acccomplished by introducing the horizontal distance by means of

$$z_H(t_0) = s\,|t_1 - t_0|\,, with\ f(t_1) = y_0\,, \tag{5.29}$$

$$d_p^H(y,\tilde{y}) = \left\|H_{y-\tilde{y}}(t)\right\|_p\,, \tag{5.30}$$

whereas s denotes, as for the Euclidean distance, a scaling factor. For the distance metrics given in Eq. (5.21) we thus exploit $m \in \{E,H,V\}$ for the Euclidean, horizontal, and vertical distances, respectively, and $p \in \{1,2,\infty,\tilde{\infty}\}$ for the l_1, l_2, and l_∞ norms, as well as the l_∞ norm variant as defined in Eq. (5.25), respectively.

All the distance metrics and error norms discussed previously are implemented as part of this methods library. They thus support a comprehensive approach to model calibration by fitting the parameter functions of functional and behavioral block models to characterization data. The layered structure of the methods library is outlined in Fig. 5.18. The library is implemented as a set of Mathematica packages, which exploit both proprietary procedures written in C and commercially available numerical packages such as the NAG[1] library in proper sequence.

The main objective of generating efficiently executable models is to approximate parameter functions by as large segments as possibile, which meet given error bounds by using linear regression models only. A segmentation process for linear regression is highlighted by the algorithm of Fig. 5.19.

In addition to linear regression models, polynomials of variable order and exponential functions such as the Lampertz-Gomez model are available in the methods library to be used for nonlinear regression modeling. A user-defined extension of the set of regression models is also supported. Details on this library and its implementation may be found in [Ros94b, Ham95, Ros01].

The proposed calibration approach allows to easily generate differently accurate models from the same raw data by just exercising the methods library for different error bound specifications. As a result, the generated representations of the parameter functions become more or less complex depending on

[1]Numerical Algorithms Group, Inc.

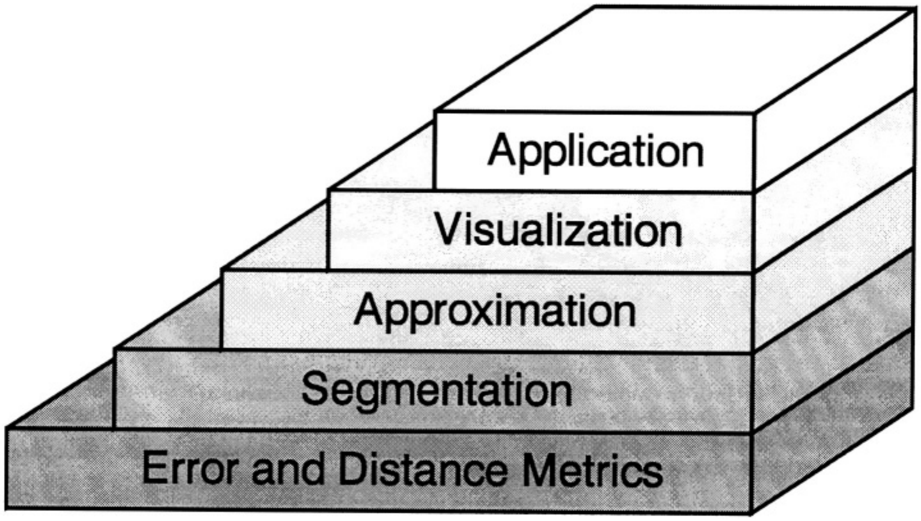


Figure 5.18. Layers of methods library

```
function Linearize (Data: set of points (Tu, Ty))

   n = Length (Data)
   f = TwoPoint (Data[1], Data[n])
   Error = MaxError (Data, f)

   (* check whether the error bound is met *)
   (* otherwise perform interval segmentation *)

   if (Error =< MaxErrorAllowed) then
       Return f

   else
       j  = IndexOfPointWith Error(Data, Error)
       f1 = Linearize (Data[1...j])
       f2 = Linearize (Data[j...n])
       Return (f1 u f2)
   endif
end (Linearize)
```

Figure 5.19. Pseudo-code of interval segmentation for linear approximation

the preselected accuracy levels. A trade-off between accuracy and execution speed of the block models is thus possible — completely under user control, within a few minutes, and without the need to rerun circuit characterization.

5.6 Case Study: A Linear Dynamic System

In the following, let us consider a simple example in order to highlight the roles of functional and behavioral models and to outline the detailed modeling approach. A linear dynamic system of second order is introduced as an example in Fig. 5.20. Its functionality is explicitly specified by differential equations. We thus can denote its functional representation in VHDL-AMS by directly exploiting the DESS class model of Chapter 1 as outlined in Fig. 5.21. The values of the coefficients A, B, and C may be assigned to constants.

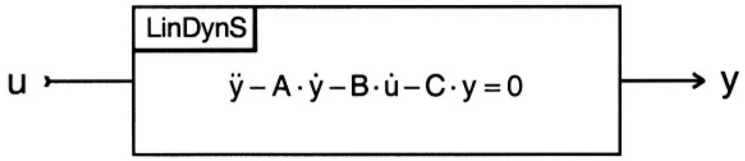


Figure 5.20. Linear dynamic system of 2. order

```
entity LinDynS is
  port ( quantity U: in Real;
         quantity Y: out Real);
end entity LinDynS;

architecture DESS of LinDynS is
  constant A: Real := ... ;
  constant B: Real := ... ;
  constant C: Real := ... ;
  quantity X: Real;

begin
  X == Y'dot;
  X'dot == A*X + B*U'dot + C*Y;
end architecture DESS;
```

Figure 5.21. DESS model of the linear dynamic system

The causal model of Fig. 5.21 can be exploited in a straightforward manner for functional level simulation of the block `LinDynS` after assigning values to `A`, `B`, and `C`, respectively. When aiming at behavioral simulation, we first have to select a physical domain for the block to operate in. Secondly, we need to redefine the entity declaration in order to accommodate the new quality of input and output signals (i.e., to arrange for the consideration of conservation laws in the chosen physical domain). According to the generic architecture of behavioral models shown in Fig. 5.13 the next steps consist of arranging for the calculation of branch quantities at the input and output connectors of the behavioral model and of instantiating the functional model.

A structural implementation in the electrical domain of the functionality denoted in Fig. 5.20 is depicted in Fig. 5.22. From the component values and the load condition of the circuit we have to establish how the additional information for the behavioral model has to be provided. The resulting model code is summarized in Fig. 5.23.

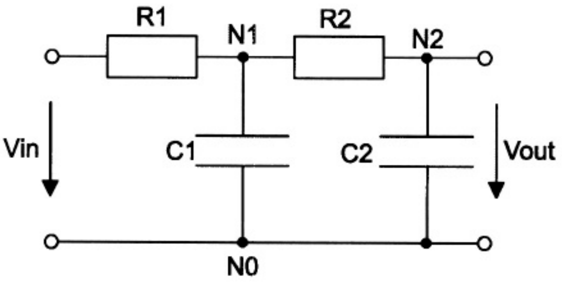


Figure 5.22. Passive RC circuit

In case that a DAE representation of the functionality of the block available already exists, as for the DESS model of the example system, then there is no need at all to apply the proposed methodology for the generation of functional block descriptions. The common situation in practice, however, is given by the fact that one has some structural description of a block available either at component or at procedural level and is faced with the requirement to somehow provide efficient and, at the same time, accurate functional or behavioral algorithmic level models.

So, let us exercise the proposed model generation process for the linear dynamic system at hand. Starting from its structural representation in the electrical domain as depicted in Fig. 5.22, a functional block model is to be generated and then to be represented as a new architecture denoted as FctBlockModel for the entity LinDynS of Fig. 5.21. This model may then be instantiated within the behavioral model of Fig. 5.23 by just replacing the DESS architecture of LinDynS. This reads as

```
FM: entity LinDynS(FctBlockModel)
  port map ( U => Vin, Y => VO);
```

```
entity LinDynS_Electrical is
  generic ( Rload : Real := 100 );
  port ( terminal InP, OutP : Electrical );
end entity LinDynS_Electrical;

architecture Behavioral of LinDynS_Electrical is
  quantity Vin across Iin through InP to Electrical'reference;
  quantity Vout across Iout through OutP
                to Electrical'reference;
  quantity VO: Real;

  function Rin(...) return Real is ... return RI; end;
  function Rout(...) return Real is ... return RO; end;

begin
  Vin == Iin * Rin(...);
  Vout == VO - Rout(...) * Iout;
  FM: entity LinDynS(DESS)
    port map (U => Vin, Y => VO);
end architecture Behavioral;
```

Figure 5.23. Behavioral model of RC circuit

According to Table 5.2 we need the parameter functions `DCtf`, `TDly`, and `SlewRate` calibrated to the characterization results gained from component level simulation for the generation of a functional model. These simulation results are to be viewed as the reference for the subsequent assessment of modeling quality.

At this point, it is highly recommended to express one's expectations on the outcome of the calibration process. From inspecting the circuit we thus expect an identity, a constant, and a linear function for `DCtf`, `TDly`, and `SlewRate`, respectively. A characterization plan defines the analysis process to be performed: First, a sweep in DC domain of the input signal magnitude within given bounds and, secondly, a sequence of simulation runs in time domain using a parameterized step function as the input signal to the circuit. Transfer curve values, slew rate, and propagation delay values are calculated from these simulation results, which are then arranged into tabular form. The outlined model calibration process takes this raw data and produces — as detailed in the previous section — the methods for an approximation of the parameter functions according to a user specified error bound of 1% in this case. These calibration results are depicted in Fig. 5.24, 5.25, and 5.26, respectively.

The graphical representations of `DCtf` and `SlewRate` as depicted in Fig. 5.24 and 5.25, respectively, are very close to our previously mentioned expectations. The calibration results for `TDly` in Fig. 5.26, however, seem to

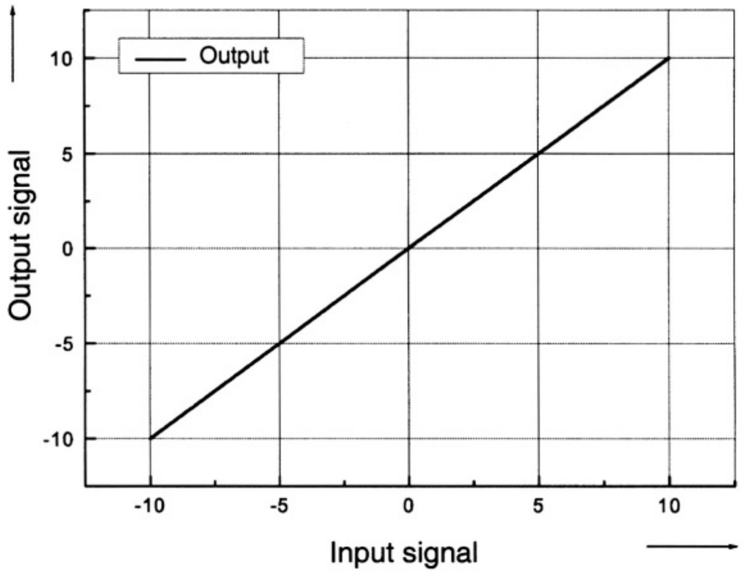


Figure 5.24. Calibrated parameter function `DCtf`

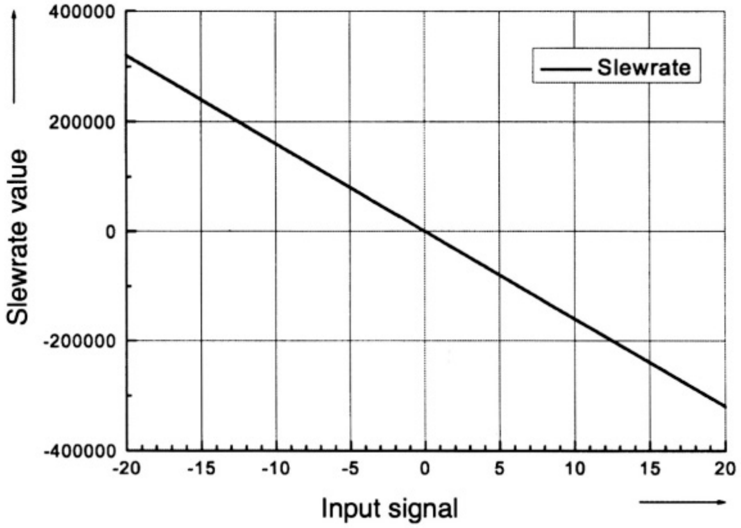


Figure 5.25. Calibrated parameter function `SlewRate`[2]

[2]The input signal was defined oppositely to usual definitions in this case thus resulting in the depicted relationship.

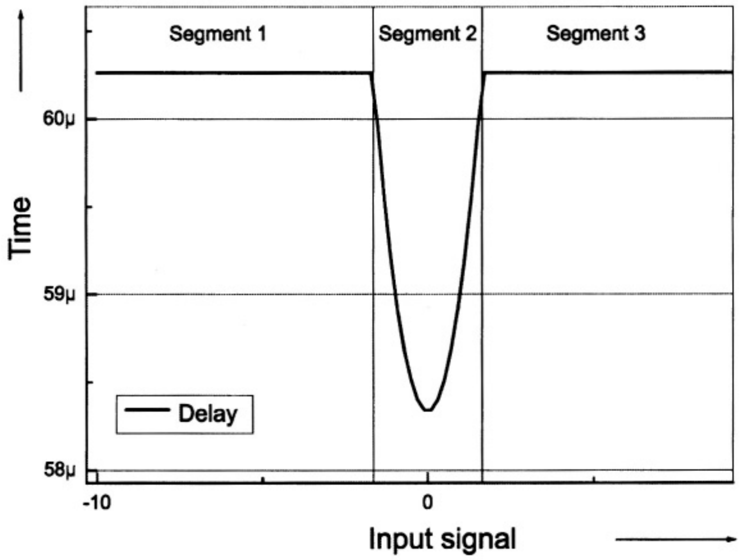


Figure 5.26. Calibrated parameter function `TDly`

be strange. Instead of a constant delay for the complete range of input signals we yield a constant value only for input signals exceeding a minimum magnitude. When recalling how the propagation delay is calculated (i.e., from a difference operation at points in time for input and output signals featuring the same value of magnitude such as 50% of the swing), it becomes clear that the problem stems from a loss of precision while performing numerical subtraction operations on numbers, which are both close to zero and are represented by the same mantissa length. Thus, we should correct these calibration results in our model representation accordingly. At this point it should have become clear, what the term *model engineering* as used throughout this book really means. It is always good advice to any user of computer-aided design tools to rely more on her or his own engineering skills and problem knowledge than on automatically produced results.

