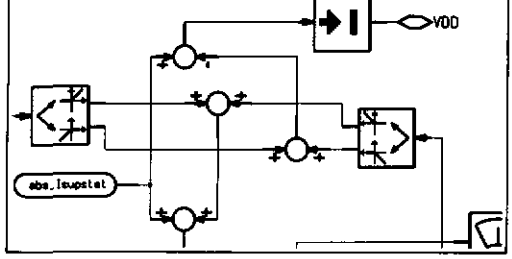
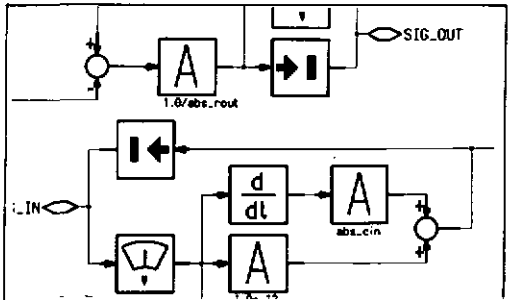


1320

Computer-Aided Behavioural Modelling of Analogue Systems

Vincent Moser

THÈSE SOUMISE À LA FACULTÉ DES SCIENCES
DE L' UNIVERSITÈ DE NEUCHÂTEL POUR L'OBTENTION
DU GRADE DE DOCTEUR ÈS SCIENCES



Computer-Aided Behavioural Modelling of Analogue Systems

Vincent Moser

THÈSE SOUMISE À LA FACULTÉ DES SCIENCES
DE L' UNIVERSITÉ DE NEUCHÂTEL POUR L'OBTENTION
DU GRADE DE DOCTEUR ÈS SCIENCES

IMPRIMATUR POUR LA THÈSE

Computer-Aided Behavioural Modelling of Analogue Systems.

de M. Vincent Moser

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

Messieurs F. Pellandini, N.F. de Rooij,
A. Vachoux (EPFL) et J.-Cl. Robert (ETA, Granges)

autorise l'impression de la présente thèse.

Neuchâtel, le 16 août 1996

Le doyen:

R. Dändliker

R. Dändliker

Abstract

In a top-down analogue design methodology, behavioural modelling is very useful because it allows the engineer to simulate a complete system at each development step. In this work, we developed the computer-aided analogue behavioural modelling tool ABSynth (Analogue Behavioural model Synthesizer). The behaviour to model is expressed graphically in the form of a *Functional Diagram* (FD) drawn as the interconnection of *Graphical Building Symbols* (GBS), each of which stands for some elementary analogue behaviour. The functional diagram describes the behaviour of the system only, not its physical structure. The corresponding AHDL (Analogue Hardware Description Language) code is then generated automatically. The generated code is behavioural—i.e., it is composed of differential algebraic equations and procedural statements—it is not a netlist.

The novel contribution of this work is threefold: first, analogue behaviour can be described in a dedicated graphical form, secondly, the graphical description is translated into behavioural AHDL code, lastly, the graphical description method and the code generation process have been implemented as a software tool. This tool, ABSynth, is user-friendly, easy to extend, and integrated in a complete design environment. The generated code is syntactically right by construction and, consequently, the user does not need to know the syntax of the target hardware description language anymore.

Résumé

Lorsque l'ingénieur suit une méthodologie de conception descendante, la modélisation comportementale est très utile parce qu'elle lui permet de simuler un système complet à chaque étape du développement. Dans le cadre de ce travail, nous avons développé l'outil de modélisation comportementale assistée par ordinateur ABSynth (Analogue Behavioural model Synthesizer). Le comportement à modéliser est exprimé sous la forme d'un *Diagramme Fonctionnel* (Functional Diagram, FD) construit en interconnectant des *Symboles de Construction Graphiques* (Graphical Building Symbols, GBS) qui représentent chacun un comportement analogique élémentaire. Le diagramme fonctionnel décrit uniquement le comportement du système, pas sa structure physique. Le code correspondant dans un langage de description de matériel analogique (Analogue Hardware Description Language, AHDL) est ensuite généré automatiquement. Ce code est dit comportemental — il est composé d'équations différentielles et d'instructions séquentielles — ce n'est pas un «netlist».