Transparent methods for generating behavioral or functional models and a support of sensible user interactions are obvious prerequisites for self-dependent design engineers.

Fig. 5.27 details the automatically produced functional analytical level model and the manual corrections of the parameter function `TDly`. The architecture of the model consists of piecewise definitions of parameter functions based on a model calibration by means of the methods library, the calculation of the output signal from the rate of chance function (i.e., the time derivative), and the insertion of events by break statements. In addition, the range of input values, in which the model has been calibrated, is considered in the model code. This is a good practice because a model generally does not predict the block behavior reliably in another application context than that being addressed during calibration. Finally, Fig. 5.28 depicts some results of a simulation run gained from executing this functional block model. The overall maximum error compared to the reference result produced by circuit level simulation is less than 3%.

The parameter functions `Rin` and `Rout`, respectively, as required for the behavioral model of Fig. 5.23 are strongly related to both the electrical implementation of the linear system as shown in Fig. 5.20 and to the context the block operates in as denoted in Eq. (2.2). In addition, their results `RI` and `RO`, respectively, are nonstatic (i.e., time dependent), as can easily be derived from an inspection of the circuit in Fig. 5.22. These circuit-specific and time-dependent parameter functions may be expressed [AB95], but a detailed analysis of what is to be done for their specification and execution results in almost the same work a SPICE-like simulator would accomplish anyway based on a structural component level description as its input. A modeler should, therefore, stick to the component abstraction level for efficiency reasons and should not attempt to produce an artificial high-level behavioral model when dealing with such circuits.

This situation changes completely when aiming at an implementation of the linear dynamic system example by means of active circuits — either by simply inserting impedance converters into the signal flow of Fig. 5.22 or by exploiting some active filter structure. Now, the input and the output of the resulting architecture are almost decoupled from each other, thus opening the door for a straight forward application of the generic high-level behavioral model depicted in Fig. 5.13.

Selections of abstraction levels and of modeling styles are obviously not independent from architectural details of the circuit to be modeled. This fundamental characteristic has far-reaching consequences for model generation as demonstrated in Chapter 6.

```
architecture FctBlockModel of LinDynS is

-- Begin ******* method SlewRate ********
  function SlewRate(U: real) return real is
    variable SR: real;
  begin
    if U <= -1.994e-2 then
      SR := -1.5485123e1 - 3.1997704e4 * U;
    elsif U <= 1.994e-2 then
      SR := -1.0659833e-4 - 3.1221118e4 * U;
    else
      SR := 1.5485123e1 - 3.1997704e4 * U;
    end if;
    return SR;
  end;
-- End ******* SlewRate ********

-- Begin ******* method DCtf ********
  function DCtf(U: real) return real is
    variable DC: real;
  begin
    DC := 1.00425e0 * U;
    return DC;
  end;
-- End ******* DCtf ********

-- Begin ******* method TDly ********
  function TDly(U: real) return real is
    variable TD: real;
  begin
-- Begin MANUAL CORRECTION
--if U <= -1.64422e0 then
--      TD := 5.5312937e-5 - 3.8135127e-11 * U;
--elsif U <= 1.64422e0 then
--      TD := 5.71405e-5 - 6.1171557e-21 * U
--                       - 6.7598588e-7 * U ** 2.0e0;
--else
--      TD := 5.5312937e-5 + 3.8135127e-11 * U;
--end if;

    TD := 5.5312937e-5;

-- End MANUAL CORRECTION
    return TD;
  end;
-- End ******* TDly ********
```

Figure 5.27. Functional block model of the linear dynamic system

```
  quantity TDv: real;
  constant Ymin: real := -10.0;
  constant Ymax: real := 10.0;

begin
  Ueff == DCtf( U ) - Y;
  TDv == TDly( Ueff );

  if (now < 0.2e-9 ) use
    Y == DCtf( U );
  elsif (Y >= Ymax) and
    (SlewRate( Ueff'delayed( TDv )) > 0.0) use
    Y == Ymax;
  elsif (Y <= Ymin) and
    (SlewRate( Ueff'delayed( TDv )) < 0.0) use
    Y == Ymin;
  else
    Y'dot == SlewRate( Ueff'delayed( TDv ) );
  end use;

  break when now = TDv;
  break when Y'above(Ymax);
  break when not(Y'above(Ymin));

end FctBlockModel;
```

Figure 5.27 (continued). Functional block model of the linear dynamic system

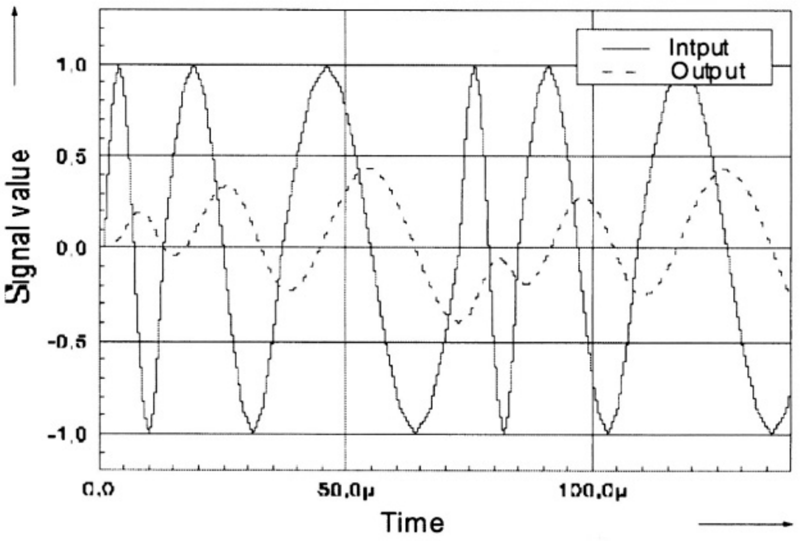


Figure 5.28. Simulation results gained from executing the functional block model

Chapter 6

APPLICATION EXAMPLES

Modeling the behavior of mixed-signal circuits in time domain may or may not be a complex problem — it depends on accuracy and efficiency requirements and, to a high degree, on the the view a modeler prefers. Various trade-offs concerning model accuracy and, last but not least, the reuse of third party IP and standard products have to be thoroughly considered, when introducing high-level models into the design flow of integrated circuits.

6.1 Overview

This chapter is dedicated to a demonstration of the proposed methodology to show that it is suited for real world applications. A direct generation of behavioral algorithmic level models and their exploitation within mixed-level structural descriptions are emphasized. However, a sensible trade-off has to be established between circuit sizes, length of model descriptions, proof of objective fulfillment, and available text space for its documentation. Two quite different examples were taken from the set of circuits modeled by the proposed model generation approach: a bipolar active filter (i.e., an analog functional block) and a CMOS A/D converter (i.e., a mixed-signal circuit). These application examples differ in terms of application areas, circuit sizes, and semiconductor fabrication technologies. However, they have many properties in common. First, both circuits operate at large input signal conditions in time domain unveiling a nonlinear behavior. Secondly, they are both constructed from an interconnection of standard components being available as IP products. Thirdly, each of the resulting models is represented by a mixed-level structural description of the circuit. This model consists of instantiations of the behavioral algorithmic models level in addition to the representations of IP blocks, respectively, and of primitives denoted at component level.

Section 6.2 emphasizes characterization and calibration of parameter functions to be used within a behavioral model at algorithmic level. It is mainly aimed to a demonstration of the achievable accuracy and efficiency of the proposed methodology with analog waveform representation in mind. In contrast, Section 6.3 addresses the modeling of mixed-signal circuits. The complete model of a 6 bit A/D converter is developed step by step and, although lengthy, documented such that it may easily be executed by means of a VHDL-AMS simulator. Some advantages of this detailed model compared to a generic representation of the converter are highlighted. Finally, Section 6.4 is dedicated to some conclusions and to remaining open research areas.

6.2 Active Filter Circuit in Bipolar Technology

The first application example is a biquad filter taken from [CY94] and depicted in Fig. 6.1. The schematic of this block consists of several resistors, capacitors, and four operational amplifiers — all of the same type. A generation of a behavioral model for the entire filter block according to the outlined methodology is possible, but not feasible in this case for the following reasons. First, the characterization of the entire, somewhat complex block at component level consumes a considerable computation time and requires a lot of storage space for its results. The second and most important reason for not trying to generate a model for the whole block is directly related to reuse issues. A model of this specific filter is not likely to be used in as many system design applications as the behavioral model of an opamp (i.e., a basic building block of many different analog and mixed-signal circuits), which is instantiated four times in this filter circuit only.

With model engineering in mind, it is obviously much more appropriate for reuse purposes to generate a behavioral algorithmic level opamp model first and then to instantiate it in a mixed-level structural description of the filter according to the schematic of Fig. 6.1. The operational amplifier to be used in this filter circuit is a standard component fabricated in a bipolar technology [Ray90]. The vendor of this product denoted as MOPA1 provides two structural models represented in SPICE code as part of its product documentation: a circuit and a macro level model. These are component and procedural level models respectively according to the terminology introduced in Chapter 2. The schematic of the opamp derived from the netlist of the component level model is depicted in Fig. 6.2.

The component level model is taken as reference for the calibration of the behavioral model, which as to be done in both DC and time domain. The characterization data, which forms the basis for the calibration by the methods library discussed in Chapter 5, has to be produced with different testbenches for the circuit under test (CUT). This is quite a common situation in analog

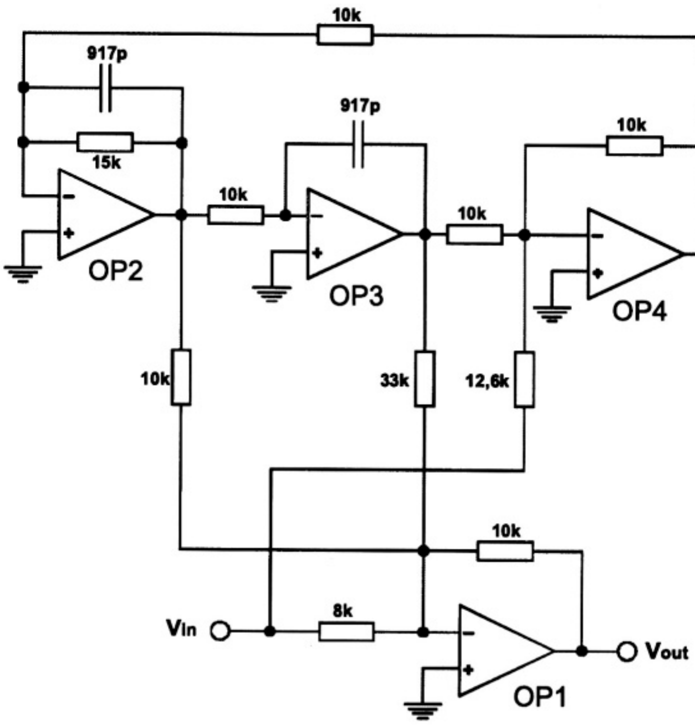


Figure 6.1. Schematic of a biquad filter

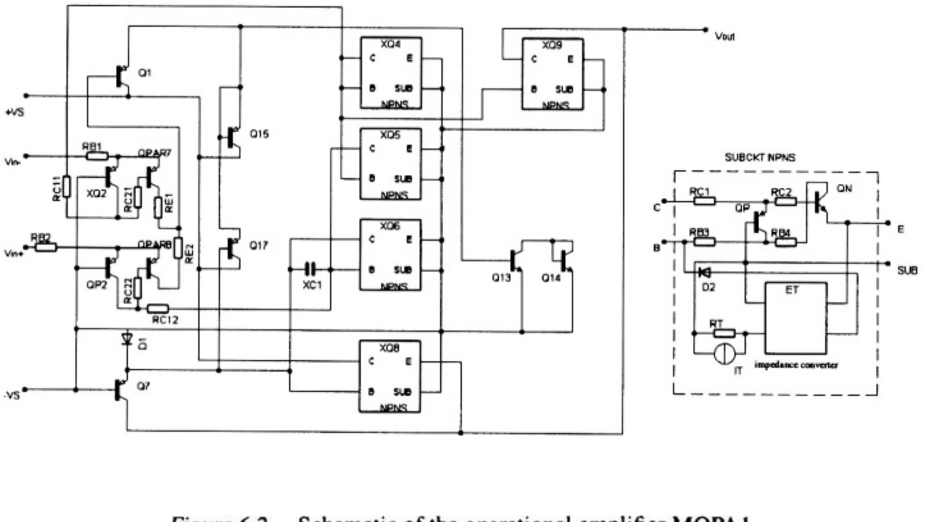
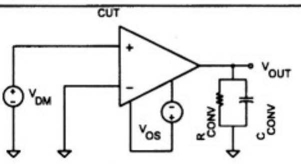
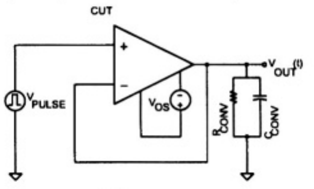
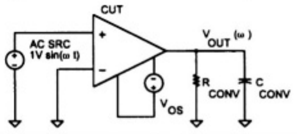


Figure 6.2. Schematic of the operational amplifier MOPA1

circuit design. Table 6.1 depicts some testbenches required for an extraction of key properties in DC, time, and frequency domain, respectively.

Table 6.1. Testbenches for opamp property extraction

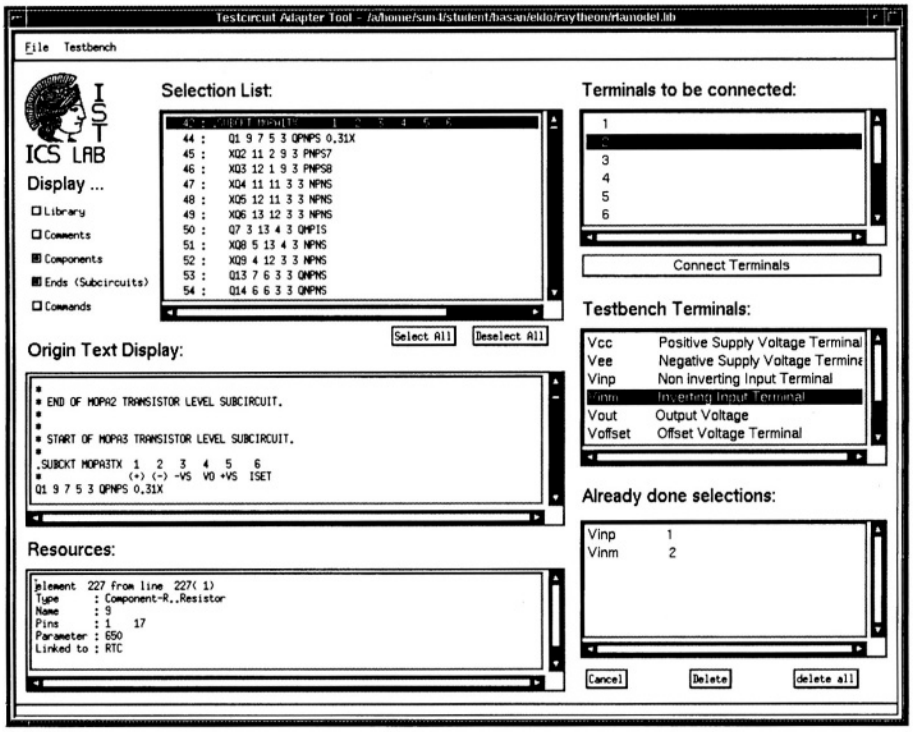
Property	*Domain*	*Testbench*
Offset voltage, output range	DC	 CUT V_{OUT} V_{DM} V_{OS} R_{CONV} C_{CONV}
Slew rate, delay, setting time	TRAN	 CUT $V_{OUT}(t)$ V_{PULSE} V_{OS} R_{CONV} C_{CONV}
Gain, transit frequency, phase margin	AC	 CUT AC SRC 1V sin(ω t) $V_{OUT}(\omega)$ V_{OS} R CONV C CONV

As a consequence, the circuit description (i.e., the netlist) first has to be connected to different testbenches, which are then sequentially compiled together with definitions of signal ranges and the control of the circuit simulator into a characterization plan as already discussed in Chapter 4. A manual setup of testbenches is too tedious and error prone. This task is best supported by a testbench adapter tool. The user interface of such a tool, which is part of the ViCE characterization environment, is highlighted in Fig. 6.3 for the setup of the opamp. Details of this tool may be found elsewhere [Goe01].

Next, the characterization plan has to be specified in order to extract all the raw data from simulation, which is needed to calibrate at least the parameter functions `DCtf`, `SlewRate`, and `TDly`, respectively, according to the modeling methodology presented in Chapter 5. The plan is rather complex even for a relatively small circuit such as this opamp. Therefore, Fig. 6.4 depicts only a part of it — the visually specified plan for slew rate data extraction to be executed by ViCE.

As previously detailed, the functional level kernel of the model according to the `FctBlockModel` of Eq. (5.2) for the opamp is the central part of the behavioral model of MOPA1 according to Fig. 5.13 and summarized in Eq. (6.1).

$$\texttt{BehMOPA1Model} = \{u, \vec{y}, \vec{x}, \delta, ta, \lambda, Rin, Rout\} \tag{6.1}$$



```
42 : .SUBCKT MOPA3TX     1   2   3   4   5   6
44 :   Q1 9 7 5 3 QPNPS 0.31X
45 :   XQ2 11 2 9 3 PNPS7
46 :   XQ3 12 1 9 3 PNPS8
47 :   XQ4 11 11 3 3 NPNS
48 :   XQ5 12 11 3 3 NPNS
49 :   XQ6 13 12 3 3 NPNS
50 :   Q7 3 13 4 3 QMPIS
51 :   XQ8 5 13 4 3 NPNS
52 :   XQ9 4 12 3 3 NPNS
53 :   Q13 7 6 3 3 QNPNS
54 :   Q14 6 6 3 3 QNPNS
```

```
*
* END OF MOPA2 TRANSISTOR LEVEL SUBCIRCUIT.
*
*
* START OF MOPA3 TRANSISTOR LEVEL SUBCIRCUIT.
*
.SUBCKT MOPA3TX  1   2   3   4  5    6
*               (+) (-) -VS  VO +VS  ISET
Q1 9 7 5 3 QPNPS 0.31X
```

```
element  227 from line  227( 1)
Type      : Component-R..Resistor
Name      : 9
Pins      : 1    17
Parameter : 650
Linked to : RTC
```

```
1
2
3
4
5
6
```

```
Vcc      Positive Supply Voltage Terminal
Vee      Negative Supply Voltage Termina
Vinp     Non inverting Input Terminal
Vinm     Inverting Input Terminal
Vout     Output Voltage
Voffset  Offset Voltage Terminal
```

```
Vinp    1
Vinm    2
```

Figure 6.3. Testbench adapter tool

The input to the block is thus the voltage difference at the opamp input pins denoted as `Pinp` and `Pinm`. At the output pin denoted as `Pout` the branch quantities have to be defined next. The associated code segment in the entity definition reads as

```
port ( terminal Pinp : Electrical;
       terminal Pinm : Electrical;
       terminal Pout : Electrical);
```

The definitions of u as `Vin` and $\vec{y}$ as `Vout`, `Iout` at `Pout`, respectively, in the architecture section of the model are then

```
quantity Vin across Pinp to Pinm;
quantity Vout across Iout through Pout;
quantity Y: Real;
```

Note that the high input impedance typical to opamps has been exploited directly in this model by omitting input currents (i.e., the parameter function `Rin` is not addressed explicitly). The elements of the state vector $\vec{x}$ contain the

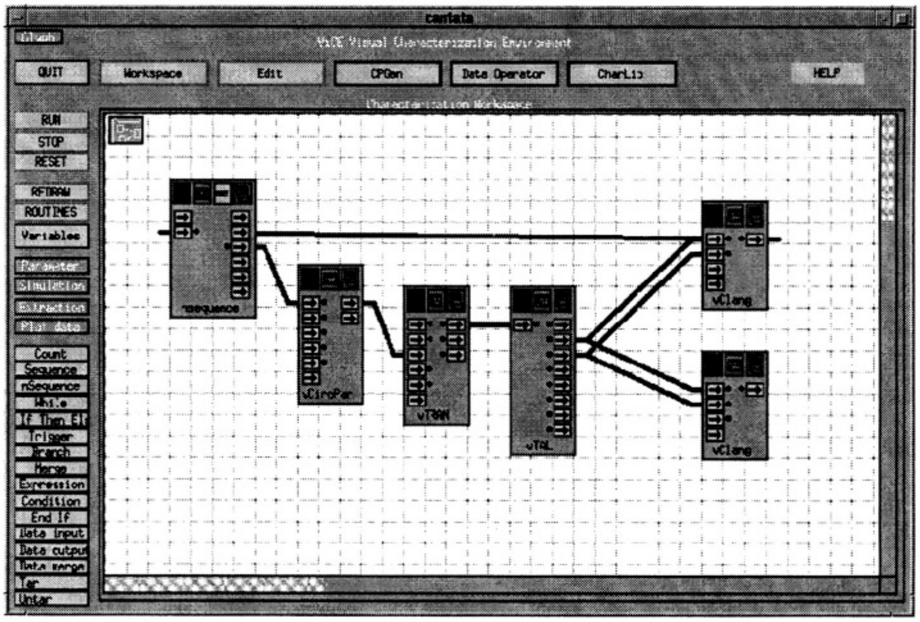


Figure 6.4. Subset of a visually specified characterization plan

actual values of the parameter functions. *Y* denotes the intermediate output quantity, which finally has to be mapped to the output branch quantities `Vout` and `Iout` by means of the parameter function `Rout`.

The transition function δ is composed from the parameter functions `DCtf`, `SlewRate`, and `TDly`. They are calibrated from the raw data produced during the execution of the characterization plan and are then compressed into methods, that is, as a set of fitted regression models selected for each of the elements of the set of segments according to the definitions for the methods library given in Chapter 5. For comparison purposes Fig. 6.5 depicts the DC transfer curve of the opamp gained from characterization and the outcome of the calibration process by means of the methods library.

The resulting method for `DCtf` is detailed in Table 6.2. The input voltage range is subdivided automatically into 5 segments, and the regression models used in these segments are polynomials up to degree 5. The differences between characterized and fitted data are less than 1% as visible from Fig. 6.5.

The parameter functions `SlewRate` and `TDly` are then calibrated in a similar way. However, the characterization plan for these functions is much more complex because it has to execute a sequence of transient simulation runs according to the plan subset depicted in Fig. 6.4 for the slew rate. Step functions featuring different amplitude values are applied to the component level model

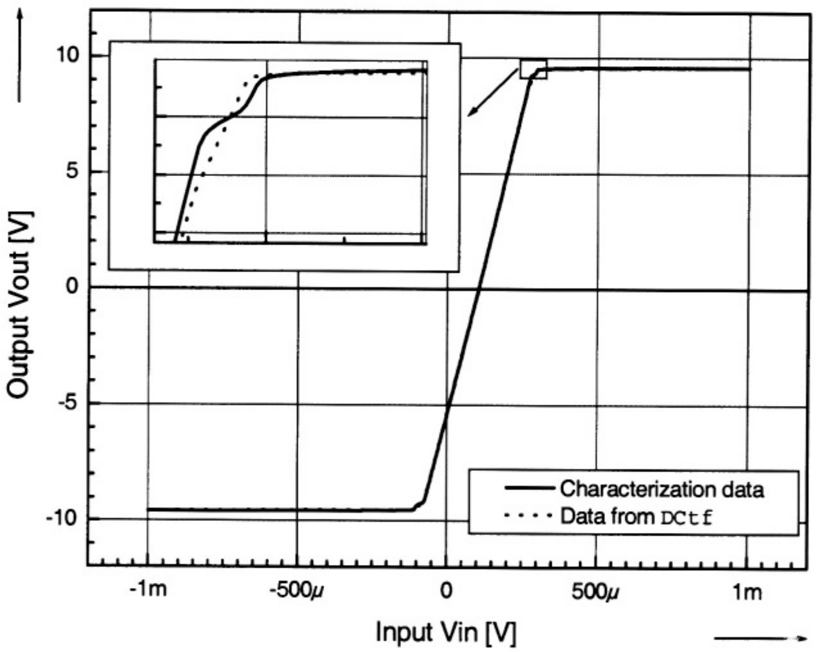


Figure 6.5. Calibration of the DC transfer function

Table 6.2. Segments and calibrated regression of the method `DCtf`

Segment		*Calibrated*
from	*to*	*Regression Model*
−0.0010	−0.0001	$68.099x - 9.506$
−0.0001	−0.00058	$4.182E8x^2 + 90908.679x - 4.544$
−0.00058	0.00009	$51141.866x - 5.387$
0.00009	0.00031	$-3.8E9x^5 + 3.E16x^4 - 9.17E12x^3$ $+1.338E9x^2 - 40894.964x - 3.024$
0.00031	0.001	$78.801x + 9.501$

instantiated in the testbench as given in Table 6.1, whereas the block response to input amplitude values around zero is crucial for accurate modeling. Fig. 6.6 shows the calibration results for the method `SlewRate`. Note the emphasis on the region around Vin=0. The representation of the resulting method in VHDL-AMS is highlighted in the following code fragment.

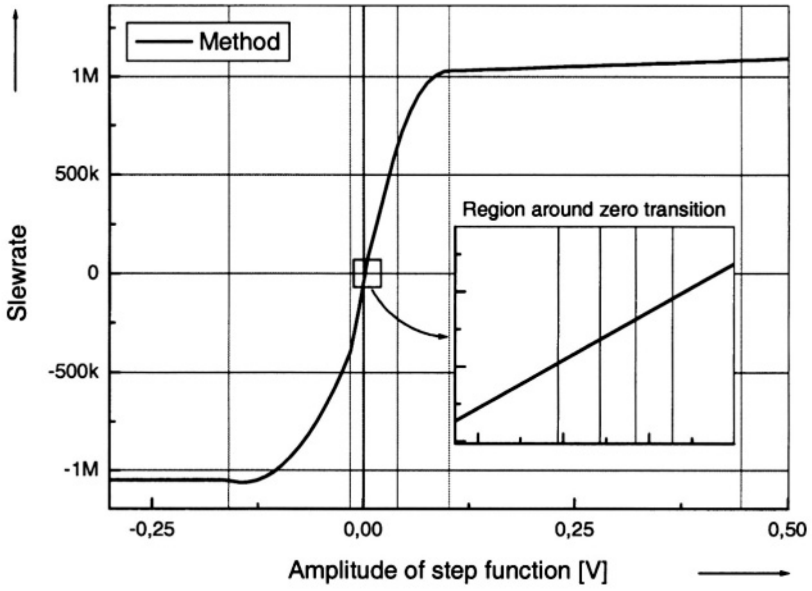


Figure 6.6. Calibration of slewing behavior

```
function SlewRate( Vin: Real) return Real is
  variable SR: Real;
begin
  if Vin < -0.1702 then
    SR := -155077.86 * Vin - 1.01541e6;
    ...
  end if;
  return SR;
end;
```

The time advance function *ta* does not have to be denoted explicitly because the resulting behavioral model is evaluated at the internally set integration time steps during transient simulation. The rate of change function *f*, however, is the key to the calculation of the dynamic behavior of the model. It is defined directly by

```
    Y'dot == SlewRate (Vin' delayed (TD));
```

The function λ has to be initialized to an appropriate DC value acting as the boundary condition for the calculation of y by integrating the rate of change function. This is done without the necessity of defining λ as an explicit function simply by setting

```
if (now < TD) use
  Y == DCtf(Vin);
```

After completion of an intermediate representation, the behavioral model should be compared to the reference model by assessing their pulse responses, for example. An instantiation of the opamp model as part of a voltage follower circuit is used for this purpose because of the high requirements this testbench puts on the accuracy of dynamic property modeling. Fig. 6.7 depicts the results of this validation step. Obviously, the dynamic behavior of the component level model of the opamp is reproduced quite accurately by the generated model.