La contribution originale de ce travail se résume en trois points: premièrement, le comportement analogique peut être décrit graphiquement sous une forme dédiée, deuxièmement, la description graphique est traduite dans un langage de description de matériel, finalement, la méthode de description graphique et le processus de génération de code ont été implémentés sous forme d'un outil logiciel. Cet outil, ABSynth, est convivial, facile à étendre et bien intégré dans un environnement de développement. Le code généré est par construction conforme à la syntaxe du langage AHDL cible que l'utilisateur n'a donc plus besoin de maîtriser.

Zusammenfassung

Die Verhaltensmodellierung ist bei einer von oben nach unten verlaufenden Designmethode sehr nützlich, weil sie dem Ingenieur erlaubt, auf jeder Entwicklungsstufe ein vollständiges System zu simulieren. Im Rahmen dieser Arbeit haben wir das Programm ABSynth (Analogue Behavioural model Synthesizer) entwickelt, ein Werkzeug für die computerunterstützte Verhaltenamodellierung. Das zu modellisierende Verhalten wird anhand einea *funktionellen Diagramms* (Functional Diagram, FD) ausgedrückt. Dieses wird durch das Verbinden *grafischer Konstruktionssymbole* (Graphical Building Symbols, GBS), von denen jedes ein analogea Grundverhalten darstellt, erstellt. Das funktionelle Diagramm beschreibt nur das Verhalten des Systems, nicht seine innere Struktur. Der entsprechende Code wird dann automatiach in einer analogen Hardwarebeschreibungasprache (Analogue Hardware Description Language, AHDL) erzeugt. Dieser Code wird Verhaltenscode genannt und besteht aus Differentialgleichungen und sequentiellen Anweisungen, es handelt sich also nicht um eine Netzliste.

Der persönliche Beitrag dieser Arbeit besteht aus drei Teilen: erstens kann das analoge Verhalten in einer spezifischen Form grafisch beschrieben werden und zweitens wird die grafische Beschreibung in eine Hardwarebeschreibungssprache übersetzt. Schliesslich wurden die grafische Beschreibungsmethode und der Code-Erzeugungsprozess als Software-Werkzeuge unter dem Namen ABSynth implementiert. ABSynth ist benutzerfreundlich, kann leicht erweitert werden und ist zudem gut in einer Designumgebung integriert. Der erzeugte Code respektiert die Syntax der verwendeten AHDL-Sprache, und der Benützer braucht daher diese Sprache nicht mehr zu beherrschen.

Acknowledgements

A Ph.D. thesis presents theoretically the results of a personal research. However, such a work would not be possible without the contribution of numerous people.

First of all, I would like to thank my supervisor Professor Fausto Pellandini who put his trust in me and let me carry out this research with complete freedom. I would also like to thank Professor Nico de Rooij, Dr. Alain Vachoux and Mr. Jean-Claude Robert for accepting to co-examine this thesis and for their constructive comments.

I am also grateful to all the people who directly participated in this project: Dr. Jean-Pierre Amann helped me define the orientation of this research and accepted to share his experience in design methodologies and CAD development. Besides he was always available to solve any technical or organisational problem. He also devoted much time and attention in proofreading this text. Dr. Ruud Riem-Vis also played an important role in this work in laying the foundations of the program ABSynth. Pascàl Nussbaum and Luc Astier, both with CSEM SA Neuchâtel, have been directly involved in this project. ABSynth and the associated modelling method include many of their ideas. Christophe Calame contributed in studying applications of behavioural modelling and of the meet-in-the-middle analogue design strategy. He was also a valuable proofreader. Louisa Grisoni brought me her designer's point of view on A/D converter simulation and Sun Dequn studied mixed-mode simulation. Michel Weber, Michiel Klaarwater and Peter Annema also devoted much energy in exploring parts of this arduous research field. I also acknowledge and appreciate the financial support provided by the Swiss Foundation for Research in Microtechnology (FSRM, Neuchâtel).

Beaides, aome people highly contributed to the formal quality of this report. My thanks go to Lise-Marie Moaer who did an important effort in correcting thia text with reapect to Engliah, to Michel Haas who provided me with very useful advice of typographical matter, to Roseline Saam who realized the cover page design and to Thérèae Holzer who did the German translation.