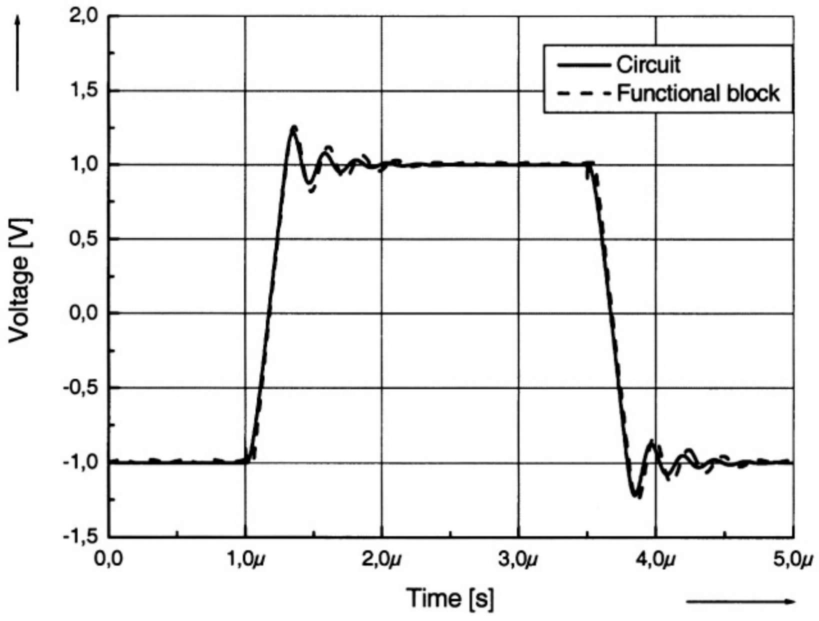


Figure 6.7. Comparison of dynamic behavior of opamp models

However, this intermediate model is not yet ready to be used as a behavioral model in different contexts because of the still missing calculation of the output branch quantities. A detailed analysis of the output resistance of the opamp for different static load conditions results in a highly nonlinear relation as depicted in Fig. 6.8. A limitation of the maximum output current of the behavioral model as found in macro models of opamps sometimes is clearly by far not sufficient to cover this complex relationship between output resistance and ohmic load condition.

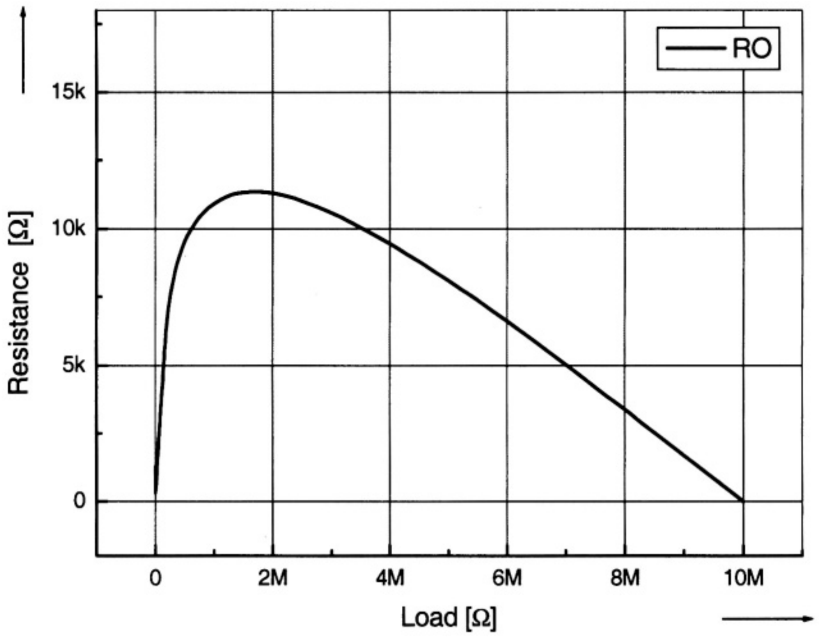


Figure 6.8. Output resistance as a function of the load condition

The resistance value of `RO` depicted in Fig. 5.13 as part of the Thevenin equivalent circuit introduced for the calculation of the output branch quantities is thus not constant and needs to be calculated from the load resistance value by the parameter function `Rout`. This function, too, has to be represented by a method gained from a calibration based on raw characterization data. The basic approach is the same as for the parameter functions of δ, but the resulting complexity of the whole characterization plan is considerably increased. At this point it becomes clear that the flexibitity and the efficiency of the tools used for characterization purposes are crucial to an exploitation of the proposed modeling approach in practice.

The now complete behavioral model at algorithmic level is next compared to both the component and the procedural level model provided by the vendor of this opamp. Fig. 6.9 depicts the results of a DC analysis of these three models for the same load condtion. Modeling of the output resistance by the parameter function `Rout` yields results which are very close to the reference. In contrast, the Boyle et al. style macro model, which is available as part of the product documentation, unveils a rather poor approximation of the output resistance of the MOPA1 opamp.

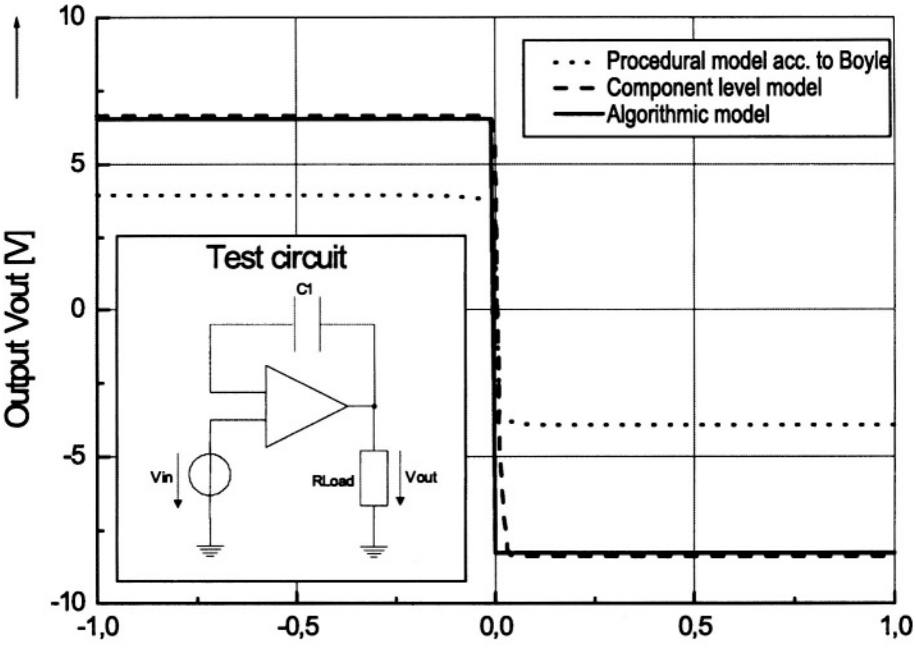


Figure 6.9. Comparison of the DC behavior of models at identical output load conditions

We now come back to the initial application example, the active filter depicted in Fig. 6.1. A model of this circuit is produced according to the objective of IP reuse, thus resulting in a structural mixed-level model description. The behavioral model of the MOPA1 opamp is instantiated four times in the netlist of the filter — together with instances of primitive components of the type capacitor and resistor, respectively, according to the schematic given in Fig. 6.1. For comparison purposes two additional versions of the filter model are produced by exploiting the available component and the procedural level models, respectively, in the opamp instances. Fig. 6.10 visualizes a comparison of results gained from the transient simulation of these three filter models for an excitation by a large swing step function.

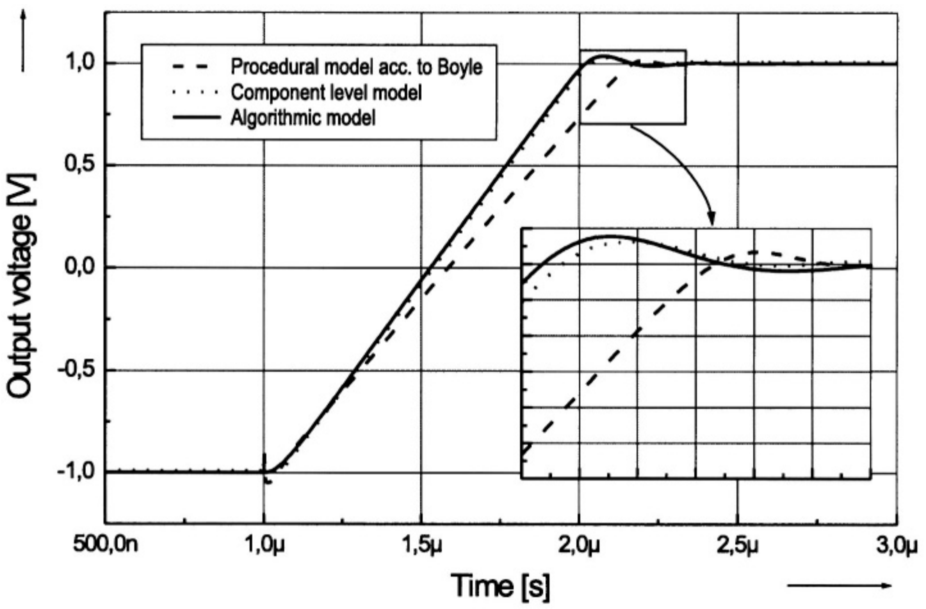


Figure 6.10. Comparison of the time domain response of different filter models

Again, the algorithmic and the component level-based filter models feature quite close analog waveform results both in terms of slewing and settling time accuracy. In contrast, the output waveform of the procedural level-based filter model differs considerably from the reference, as clearly visible from Fig. 6.10. Next to accuracy issues a review of resource utilization during simulation is of special interest to most system design engineers for an assessment of the feasibility of behavioral modeling in practice. Table 6.3 summarizes the results gained from a comparison of resource requirements of both the opamp MOPA1 and the complete active RC filter circuit, each having been modeled at three abstraction levels.

The bottom line of this application example is that analog waveform accuracy is guaranteed to a large extent by the proposed modeling approach and that, at the same time, the runtime efficiency is improved considerably when exploiting the detailed behavioral models for IP components. However, the high computational effort indispensable for accurate signal waveforms has been shifted from model execution to model characterization.

Table 6.3. Comparison of requirements in terms of simulation resources

Abstraction Level	*Circuit*	*Nodes*	*Memory [KB]*	*Time [s]*	*Speedup [Factor]*
Component	MOPA1	83	930	55	1
	Filter	327	1324	123	1
Procedural	MOPA1	15	877	18	3
	Filter	55	1098	27	4
Algorithmic	MOPA1	3	596	6	9
	Filter	9	915	12	10

6.3 A/D Converter in CMOS Technology

The second application example is a true mixed-signal circuit — an A/D converter with 6 bit resolution. This CMOS circuit is about four times larger compared to the filter example detailed in the previous section. Although still being a rather small component from a systems point of view, it poses high requirements to accurate waveform calculation. A transient analysis run of even a procedural level description of this converter takes more than 10 minutes on top of an UltraSPARC[1] server (double processor, 300 MHz) using Spectre[2] as simulation engine. These requirements stem from both the circuit complexity and the long time intervals needed for a validation of the correct function of the converter.

A typical application of the converter block within a mixed-signal system is depicted in Fig. 6.11. A time-dependent voltage waveform v may be produced by some sensor. This waveform is sampled (S/H), then subjected to a conversion to a binary representation (ADC), and finally stored in a register (Reg) as a 6 bit data word which, in turn is one of the input data items to a digital signal processor (DSP).

The conversion approach exploited in this ADC is successive approximation [Jes01], thus yielding the modular generic architecture detailed in Fig. 6.11. Each of the converter stages perform the conversion according to the simple algorithm given in Fig. 6.12 as C code. The input range of the converter is defined by the minimum and maximum voltages `Vmin` and `Vmax`, respectively. The output bit is set to 1 in case the input voltage `Vin` is greater than or equal to the mid-range value. An offset `Va` is then subtracted from `Vin` and a shift is applied to compensate for the conversion of the stage thus resulting in the final output voltage `Vout`. This output is used as the input to the next stage. The

[1] UltraSPARC is a trademark of Sun Microsystems, Inc.
[2] Spectre is a trademark of Cadence Design Systems Corp.

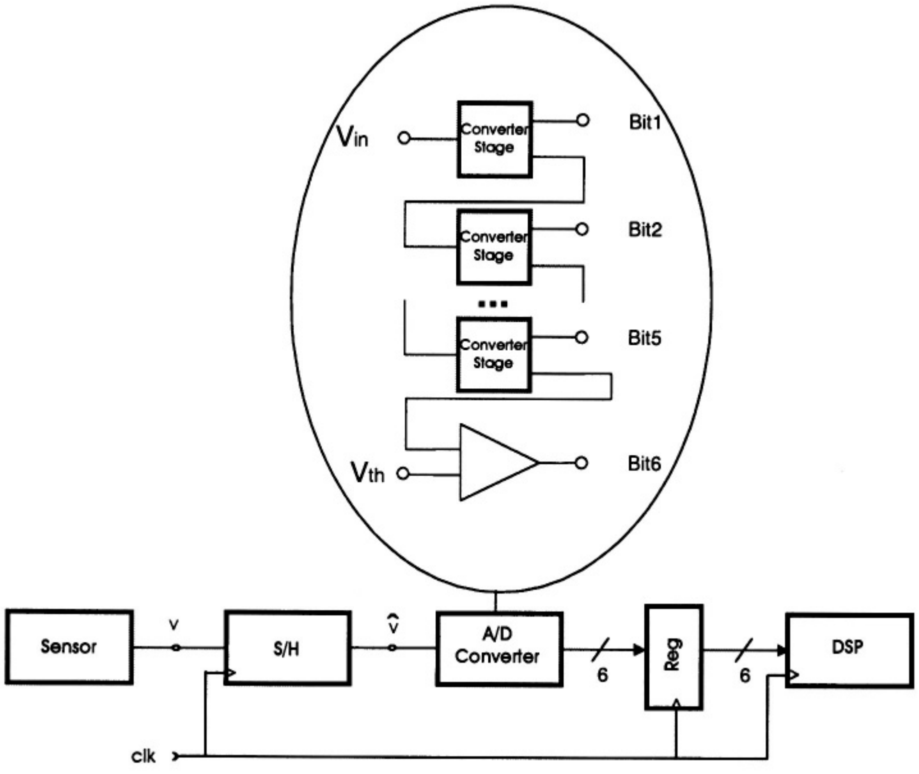


Figure 6.11. Context and generic architecture of the A/D converter

calculation proceeds sequentially from the most significant (MSB) to the least significant bit (LSB), which may be established directly from a comparison operation as depicted in Fig. 6.11.