I would also like to thank all my other colleaguea who make IMT a very agreeable place to work. I think in particular of our aecretary Catherine Lehnherr and of our system manager Heinz Burri who both let the whole infrastructure run very well deapite our continuously changing requeats. I also thank Sara Grassi and Ivan Defilippis for the very varied coffee corner discuasions.

Finally, I would like to thank all my friends and family, and especially Thérèae, who make life even more exciting than just microtechnology research.

Table of Contents

Chapter 1
Introduction

The research presented in this Ph.D. report addresses the process of modelling systems with a view to performing simulations. The systems considered here are those which exhibit an analogue behaviour. This behaviour is expressed in an analogue hardware description language with the help of a graphical computer program.

1.1 Motivation

The outstanding evolution of electronics from the realization of the first transistor in 1948 to the design of the 4.6 million transistors UltraSPARC™-I microprocessor in 1995 has been engendered by two main causes: the development of the technology and the broadening of the market. At the same time, the need for design methods and computer-aided design tools has grown dramatically.

The emergence of the technology of Integrated Circuits (IC) in the late 60s led to a big change in the design flow of electronic systems. The functionality of systems made out of discrete components can be tested using real components in a breadboard approach. An IC, however, once fabricated, cannot be modified. Its design must then be thoroughly verified before it is produced. This was the motivation for the development of the first circuit simulation tools. Nowadays, the technology continues to evolve towards smaller size and larger integration, which allows for more complex designs. This, consequently, leads to the need for more accurate and powerful

Computer-Aided-Design (CAD) tools to treat the huge amount of design data involved.

From the market point of view, the influence is twofold. On the one hand, the market of high performance applications, as in telecommunications and in the medical field, asks for more complex and faster systems. On the other hand, consumer electronics requires smaller and less expensive integrated systems and a shorter time to market. In both cases, CAD tools become a key issue.

In this discussion, we can also point out the links that exist between electronic design and computer science. Computers are used to design more powerful electronic circuits, which are used to build more powerful computers, which can then again be used to develop better circuits, and so on. Consequently, in most universities, the computer science department originates from the electrical engineering department. This can also explain why electronic CAD is so developed.

If we now briefly look at the recent history of integrated electronics and associated CAD tools we notice that one of the first CAD activities in the early 70s was analogue simulation. Then, in the late 70s and in the 80s, digital electronics experienced a tremendous success and much progress was made in digital CAD, which led to the development of standard hardware description languages and, in the early 90s, to automatic circuit synthesis tools.

Since the late 80s, however, mixed-signal integrated circuits have become very popular and the tendency to design all-digital circuits has gone down. In high performance applications, for instance, some critical parts may be considered as analogue and simulated in detail with continuous-time algorithms. Furthermore, particular functions are more efficient with respect to speed, surface or power when realized in an analogue way. Likewise, in signal processing system design, the interface with the outside world, which is essentially analogue, may be integrated in the same chip as the actual digital signal processing unit. Lastly, in microsystem design, analogue parts of different natures (e.g., sensors, actuators, pumps) may be built on the same system. For all those reasons, much research effort has been invested in analogue CAD development since the mid-80s.

However, an important gap still exists between digital and analogue CAD, which leads to high development time and cost. Furthermore, there is no global approach to microsystem design, which may combine digital, analogue and non-electrical components.

In this work, we intend to participate in the effort towards a reduction of this gap.

1.2 Scope of the Research

One important CAD task is the simulation of complex mixed-signal and mixed-nature systems. A computer simulation consists of two parts: a computer program, which implements a simulation algorithm and a model of the system to simulate. Traditionally, the system was described as the structural interconnection of basic standard components (e.g., transistors, capacitors) which were considered part of the simulator and hidden to the user. Some new analogue simulation environments, however, like most digital simulators, include a Hardware Description Language (HDL) that allows the users to write their own behavioural component models. This improvement is essential to faster top-down analogue design. Nevertheless, behavioural modelling is not an easy task and may be rejected by analogue designers, some of whom are not willing to learn and practice yet another computer language.

For this reason, we intend to ease the creation of analogue behavioural models by developing a computer-aided modelling tool based on a graphical user interface. The specifications of such a program can be summarized as follows:

- The behaviour of the component to model must be specified graphically using a simple formalism.
- Users do not need to know the syntax of the target hardware description language.
- The HDL code must be generated automatically based on the graphical description of the behaviour.
- The tool must include a user-friendly interface.
- The tool must be easy to extend towards more functionality and towards various HDLs. In particular, it must be possible to update the tool when a standard hardware description language (i.e., VHDL-AMS) is available.
- The tool must also offer a direct HDL entry to experts who want to by-pass the graphical interface.

This work describes a computer-aided modelling tool called ABSynth (Analogue Behavioural model Synthesizer) that is intended to meet these requirements. It has been implemented in Mentor Graphics's Falcon Framework — a commercial electrical design environment — and it generates models of analogue hardware in HDL-A — a commercial hardware description language by ANACAD.

1.3 Structure of the Report

After this short introduction, we will pass on, in chapter 2, to a survey of the basics of analogue design, which will introduce the reader more deeply into the general context of this work. Then, in chapter 3, some rules for the manual coding of behavioural models will be given based on the experience acquired during this project. Those readers who are not directly interested in manual modelling can skip this chapter without much loss of understanding of the next chapters. Then we will come to the heart of our work, which will be split up into two parts. In the first one, chapter 4, our method for the graphical description of the behaviour of analogue components will be exposed. In the second part, chapter 5, a description of the automatic code generation process implemented will be given. Then, the implementation of the tool ABSynth will be presented in chapter 6, and chapter 7 will be devoted to an application example. Finally, the conclusions will be drawn.

1.4 Previously Published Documents

The work described in this thesis has already been the subject of some publications. A first scientific paper was presented at the European Design and Test Conference in Paris in March 1994 [Mos94]. It includes the description of the graphical behavioural modelling method discussed in more detail in chapter 4. It also introduces the notion of *Functional Diagram* and *Graphical Building Symbol*. A second paper [Mos95], presented at EURO-VHDL'95 in Brighton, was devoted to HDL code generation—based on our graphical modelling method—as described in chapter 5. Finally, a user's guide for ABSynth has been written [Mos96].

[Mos94] V. Moser, P. Nussbaum, H. P. Amann, L. Astier and F. Pellandini, "A Graphical Approach to Analogue Behavioural Modelling", *The European Design and Test Conference,* pp. 535-539, 1994.

[Mos95] V. Moser, H. P. Amann, P. Nussbaum and F. Pellandini, "Generating VHDL-A-like Models Using ABSynth", *EURO-DAC'95 European Design Automation Conference with EURO-VHDL'95,* pp. 522-527, 1995.

[Mos96] V. Moser and P. Nussbaum, *ABSynth User's guide,* IMT Report 380 PE 01/95, Version 1.1, Université de Neuchâtel, Institut de microtechnique, 1996.

Chapter 2
Fundamentals of Analogue Design

The following chapter provides a short introduction to analogue design, simulation and modelling in order to allow the reader to catch the context of the work described in the next chapters. As many excellent textbooks already exist on analogue circuit design [All87], [Unb89], [Hui93], [vdP94], [San95], on analogue simulation [Sal94] and on analogue modelling [Ber95], [Man95], only the most important points and definitions are mentioned here. The emphasis is laid on simulation and modelling. When appropriate, the state-of-the-art of analogue design will be compared with that of digital design. Finally, some general properties of analogue hardware description languages are given.

The keyword *analogue* will be extensively used throughout this text. It designates signals which are *continuous* in time and in amplitude, systems which process such signals and methods used to design such systems.

Furthermore, only *lumped parameter systems*—i.e., systems that work at relatively low frequencies or, in other words, systems which characteristic dimensions are much smaller than the wave length of the processed signals—are considered. The dimensions of the components can be neglected and energy conservation is expressed as Kirchhoff's laws.

2.1 Analogue Design Flow

First of all, it is important to note that there is not a unique method for designing analogue electronic components. The development flow of this kind of designs highly relies on the designer's experience and may strongly vary from one project to another. Furthermore, the analogue design decisions, unlike their digital counterparts, cannot always be easily formalized in an algorithmic form. For these reasons, computer-aided analogue synthesis can be said to be still in its infancy.

2.1.1 Design Strategies

When talking about the process of designing analogue circuits, people usually mention top-down and bottom-up methodologies. We will introduce here those two classical approaches as well as an intermediate one called "meet-in-the-middle".