A circuit implementation of the conversion algorithm in each but the last stage is shown in Fig. 6.13. This circuit is a modified version of a generic circuit taken from [TS91]. It consists of a comparator acting as a switch, which sets the output bit directly, of a second comparator and an analog adder, which jointly perform the subtraction and shift operations as part of the output voltage calculation.

The active components used are off-the-shelf products: two LMC6762 comparators [Sem99] and one MAX492 opamp [Max95]. A total of seven resistors complete the circuit of Fig. 6.13. The vendors of the active components provide procedural level models (i.e., SPICE macros) as part of the documentation of their products, but no circuit descriptions. This means that these vendor-provided models are the only information available to perform the characteri-

```
typedef int boolean;
void ADC_Stage(double Vmax,
               double Vmin,
               double Vin,
               boolean *Bit,
               double *Vout)
{
  double V50,Va;

  V50 = (Xmax + Vmin) * 0.5;
  if (Vin >= V50) {
    Va = (V50 - Vmin);
    Bit = 1;
  } else {
    Va = Vmin;
    Bit = 0;
  }
  Vout = 2.0 * Vin - Va;
}
```

Figure 6.12. Conversion algorithm of a stage

zation prior to behavioral model generation, and they are the only reference for model accuracy assessment. From the experiences made with macro models as detailed for the MOPA1 opamp, it is clear that one would certainly prefer component level models as an entry point to behavioral modeling, but in this case there is no choice.

When analyzing the generic architecture of the ADC in Fig. 6.11, it seems to be rather obvious to try to arrange for a behavioral model of the complete converter stage because it may be reused up to n-1 times in an n bit A/D converter. The DC transfer curve resulting from a first characterization of the converter stage of Fig. 6.13 is depicted in Fig. 6.14 for the input voltage ranging from 1V to 2V (i.e., the envisaged conversion range of the 6 bit ADC). This curve is not steady; there is a sudden change of the `Vout` value when the `BIT` value changes from 0.0 to 2.5V (i.e., from logical 0 to logical 1). In other words: This transfer curve depends not only on external input signals as usual, but on values of internal signals, too. A behavioral algorithmic level model of the converter stage, therefore, is not directly achievable from the proposed methodology.

A steady representation of the transfer curve for the entire input range may be achieved from partitioning the converter stage such that a change of the `BIT` value takes place externally to the block to be modeled. Thus, a previous internal signal of the converter stage is moved to an external pin, which generates the input signal to an appropriate subblock. Fig. 6.15 depicts how the partitioned converter stage now looks like. All reference and supply volt-

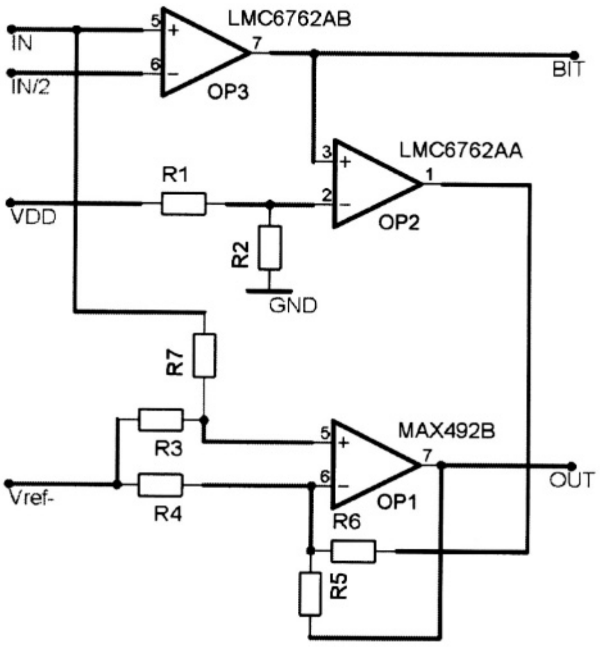


Figure 6.13. Circuit schematic of a converter stage

age connections have been removed from this figure for the sake of clarity. It consists of the switch modeled directly by the `LMC6762` macro provided by the vendor of the component and by a new subblock named `ConvSubStage` with two input and one output signals, respectively. A behavioral model of this block is being developed in the sequel according to the proposed methodology.

DC characterization of `ConvSubStage` now produces two tranfer curves over the whole range of the input signal `Vin`: `Vout0` (i.e., transfer curve `Vout (Vin)` for `BitIn=0`), and `Vout1 (Vin)` for `BitIn=1`, respectively, as depicted in Fig. 6.16. Such transfer curves are quite common for circuits featuring a hysteresis behavior. The model generation for this class of circuits takes place exactly as detailed for the converter stage: First, partitioning of the circuit, if necessary and, secondly, extraction of transfer curve data subjected to different input signal conditions from running a characterization of a lower abstraction level representation of the block to be modeled. The characterization in time domain of the partitioned subblock results in two versions each of the related parameter functions of the algorithmic model.

After completion of all, in this case, rather sophisticated characterization and calibration tasks according to the methodology of Chapter 5, it is — as

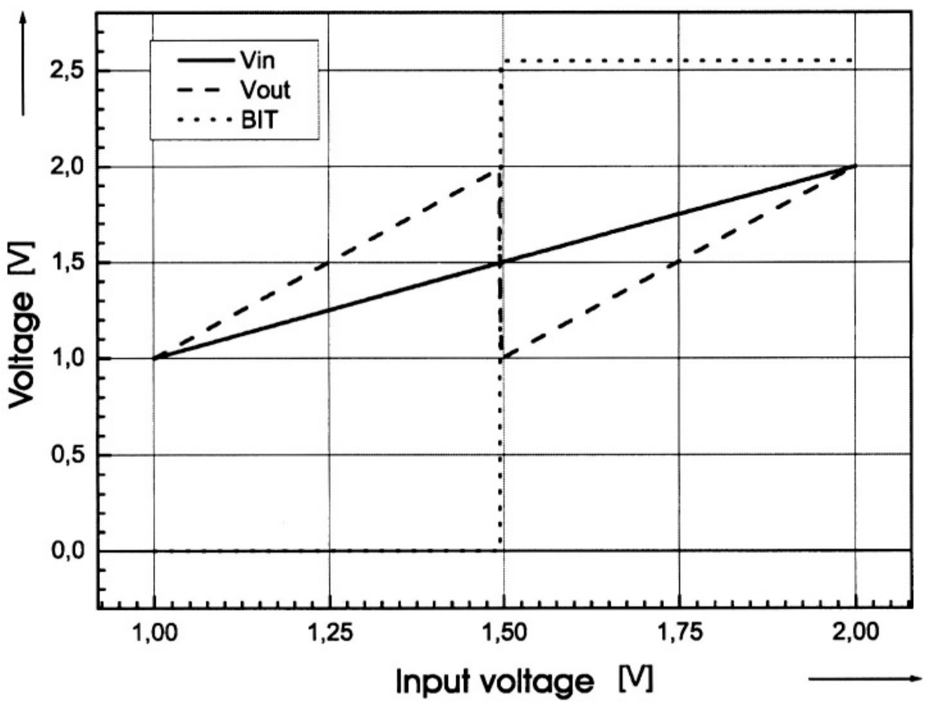


Figure 6.14. DC transfer curve of the converter stage

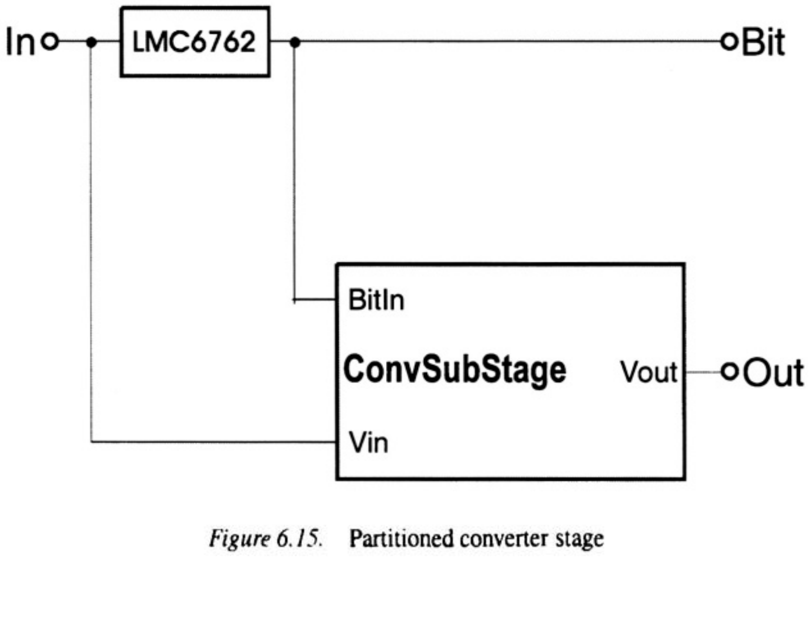


Figure 6.15. Partitioned converter stage

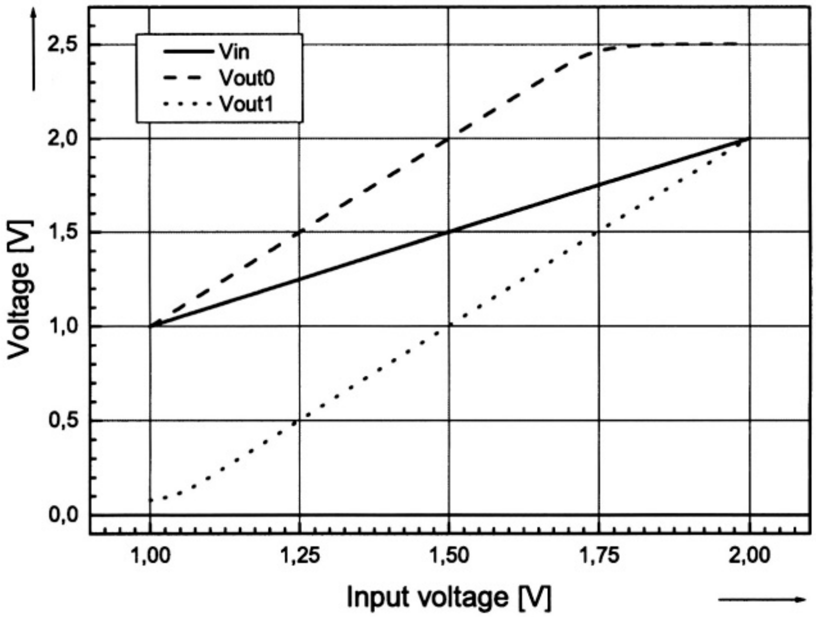


Figure 6.16. DC transfer curves of the ConvSubStage block

always — highly recommended to validate the generated behavioral model as the next step. In general, this is done by an appropriate testbench and by comparing the response of the behavioral model to waveforms gained from a simulation of the reference representation of the circuit (i.e., the procedural level model of the subblock `ConvSubStage`). For convenience, the complete model representations in VHDL-AMS of the subblock, the converter stage, and the 6 bit A/D converter, respectively, are given at the end of this section.

The output waveforms `Vout` of both models resulting from an excitation by a large-swing pulse — nearly over the whole range of the input voltage to the converter — are depicted in Fig. 6.17. Here, even the asymptotic dynamic behavior of the reference model is reproduced accurately by the generated behavioral model — without any manual fine tuning of the latter.

Next, the converter stage model — denoted as `ConverterStage` in the following VHDL-AMS code — is assembled according to Fig. 6.11 in order to run the validation for this basic block of the A/D converter. The input to the stage is a sine function — a typical test function for converters. The outputs are the `BIT` and the `Vout` signal, respectively. The comparison is performed

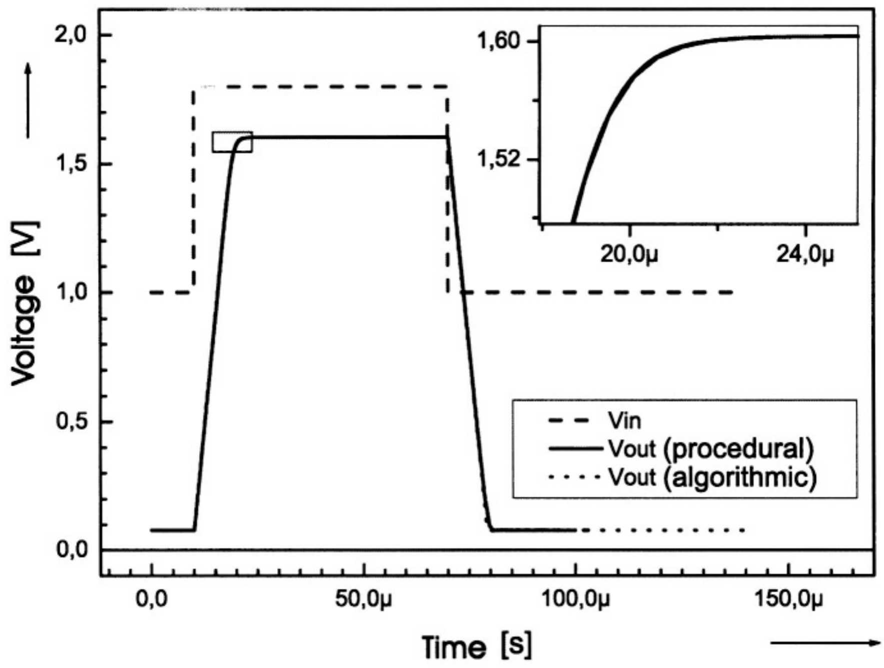


Figure 6.17. Comparison of the time domain responses of different subblock models

as before (i.e., by exercising both the compound procedural/algorithmic and the procedural level models of the stage). Fig. 6.18 visualizes the results produced by these models, which implement the conversion algorithm outlined in Fig. 6.12. Again, there is nearly no difference in the waveforms from these models, which exploit rather different modeling methods.