Top-down

In a top-down approach, as can be seen in figure 2.1, the system to design is first described globally based on the specifications. Then, some architectural solutions are studied and the system is decomposed into sub-systems. This is repeated hierarchically as many times as the complexity of the system makes it necessary, until the device level is reached. In analogue electronics, typical devices are transistors, capacitors and resistors. At each decomposition step, the new componenta are sized in order that the new architecture meets the specifications of the (sub-)system it represents. This decomposition and sizing process is called *synthesis*.

In order to validate the results of a synthesis step, the behaviour of the new sized architecture must be verified. To do this, a model of each sub-system must be available as well as a description of the architecture. The *verification* itself can be done either by *symbolic analysis* or by *simulation*.

The advantages of top-down design are numerous. As the behaviour of the system is verified at each synthesis step, most of the design errors can be found early in the design process. Furthermore,

as the description of the sub-systems does not contain all the implementation details, this verification procedure is quite fast.

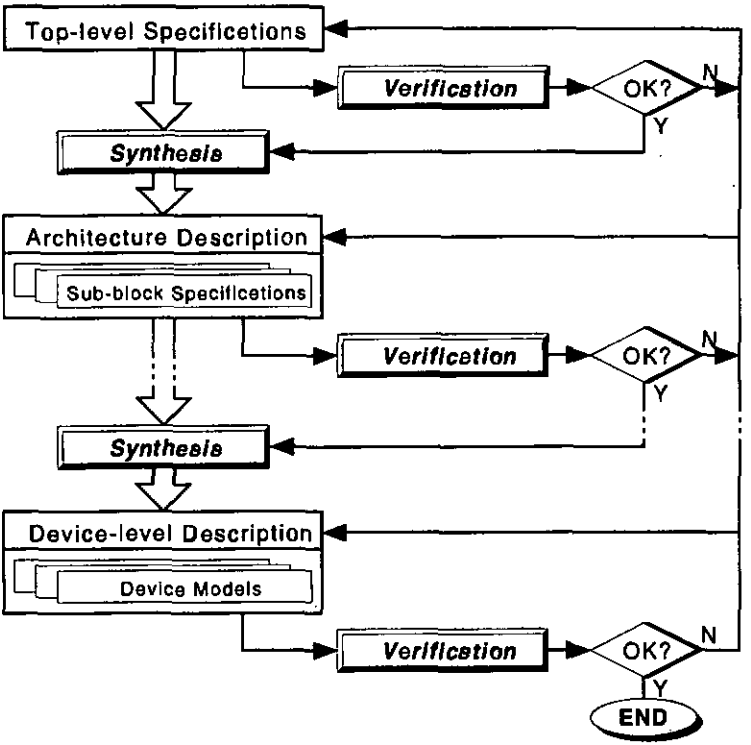


Fig. 2.1 Top-down design approach.

This approach has already been widely used for the design of digital electronics, where the devices are digital gates. It has even already been successfully implemented as automatic synthesis tools. In the analogue domain, however, this approach suffers from the fact that it is not possible to construct physically any sub-block one could ever specify. This means that much knowledge must be available on which low-level sub-systems can finally be built, and which sub-systems cannot. This knowledge, then, has much influence on the architectural decomposition of the system.

Bottom-up

The traditional bottom-up approach, illustrated in figure 2.2, is the opposite of the previous one. The designer uses well characterized devices to design basic cells. These cells can then be used as building blocks to compose a more complex system. The design is incrementally created based on a purely structural view. This is called the *composition* task. Then, the parameters of each new structure can be extracted either by simulation or by symbolic analysis. This is called the *characterization* task.