Finally, we construct the resulting behavioral model of the 6 bit A/D converter as a structural description, composed from mixed-level models — the `ConverterStage` instantiations — according to the generic architecture given in Fig. 6.11. Its testbench exercises a sine input signal with an amplitude covering the whole conversion range of the module (i.e., up to 2V amplitude). Fig. 6.19 depicts the input to the converter and its output data `Bit 1` to `6` of the converted amplitude value. On the right side of the figure a zoomed representation of input and output data is depicted in order to visualize the correct functionality of the converter model. As for the filter example detailed in the previous section, the simulation times for both the procedural and the mixed procedural/algorithmic models of the complete converter are compared. The results summarized in Table 6.4 indicate that the larger the circuit is modeled by behavioral models, the more significant the speed-up becomes. Note that the speed-up values in Table 6.4 are based on a comparison of procedural level

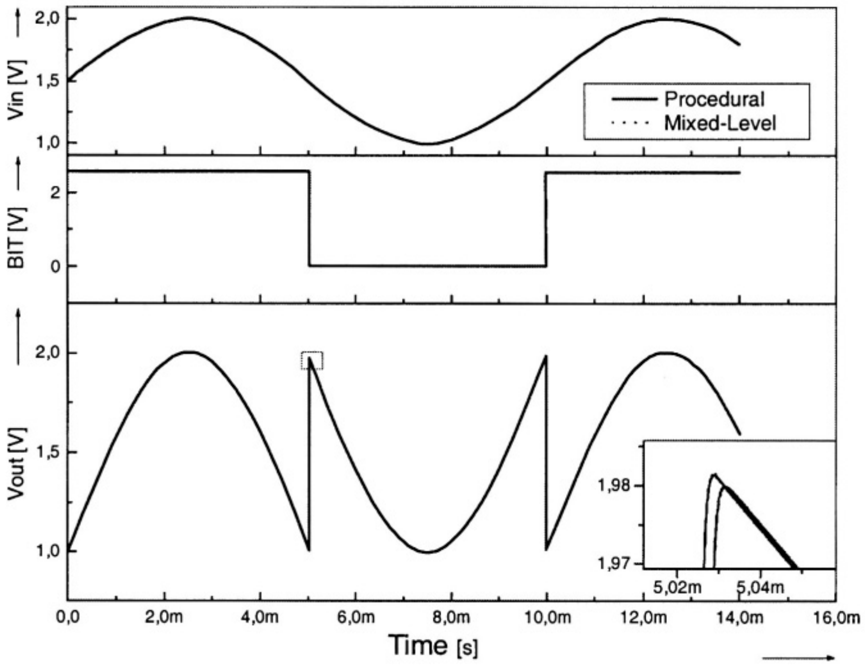


Figure 6.18. Comparison of time domain respones of different convertrer stage models

representations of a circuit which are, in practice, up to one order of magnitude faster in simulation than component level models. This achievable increase in simulation speed is a very interesting result, at least for systems engineers who are somewhat reluctant to use behavioral models for system design purposes. An additional asset is that the more complex the circuit replaced by behavioral models is, the higher the speed-up figure becomes — as visible from Table 6.4.

Table 6.4. Comparison of simulation times for different models of the A/D converter.

Module	*Procedural*	*Mixed-Level*	*Speedup*
Converter subblock	3 sec	2 sec	1.5
Converter stage	2 min 18 sec	38 sec	3.6
6 bit A/D converter	10 min 50 sec	54 sec	12

An interesting question arises about the properties of calibrated generic models of converters, which directly exploit some coded representations of conversion algorithms. This is a wide-spread approach to providing high-level and equally "accurate" models for IP blocks intended for use in system level

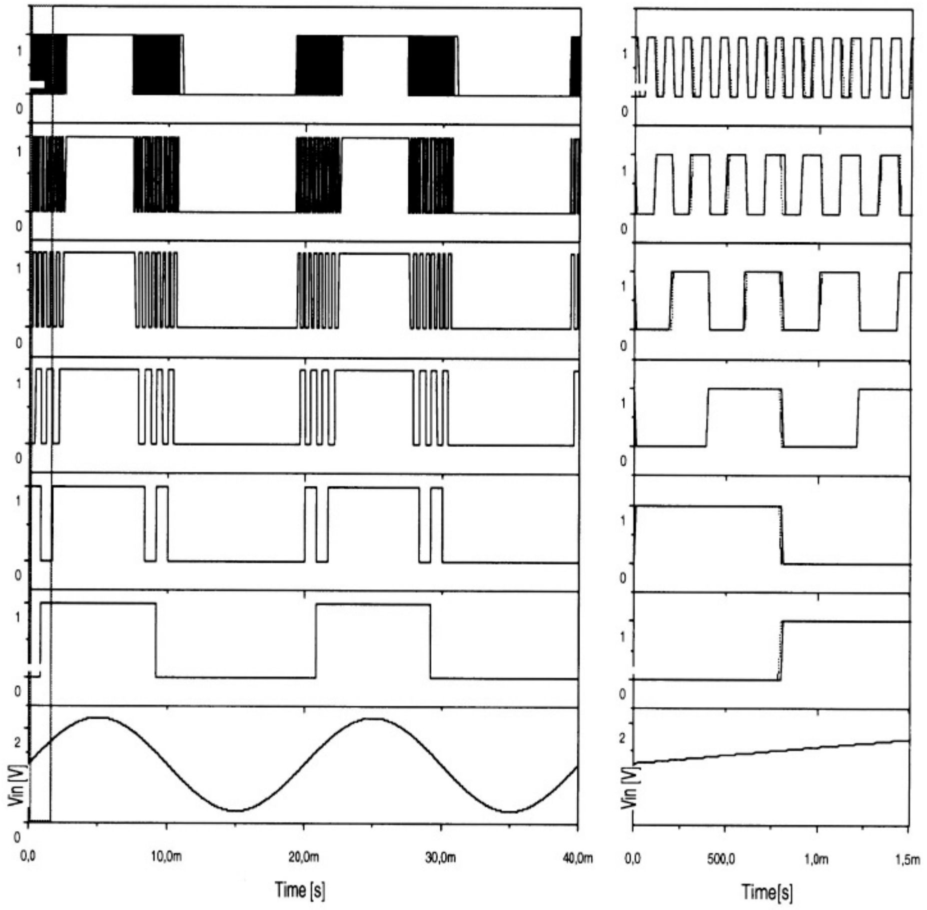


Figure 6.19. **Simulation results for mixed-level model of the A/D converter**

simulation of complex information processing systems as discussed in Chapter 5.

Let us come back to the generic model outlined in Fig. 5.2. Now, we will take the conversion time extracted from characterization results of the procedural level model of the converter stage and instantiate a structured version of the generic model outlined in Chapter 3 accordingly. The resulting code of the property mock-up, mixed-mode algorithmic model is given in Fig. 6.20.

This model operates synchronously to the system clock; it may thus be viewed as an assembly of the sample and hold and of the converter modules of Fig. 6.11. The model will now be compared to both the reference and to the mixed-level models of the ADC by exploiting the same testbench. The simulation results of these three models are almost identical when displayed

```
library IEEE
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library disciplines;
use disciplines.electromagnetic_system.all;

entity ADC is
  generic (
    MinVoltage : emf := 1.0;
    MaxVoltage : emf := 2.0;
    stageDelay : time := 60 ns;
    convDelay : time := 2 ns;
    Res : integer := 6);
  port(terminal U_in_P : Electrical;
       terminal U_in_N : Electrical;
       signal Clock, Reset : in std_logic;
       signal Data : out std_logic_vector((Res-1) downto 0));
end ADC;

architecture Analog_to_Digital of ADC is
  quantity U_In across U_In_P to U_In_N;
  constant typDelay : time := (res-1) * stageDelay + convDelay;

  function Convert_AnalogToDigital
    ( MinVoltage : real; MaxVoltage : real; Res : integer;
      U_in : real ) return Std_logic_vector is
    variable r : real;
    variable i : integer;
    variable u : unsigned(Res-1 downto 0);
    variable slv1 :std_logic_vector((Res-1) downto 0);
  begin
    r := (U_in - MinVoltage ) / (MaxVoltage - MinVoltage);
    i := integer(r * (2.0**(Res-1)));
    u := conv_unsigned(i,u'length);
    slv1 := std_logic_vector(u);
    return slv1;
  end Convert_AnalogToDigital;
begin
  P1 : process (Clock, reset)
    variable r : real;
  begin -- process P1
    if reset = '1' then
      Data <= (others => '0') after typDelay;
    elsif Clk'event and Clk = '1' then
      Data <= Convert_AnalogToDigital(MinVoltage, MaxVoltage,
                                      Res, U_in) after typDelay;
    end if;
  end process P1;
end Analog_to_Digital;
```

Figure 6.20. Calibrated generic model of the A/D converter

at a coarse resolution, as shown on the left side of Fig. 6.19. However, when zooming into these waveforms for a more detailed analysis, it becomes clear that the switching time points of the MSB are almost identical, but not for the LSB of the converter models as detailed in Fig. 6.21. The reference and the mixed-level models, again, show almost the same result, whereas the switching time point of the calibrated generic model seems to come up 'too early'. An in-depth analysis of the switching time points of all bit positions unveils that this shift in time is visible from both the mixed-level and the reference model and that it increases step by step from MSB to LSB.

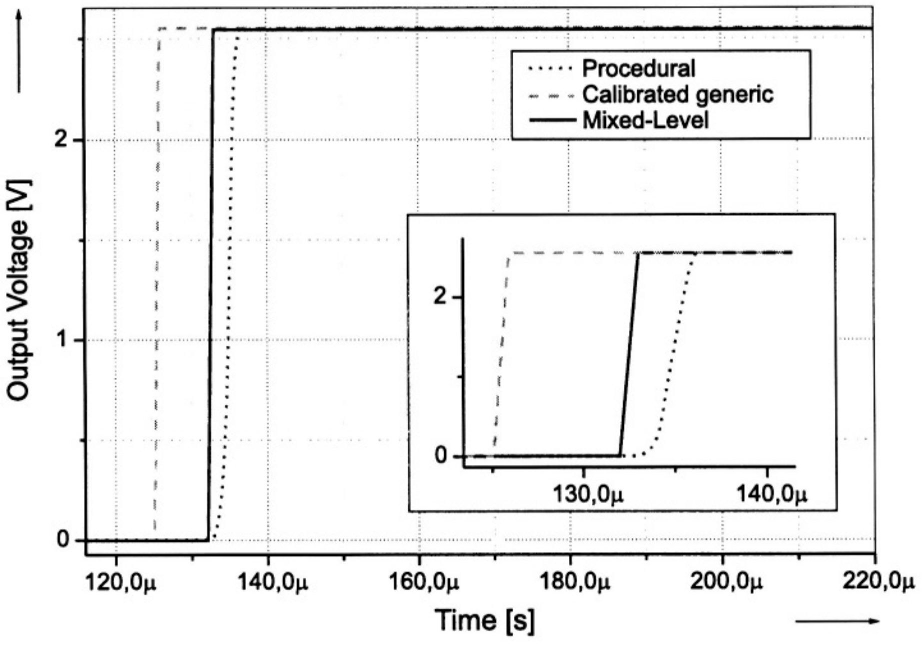


Figure 6.21. Comparison of switching time points for the LSB of the A/D converter

The reason for this strange and, at a first glance, unexpected property of the mixed-level and reference models again is the underlying analog operation of the converter. The transition of the output voltage of each converter stage forming the input to the next stage does not take place in zero time as implicitly assumed in the calibrated generic model of the converter. The analog output waveform from a stage, therefore, reaches the check point (i.e., 50% of the swing) somewhat later, which in turn results in a shift of the switching time point of the associated bit value. This intimate error of the generic model (i.e., not predicting correctly the timeshift) occurs even when the calibration of the conversion time of a stage is highly accurate. In contrast, models generated from the advocated methodology implicitly reproduce this time point shifting property quite accurately because of their intrinsic analog signal waveform calculation as clearly visible from Fig. 6.17. A delay of the LSB switching time point may result in a violation of the setup time specification of the subsequent output register as depicted in Fig. 6.11, thus causing a dynamic error condition which may affect the whole systems operation.

The next question probably asked by a systems engineer is whether this modeling error is of academic interest only. The answer is: It depends on the application context of the models. In case a system design is highly aggressive, setup time violations are likely to occur. One should be aware that there is no way to detect this situation well in advance by means of simulation runs when using calibrated digital-style stage models for successive approximation converters.

The detailed model of the 6 bit A/D converter concludes this section. It is introduced by the following top-down presentation consisting of a total of three levels of hierarchy. The listings to follow are rather lengthy because of an almost complete presentation of the VHDL-AMS code. This part may simply be skipped by those readers who are not interested in coding details.

Fig. 6.22 depicts the code of the top level entity `ADC6` and its architecture definition according to the structural representation depicted in Fig. 6.11. It consists of a total of five instances of `ConverterStage` and one instance of the commercially available macro model of the standard comparator component `LMC6762`, respectively.

Next, the coding of the `ConverterStage` is given in Fig. 6.23. This forms the intermediate level of hierarchy according to the partitioning step presented in Fig. 6.15. It consists mainly of a comparator and a `ConvSubStage` instance each. The latter is the behavioral model already detailed, thus resulting in a mixed-level representation of `ConverterStage`.

Finally, the model of `ConvSubStage` (i.e., the lowest level of hierachy) as generated by the proposed modeling methodology and automatically mapped to VHDL-AMS is given in Fig. 6.23. It consists mainly of the calibrated parameter functions according to the `BlockModel` definition. In this special case the `TDly` function value denoted `TD` is passed by a generic parameter. This is because the characterization and the calibration process resulted practically in a constant function value. The coded calibration results of the parameter functions are documented in Fig. 6.24 in terms of fitting accuracy, number of segments, and maximum degree of polynomials used as regression models. These function definitions account by far for most of the model text. Note how the calculation of the effective input signal value denoted as `VI` is being defined right at the beginning of the executable section of the model.

The implemented functionality of a block modeled in the proposed way cannot be deduced from an inspection of the code as depicted in Fig. 6.24.

```
------------------------------------------------------
-- 6Bit ADC
-- Combining  5 Converter-Stages and one
-- Comparator results in a 6 Bit ADC
------------------------------------------------------
library disciplines;
use disciplines.electromagnetic_system.all;
entity ADC6 is
generic(
   vlow : REAL := 1.0;
   vhigh : REAL := 2.0);

port(
   terminal  p1, pgnd : Electrical;
   signal bp : out bit_vector(5 downto 0));
end entity ADC6;


library disciplines;
use disciplines.electromagnetic_system.all;
use WORK.all;
architecture struct of ADC6  is
   signal bit    : bit_vector(bp'range);
   terminal pout1, pout2, pout3, pout4, pout5 : Electrical;
   constant vthresh : REAL := (vhigh + vlow) / 2.0;

begin
   cs1       : entity ConverterStage(struct) generic map(vthresh)
        port map(p1, pout1, pgnd, bit(0));
   cs2       : entity ConverterStage(struct) generic map(vthresh)
       port map(pout1, pout2, pgnd, bit(1));
   cs3       : entity ConverterStage(struct) generic map(vthresh)
        port map(pout2, pout3, pgnd, bit(2));
   cs4       : entity ConverterStage(struct) generic map(vthresh)
        port map(pout3, pout4, pgnd, bit(3));
   cs5       : entity ConverterStage(struct) generic map(vthresh)
        port map(pout4, pout5, pgnd, bit(4));
   comp1     : entity LMC6762AB(behavioral) generic map(vthresh)
        port map(pout5, pgnd, bit(5));
   bp <= bit;
end architecture struct;
```

Figure 6.22. Top level model of the A/D converter

This drawback, however, turns into a considerable advantage when it comes to disclosing models of IP products.

```
-------------------------------------------------------------
-- Converter stage, consists  of comparator, switch,
-- and adder/amplifier
-------------------------------------------------------------
library disciplines;
use disciplines.electromagnetic_system.all;
entity ConverterStage is
generic (
   vthresh: REAL := 1.5);

port(
   terminal  p1, pout, pgnd   : Electrical;
   signal pbit  : out bit);

end entity ConverterStage;


library disciplines;
use disciplines.electromagnetic_system.all;
use WORK.all;
architecture struct of ConverterStage  is

begin
   comp1    : entity LMC6762AB(Behavioral)
         generic map(vthresh)
         port map(p1, pgnd, pbit);
   stage1   : entity generic_ConvSubStage(Behavioral)
         port map(p1, pout, pbit);
end architecture struct;
```

Figure 6.23. Model of the converter stage

```
library ieee;
use ieee.math_real.all;
library disciplines;
use disciplines.electromagnetic_system.all;

entity generic_ConvSubStage is
   generic ( V_out_min : emf  := -9.6;
             V_out_max : emf  :=  9.6;
             TD        : real := 5.8e-8);
   port( terminal  Uin, Uout   : Electrical;
         signal Bit    : BOOLEAN);
end generic_ConvSubStage;

architecture behavioral of generic_ConvSubStage is
-- Begin *******  Method slewrate0  ********
-- Number of sections: 5 Maximum Error: 2.5% Maximum Degree: 4
function slewrate0(Vdiff: real) return real is
   variable sign, V_diff, SR: real;
begin
   if Vdiff < 0.0 then
      sign := -1.0;
      V_diff := -1.0 * Vdiff;
   else
      V_diff := Vdiff;
      sign := 1.0;
   end if;

   if V_diff <= 2.96e-6 then
      SR := 0.0;
   elsif V_diff <= 1.419e-1 then
      SR := -4.4832957e3 - 1.1722595e6 * V_diff
           + 1.7897751e9 * V_diff ** 4.0e0
           - 5.8280225e8 * V_diff ** 3.0e0
           + 6.0415514e7 * V_diff ** 2.0e0;
   elsif V_diff <= 3.9859e-1 then
      SR := 3.1599691e4 + 6.3580799e5 * V_diff
           - 7.7975484e5 * V_diff ** 2.0e0;
   elsif V_diff <= 1.61786e0 then
      SR := 1.5973821e5 + 3.5260018e3 * V_diff;
   else     SR := 165442.786;
   end if;
   return sign * SR;
end;
-- End *******  Method slewrate0  ********
```

Figure 6.24. Behavioral model of ConvSubStage

```
-- Begin *******  Method slewrate1  ********
-- Number of sections: 9 Maximum Error: 2.5% Maximum Degree: 3
function slewrate1(Vdiff: real) return real is
   variable SR, V_diff, sign: real;
begin
   if Vdiff < 0.0 then
      sign := -1.0;
      V_diff := -1.0 * Vdiff;
   else
      V_diff := Vdiff;
      sign := 1.0;
   end if;

   if V_diff <= 2.96e-6 then
      SR := 0.0;

   elsif V_diff <= 1.36e-3 then

      SR := -1.1791821e2 + 1.2488487e6 * V_diff;
   elsif V_diff <= 1.772e-2 then

      SR := -1.1376556e2 + 1.2457953e6 * V_diff;
   elsif V_diff <= 3.68e-2 then

      SR := 1.7868687e3 + 1.138536e6 * V_diff;
   elsif V_diff <= 1.1176e-1 then

      SR := -3.5795385e1 + 1.3051697e6 * V_diff
            -3.1821959e6 * V_diff ** 2.0e0;
   elsif V_diff <= 2.5896e-1 then

      SR := 2.5414163e4 -1.5785964e6 * V_diff ** 2.0e0
            + 9.008659e5 * V_diff;
   elsif V_diff <= 5.9152e-1 then

      SR := 8.5254322e4 + 4.443516e5 * V_diff
            + 5.3807284e5 * V_diff ** 3.0e0
            - 8.4308599e5 * V_diff ** 2.0e0;
   elsif V_diff <= 1.089e0 then

      SR := 1.6416098e5 + 4.863889e2 * V_diff;
   else    SR := 165442.786;
   end if;
   return sign * SR;
end;
-- End *******  Method slewrate1  ********
```

Figure 6.24 (continued). Behavioral model of ConvSubStage

```
-- Number of sections: 3, Maximum Error: 2%, Maximum Degree: 3
function VOUTDC0(Vin: real) return real is
   variable DC0: real;
begin
   if Vin <= 1.646 then
      DC0 := 2.0 * Vin - 1.0004;
   elsif (1.646 < Vin and Vin <= 1.79 ) then
      DC0 := -25.0895 * Vin ** 3.0e0 + 122.245 * Vin ** 2.0e0
             - 196.394 * Vin + 106.242;
   else
      DC0 := -0.628326 * Vin ** 2.0e0
             + 2.45802 * Vin + 0.0998593;
   end if;
   return DC0;
end;
-- End *******  Method VOUTDC0    ********

-- Begin *******  Method VOUTDC1  ********
-- Number of sections: 3, Maximum Error: 2%, Maximum Degree: 3
function VOUTDC1(Vin: real) return real is
   variable DC1: real;
begin
   if Vin <= 1.052  then
      DC1 := 52.7017 * Vin ** 3.0e0 - 154.027 * Vin ** 2.0e0
             + 150.414 * Vin - 49.0104;
   elsif Vin <= 1.116 then
      DC1 := -76.3004 * Vin ** 3.0e0 + 253.087 * Vin ** 2.0e0
             - 277.849 * Vin + 101.159;
   else
      DC1 := 1.99983 * Vin - 1.99626;
   end if;
   return DC1;
end;
-- End *******  Method VOUTDC1  ********
-- input
quantity U_in across
   Uin;
-- output
quantity
   U_out across
   U_out_I through
   Uout;
quantity VI : emf;
begin
```

Figure 6.24 (continued). Behavioral model of ConvSubStage

```
   -- calculation of effective input value
   if Bit use
      VI == VOUTDC1( U_in ) - U_out;
   else
      VI == VOUTDC0( U_in ) - U_out;
   end use;
   -- calculation of the output voltage
   -- limit voltage to maximum / minimum values
   -- through current is not limited, set by simulator
   if (now < TD and Bit) use
      U_out == VOUTDC1( U_in );
   elsif (now < TD) use
      U_out == VOUTDC0( U_in );
   elsif (U_out >= V_out_max) and
      (slewrate0( U_in'Delayed( TD )) > 0.0) use
      U_out == V_out_max;
   elsif (U_out <= V_out_min) and
      (slewrate0( U_in'Delayed( TD )) < 0.0) use
      U_out == V_out_min;
   else
      U_out'dot == slewrate1( VI'Delayed( TD ) );
   end use;
   -- telling the simulator engine about discontinouities
   break when now=TD;
   break when U_out'above(V_out_max);
   break when not(U_out'above(V_out_min));
end behavioral;
```

Figure 6.24 (continued). Behavioral model of ConvSubStage

6.4 Conclusions

Abstract representations of behavior — modeling of physical processes taking place in the real world always exploits some more or less obvious simplifications and abstractions — may be derived in a variety of ways. There are vivid discussions in various scientific communities which address modeling issues in terms of the interrelationship of structural and behavioral domains and in terms of model quality. These fundamental questions demand reflection, and sensible answers are to be elaborated even for a restricted area such as mixed-signal integrated circuit design.

Model engineering of mixed-signal circuits and systems is a rather complex and a multi-facetted problem because this class of circuits and systems is part of two all but disjoint worlds, namely time-continuous and time-discrete signal processing. The problem may, however, be attacked by first searching for a common foundation of these almost disjoint worlds; starting from this solid ground, abstraction hierarchies then can be defined. Further, modeling methods related to one or more abstraction levels may be elaborated, and their figures of merit, possible drawbacks as well as application areas have to be identified. This requires a thorough consideration of modeling concepts, calibration methods, abstraction hierarchies, and model representation issues, which by no means are independent from each other, however.

At first glance, these mainly academic problems have a considerable impact on the everyday work of system and integrated circuit design engineers. Time to market pressure, tight design time frames, and cost constraints on the final product urge practitioners to adopt their design procedures accordingly, which essentially results in a computer-aided overall design flow, extensive exploitation of available off-the-shelf components, and IP products. Design variants and trade-offs, therefore, are more and more relying on simulation as the central validation method for design decisions. Consequently, adequate models of systems, subsystems, functional blocks, and components at different levels of detail have to be present in order to appropriately validate design variants and decisions. Interrelationships of model representations and modeling methods have to be considered, too. A modeler or a model user should always be aware that the expressivity of a HDL is a necessary, but not a sufficient condition for model engineering.

When introducing especially high-level models of all these objects into the overall design flow, one is faced with the need to establish a refinement process for the top-down, an abstraction process for the bottom-up approach to traversing the abstraction hierachy, and a viable method to combine all into a comprehensive meet-in-the-middle design approach. A replacement of lower-level models with higher-level ones and vice versa can only be accomplished if the models replacing each other are *equivalent*; in this manner significant inaccuracies can be avoided. Equivalency, however, needs a consistent definition especially when moving from the functional model class to the behavioral class as outlined in this book. But even when interchanging models at different abstraction levels, which are part of the same class, an appropriate metric is required for a quantification of the equivalence property. Models of design objects featuring differently detailed representations of their behavior simply cannot produce identical outputs in terms of properties and especially in terms of signal waveforms for all feasible model instantiation contexts. An introduction of abstraction levels would otherwise make no sense at all. Behavioral equivalence can unambiguously be attributed only if both block properties and output waveforms resulting from executing models denoted at different abstraction levels fit into predefined ranges of properties and waveforms, respectively, which represent the metric of quality. Equivalence assessment of high-level behavioral models instantiated in different contexts is still an open research area.

A comprehensive methodology for model generation for all analog and mixed-signal circuit classes and for all operation domains of these circuits is not yet documented in open literature, perhaps it cannot be achieved at all. Known approaches and methods to this problem are constrained in one way or the other. There is no Golden Rule available, which can be used adequately as a foundation for model generation with accurate and equally efficient models for mixed-signal circuits in mind — except, maybe, the following one:

Digital signals within and electrical properties of mixed-signal circuits are always results of an abstraction operation from physical time and value continuous signal waveforms.

References

[AB95] B. A. Antao and A. J. Brodersen. Behavioral simulation for analog system design verification. *IEEE Trans. VLSI Systems,* 3(3):417–29, 1995.

[Ana95] ANACAD EES GmbH, Ulm. *SimPilot Refernce Manual,* 1995.

[Ana97] Analogy Inc. *SaberDesigner Applications Reference,* 4.2 edition, 1997.

[Ant98] Antrim Design Systems. The characterization and behavioral model generation of analog intellectual property. White paper, Antrim Design Systems, Inc., 1998. http://www.antrim.com/.

[Ant99] Antrim Design Systems. Advanced techniques for the simulation of mixed-signal integrated circuits. White paper, Antrim Design Systems, Inc., 1999. http://www.antrim.com/.

[Arm89] J. Armstrong. *Chip-Level Modeling with VHDL.* Prentice-Hall, Englewood Cliffs, 1989.

[Bau94] J. Baumgart. Implementation of an object-oriented data model for the visualization of asynchronous communication protocols. Master's thesis, Darmstadt Univ. of Technology, 1994. in German.

[BCP74] R. Boyle, M. Cohn, and O. Pedersen. Macromodeling of integrated circuit operational amplifiers. *IEEE J. Solid-Slate Circ.,* pages 353–64, 1974.

[BCP89] K. E. Brenan, S. L. Campell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* Society for Industrial and Applied Mathematics, Philadelphia, 1989.

[BD87] G. E. P. Box and N. R. Draper. *Empirical Model-Building and Response Surfaces.* John Wiley and Sons, New York, 1987.