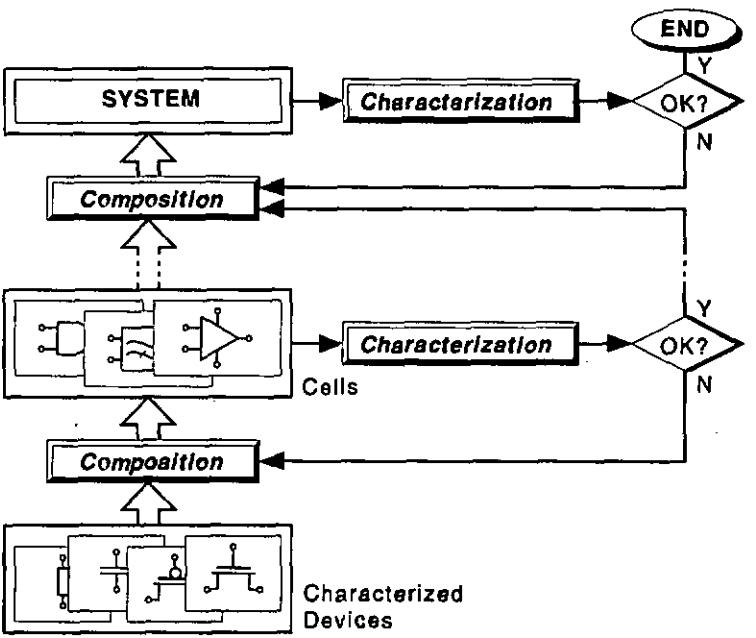


Fig. 2.2 Bottom-up design approach.

As the basic building blocks are made of well-known devices, a good simulation of the system should always be possible. However, the simulation of complex systems becomes too time-consuming, due to the large number of devices . It can even encounter convergence problems. Another drawback of this method is that it is very sensitive to changes in the specifications. Also, some basic design errors may be found only

when the whole system has been completely designed. This can lead to very costly redesigns.

On the other hand, as the designer only introduces realistic, or even optimized, building cells, the feasibility of the system is ensured.

Meet-in-the-middle

As an alternative, and in order to combine the advantages of the two previous methods, a third way can be defined as a *meet-in-the-middle* strategy, as described in figure 2.3.

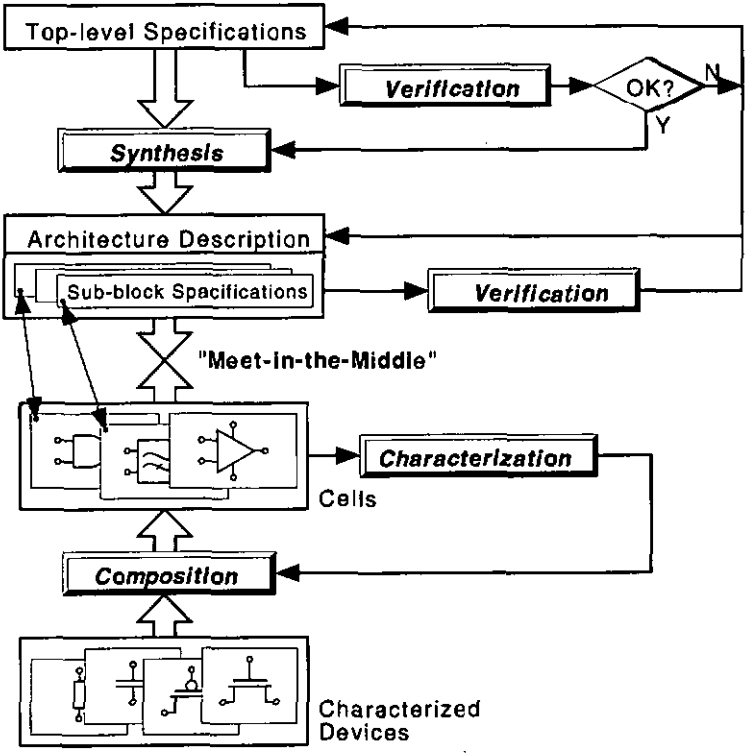


Fig. 2.3 Meet-in-the-middle design approach.

First, a library of well-known and precisely characterized building cells is developed in a bottom-up approach. The top-down method can

then be applied from the top description of the system to design until the decomposition reaches the level of these building cells. Antao and Brodersen express it in a different way in [Ant95]: *First a behavioural architecture is synthesized (...) the next stage generates an intermediate architecture in a specific circuit implementation style.*

This approach inherits the advantages of the top-down decomposition but, as it is guided towards the use of well-known cells, it is more realistic. It is similar to the standard cell based design of a digital ASIC. The difference is that, for the moment, not all the necessary analogue cells are available [Rut93].

2.1.2 Individual Tasks

In any of the aforementioned strategies, particular design tasks can be isolated. The main ones are the synthesis task, which can be divided into architecture selection, component sizing and layout, and the verification task, which can be realized either by simulation or by symbolic analysis. Symbolic analysis [Gie91], [Sed93] aims at deriving analytic equations of a system from the equations of the constituent devices or sub-systems. Simulation, on the other hand, aims at imitating the behaviour of the system using successive numerical computations; the behaviour of the system is coded into software models of the components.