[BHW99] W. Boßung, S. A. Huss, and L. Wehmeyer. A graphical-interactive tool for the configuration and the runtime control of a parallel machine. In *5. ITG/GI Workshop Analog '99,* 1999. in German.

[BLR95] J.-M. Bergé, O. Levia, and J. Rouillard, editors. *Modeling in Analog Design.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1995.

[Box78] G. E. P. Box. *Statistics for Experimenters.* John Wiley and Sons, New York, 1978.

[Cam98] P. Campisi. Analog CMOS library for analog synthesis systems. Technical report, University of Cincinnatti, 1998. http://www.ececs.uc.edu/~ddel/projects/vase/library/.

[CB97] E. Christen and K. Bahalar. *VHDL 1076.1: Analog and Mixed-Signal Extensions to VHDL,* chapter 2. Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[CC92] J. A. Conelly and P. Choi. *Macromodeling with Spice.* Prentice-Hall, Englewood Cliffs, 1992.

[Cel91] F. E. Cellier. *Continuous System Modeling.* Springer-Verlag, Berlin/Heidelberg/New York, 1991.

[Cir91] M. A. Cirit. Characterizing a VLSI standard cell library. In *IEEE Custom Integrated Circuits Conf.,* pages 25.7.1–25.7.4, 1991.

[CL75] L. O. Chua and P. M. Lin. *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques.* Prentice-Hall, Englewood Cliffs, 1975.

[CSLS00] C. Clauss, A. Schneider, T. Leitner, and P. Schwarz. Modeling of electrical circuits with Modelica. In *Proc. Modelica Workshop,* 2000.

[CY94] G. Casinovi and I.-M. Yang. Multi-level sitmulaiton of large analog systems containing behavioral models. *IEEE Trans. on Computer Aided Design,* 13(11): 1391–9, 1994.

[DG98] B. De Smedt and G. Gielen. Nonlinear behavioral modeling and phase noise evaluation in phase locked loops. In *IEEE Custom Integrated Circuits Conf.,* pages 53–6, Santa Clara, 1998.

[DJS95] E. Driouk, O. Jarov, and A. Sukhodolsky. On software development to support statistical simulation of analogue circuits. *In EDAC,* pages 539–43, 1995.

[DLG+98] G. Debyser, F. Leyn, G. Gielen, W. Sansen, and M. Styblinski. Efficient statistical analog IC design using symbolic methods. In *IEEE Int. Symp. on Circuits and Systems,* pages 21–4, Monterey, 1998.

[DP98] J. Dabrowski and A. Pulka. Discrete approach to PWL analog modeling in VHDL environment. *Analog Integrated Circuits and Signal Processing,* 16:91–9, 1998.

[EZJ+00] J. Eckmüller, S. Zizala, R. Jancke, P. Trappe, and P. Schwarz. A methodology for the modeling of mixed-signal blocks. In *Workshop VHDL-VMS,* Reutlingen, 2000. in German.

[FVG00] K. Francken, P. Vancorenland, and G. Gielen. DAISY: A simulation-based high-level synthesis tool for delta-sigma modulators. In *IEEE/ACM Int. Conf. on Computer-Aided Design,* pages 188–92, San Jose, 2000.

[GCP01] D. Gibson, H. Carter, and C. Purdy. The use of hardware description languages in the development of micromechanical systems. *Analog Integrated Circuits and Signal Processing,* 2(28): 173–81, 2001.

[GH93] M. Goedecke and S. A. Huss. Ein Konzept für eine visuelle Umgebung zur Charakterisierung gemischt digital/analoger Schaltungen. In *2. ITG/GME-Workshop Entwicklung und Analogschaltungen mit CAE-Methoden,* pages 21–6, 1993.

[GH94] M. Goedecke and S. A. Huss. Ein interaktives System zur Charakterisierung gemischt digital/analoger Schaltungen. In *3. ITG/GME-Workshop Entwicklung und Analogschaltungen mit CAE-Methoden,* pages 21–6, 1994.

[Goe01] M. Goedecke. *Methods for describing characterization plans for the evaluation of analog/digital circuits.* PhD thesis, Darmstadt Univ. of Technology, 2001. in German.

[GS91] G. Gielen and W. Sansen. *Symbolic Analysis for Automated Design of Analog Integrated Circuits.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1991.

[GV94] D. D. Gajski and F. Vahid. *Specification and Design of Embedded Systems.* Prentice-Hall, Englewood Cliffs, 1994.

[GW98] Ch. Grimm and K. Waldschmidt. Repartitioning and technology mapping of electronic hybrid systems. In *IEEE/ACM DATE Conf.,* Paris, 1998.

[GWS94] G. Gielen, P. Wambacq, and W. Sansen. Is there a future for symbolic analysis? In *IEEE 1SCAS,* page 453, London, 1994.

[Ham95] H. Hamad. *Conceptualization of abstract behavioral models of analog functional blocks.* PhD thesis, Darmstadt Univ. of Technology, 1995. in German.

[HG95] S. A. Huss and M. Goedecke. Characterization of integrated circuits. *Mikroelektronik und Mikrosystemtechnik,* 9(4): 14–8, 1995. in German.

[HGT91] S. A. Huss, M. Gerbershagen, and G. Tränkle Automatic performance characterization of analog functional blocks. *Analog Integrated Circuits and Signal Processing,* 1:277–86, 1991.

[HRB73] C. W. Ho, A. E. Ruehli, and P. A. Brennan. The modified nodal approach to network analysis. *IEEE Trans. on Circuits and Systems,* 22(6):504–8, 1973.

[IEE99] IEEE Computer Society. *IEEE Standard VHDL Analog and Mixed-Signal Extensions,* 1999. IEEE Std. 1076–1999.

[Jes01] P. G. A. Jespers. *Integrated Conveners.* Oxford University Press, Oxford, 2001.

[JRS91] Y.-C. Ju, V. Rao, and R. Saleh. Consistency checking and optimization of macromodels. *IEEE Trans. on Computer-Aided Design,* 10(8):957–67, 1991.

[Kas00] M. Kasper. *Mikrosystemenrwurf.* Springer-Verlag, Berlin, 2000.

[KC87] A. I. Khuri and J. A. Cornell. *Response Surfaces: Designs and Analyses.* Marcel Decker, New York, 1987.

[Klu01] St. Klupsch. Design, integration and validation of heterogenous systems. In *IEEE Symp. on Quality Electronic Design,* San José, 2001.

[LAR+97] M. Laudon, Ch. Amacker, Ph. Renaud, A. Vachoux, B. Romanowicz, Y. Ansel, and G. Schröpfer. VHDL-1076.1 modeling examples for microsystem simulation in analog and mixed-signal hardware description languages. In *Analog and Mixed-Signal Hardware Description Languages,* Current Issues in Electronic Modeling, chapter 7, pages 131–54. Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[LDGS97] F. Leyn, W. Daems, G. Gielen, and W. Sansen. A behavioral signal path modeling methodology for qualitative insight in and efficient sizing of CMOS opamps. In *IEEE/ACM Int. Conf. on Computer-Aided Design,* pages 374–81, San Jose, 1997. IEEE/ACM.

[LMN95] F. Lémery, J.-P. Morin, and E. Nercessian. An interactive environment for analog characterization and behavioral modeling. In *Europ. Solid State Circuits Conf.,* pages 314–37, Lille, 1995.

[LMO83] M. Landry, J. L. Malouin, and M. Oral. Model validation in operations research. *Europ. J. of Operations Research,* 14(3):207–20, 1983.

[LSJ96] J.-Y. Lin, W.-Z. Shen, and J.-Y. Jou. A power modeling and characterization method for the CMOS standard cell library. In *IEEE/ACM Int. Conf. on Computer-Aided Design,* pages 400–4, 1996.

[Max95] Maxim Integrated Circuits. *MAXIM Single/Dual/Quad, Micropower, Single-Supply Rail-to-Rail Op Amps,* 1995. http://www.maxim-ic.com/.

[Men99] Mentor Graphics Corporation. *ADVance MS User's Manual,* 1999.

[MF95] H. A. Mantooth and M. Fiegenbaum. *Modeling with an Analog Hardware Description Language.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1995.

[MK90] K. Milzner and F. Krohm. Knowledge-based simulation environment. In *IEEE Custom Integrated Circuits Conf.,* pages 325–8, 1990.

[Mon91] C. Montgomery. *Design and Analysis of Experiments.* John Wiley and Sons, New York, third edition, 1991.

[NHM94] P. Nussbaum, M. Hinners, and L. Menevaut. SimBoy: An analog simulator interface for automated datasheet extraction. *In EuroASIC,* pages 37–41, 1994.

[Nut97] UC Berkeley. *Nutmeg Reference Manual,* 1997.

[OHLR00] J. Oudinot, Ch. Hui-Bon-Hoa, F. Lemery, and A. Rossi. Validation of a new methodology using VLSI-AMS on a hard-disk drive design. In *ECSI Forum on Design Languages,* pages 141–9, Tübingen, 2000.

[Pra91] H. Praehofer. *System-theoretic Foundations of Combined Discrete-Continuous System Simulation.* PhD thesis, Johannes Kepler Universität Linz, 1991.

[Rat83] D. Ratkowsky. *Nonlinear Regression Modeling,* volume 48 of *Statistics.* Marcel Decker, New York, 1983.

[Ray90] Semiconductor Division, Raytheon Comp. *RLA linear macrocell array breadboarding kit,* 1990.

[RH98] R. Rosenberger and S. A. Huss. A systems theoretic approach to behavioral modeling and simulation of analog functional blocks. In *IEEE/ACM DATE Conf.,* pages 721–8, Paris, 1998.

[Rob99] S. Robinson. Simulation, verification, validation and confidence: A tutorial. *Trans. of Soc. of Computer Simulation,* 16(2):63–9, 1999.

[Rom98] B. F. Romanowicz. *Methodology for the Modeling and Simulation of Microsystems.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1998.

[Ros94a] R. Rosenberger. Design and implementation of a methods library for behavioral model calibration. Master's thesis, Darmstadt Univ. of Technology, 1994. in German.

[Ros94b] R. Rosenberger. Integration of Mathematica into CLANG. Technical report, Darmstadt Univ. of Technology, 1994. in German.

[Ros01] R. Rosenberger. *A systems-theoretic approach to behavioral model generation in mixed-signal domain.* PhD thesis, Darmstadt Univ. of Technology, 2001. in German.

[RW91] J. R. Rasure and C. S. Williams. An interpreted data flow visual language and software development environment. *Visual Languages and Computing,* 2:217–46, 1991.

[Sem99] National Semiconductor. *LMC6762 Dual MicroPower Rail-To-Rail Input CMOS Comparator with Push-Pull Output.* National Semiconductor, Product Folder, 1999. http://www.national.com/.

[SG94] G. Strube and H. Gräb. ASIS: Automatic simulator control. In *3. ITG/GME-iWorkshop Entwicklung und Analogschaltungen mit CAE-Methoden,* pages 297–302, 1994. in German.

[SJN94] R. Saleh, S.-J. Jou, and A. R. Newton. *Mixed-Mode Simulation andAnalog Multilevel Simulation.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1994.

[SSS97] G. Schatzberger, H. Senn, and P. Söser. Automatisierte Charakterisierung von Operationsverstärkern mit Hilfe von Simulationswerkzeugen. In 5. *GI/ITG/GMM Workshop Methoden des Entwurfs und der Verifikation digitaler Systeme,* pages 337–43, 1997.

[SV95] C. J. R. Shi and A. Vachoux. *VHDL-A Design Objectives and Rationale,* volume 2 of *Current Issues in Electronic Modeling (CIEM),* chapter 1, pages 1–30. Kluwer Academic Publishers, Boston/Dordrecht/London, 1995.

[Tho90] J. U. Thoma. *Simulation by Bondgraphs.* Springer-Verlag, Berlin/Heidelberg/New York, 1990.

[Til01] M. M. Tiller. *Introduction to Physical Modeling with Modelica.* Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[TS91] U. Tietze and Ch. Schenk. *Halbleiterschaltungsstechnik.* Springer-Verlag, Berlin, 1991.

[VB97] A. Vachoux and J.-M. Bergé, editors. *Analog and Mixed-Signal Hardware Description Languages.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[VDG00] M. Vogels, B. De Smedt, and G. Gielen. Modeling and simulation of a sigma-delta digital to analog converter using VHDL-AMS. In *IEEE/VIUF Int. Workshop on Behavioral Modeling and Simulation,* Orlando, October 2000.

[VHK+91] M. Valtonen, P. Heikkila, A. Kankkunen, K. Mannersalo, R. Niutanen, P. Stenius, T. Veijola, and J. Virtanen. APLAC - a new approach to circuit simulation by object-orientation. In *Europ. Conf. on Circuit Theory and Design,* pages 351–60, 1991.

[VS83] J. Vlach and K. Singhai. *Computer Methods for Circuit analysis and Design.* Van Nostrand Reinhold, New York, 1983.

[VVGS99] J. Vandenbussche, G. Van der Plas, G. Gielen, and W. Sansen. Behavioral model of reusable D/A converters. *IEEE Trans. on Circuits and Systems,* 46(10): 1708–18, 1999.

[VVV+99] G. Van der Plas, J. Vandenbussche, W. Verhaegen, G. Gielen, and W. Sansen. Statistical behavioral modeling for A/D converters. In *IEEE Int. Conf. on Electronics, Circuits, and Systems,* pages 1713–6, 1999.

[WGS98] P. Wambacq, G. Gielen, and W. Sansen. Symbolic network analysis methods for practical analog integrated circuits: A survey. *IEEE Trans. on Circuits and Systems,* 45(10): 1331–41, 1998.

[WVD+99] P. Wambacq, G. Vandersteen, S. Donnay, M. Engels, I. Bolsens, E. Lauwers, P. Vanassche, and G. Gielen. High-level simulation and power modeling of mixed-signal front-ends for digital telecommunications. In *IEEE Int. Conf. on Electronics, Circuits and Systems,* pages 525–8, 1999.

[YA91] K. Yoon and P. Allen. An adjustable accuracy model for VLSI analog circuits using lookup tables. *Analog Integrated Circuits and Signal Processing,* 1:45–63, 1991.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, San Diego, second edition, 2000.