2.1.3 Analogue Design Automation

Since the mid-80s, analogue design automation has become quite an important field of research. We will not review here the whole history of this activity. A good survey of the state-of-the-art can be found in [Gie91] or in [Rut93], while [Ant92a], [Cha92], [Ast93] and [Och94] describe particular implementations of the top-down methodology. In the remainder of this chapter we will concentrate on simulation and modelling and leave the other tasks aside.

2.2 Analogue Simulation

The aim of simulation is to verify the behaviour of a system under development. Among the various formal definitions of simulation

found in the literature, Cellier [Cel91] quotes a good one by Korn and Wait:

> *A simulation is an experiment performed on a model*

where

> *an experiment is the process of extracting data from a system by exerting it through its inputs*

and

> *a model* (M) *for a system* (S) *and an experiment* (E) *is anything to which* E *can be applied in order to answer questions about* S.

In the context of analogue electronic design, the experiment is performed using a computer program called a *simulator,* which implements a set of dedicated algorithms called the *simulator engine.* The model must be coded in a computer language in order to be understood by the simulator.

Usually the system is modelled as the interconnection of components. The model is then written as a list of nodes and a number of behavioural descriptions of the components. This leads to a set of equations, which either represent energy conservation law expressed as Kirchhoff's current law and Kirchhoff's voltage law (KCL, KVL) on the nodes or describe analytically the behaviour of the components. On each node, the simulator evaluates the value of signals which are continuous in time and in amplitude. Some of these signals represent physical quantities and can therefore be divided into two classes: the *across* quantities and the *through* quantities. In the description of electronic circuits, the across quantity is usually the voltage and the through quantity is the current.

A model can be subjected to different types of simulations. *DC simulation* aims at calculating a steady-state of a system. *AC simulation,* also called small-signal analysis, aims at obtaining the response of the system for a sinusoidal entry at various frequencies. A *transient analysis* gives the time-domain response of a system for an arbitrary input signal. A *thermal simulation* calculates the thermal dissipation of a system depending on various external conditions. Lastly, *noise analysis* aims at extracting the noise generated by a system. In this work, we have concentrated on DC, AC and transient simulations.

Among these types of simulation, the most critical in terms of CPU-time and convergence is transient analysis because the solving algorithms are quite complex. As mentioned in [Man95], traditional simulators like SPICE [Nag75] transform a set of ordinary differential equations into a set of non-linear algebraic equations. Then this set of equations is solved using iterative techniques (e.g., Newton-Raphson) and direct matrix techniques (e.g., LU decomposition). Several parameters allow the user to tune the algorithms for a particular application. This technique has been widely used since the mid-70s and has proven to be efficient although some intrinsic limitations exist [Bra93]. Besides, the approach has shown to be highly CPU-time-consuming, which limits its application to small size circuits (several hundreds of transistors). In SPICE and in most of its descendants, the behavioural descriptions of the models are coded in the simulator itself and cannot be modified by the user.

During the 80s, much research effort has been invested in developing better simulation algorithms using relaxation [Car87], [Hen87], piece-wise-linear simulation [Lon94] and also event-driven analogue simulation [Cot90]. Simultaneously, another important direction of research was towards the development of more accurate models for the pre-defined devices and especially MOS transistors [Vit93]. Then, in the late 80s and early 90s, a new step was taken with the development of new Analogue Hardware Description Languages (AHDL) which allow the user to model custom systems at a behavioural level [Get89], [Kur90], [ElT91]. In a top-down design approach, the models of the sub-systems are used to simulate a complete system under development. In a bottom-up approach, the behaviour of already designed building cells is coded in order to perform faster simulations.

Finally, a standard AHDL called VHDL-AMS IEEE 1076.1 is currently being developed by a committee of the IEEE on the basis of the semantics of the VHDL (VHSIC Hardware Description Language) language defined in reference [IEE93]. VHDL-AMS will be a super-set of VHDL and will therefore include complete mixed analogue-digital modelling facilities. This initiative is one of the numerous activities supported by the IEEE to widen the scope of VHDL. It will help to promote unified modelling techniques and to facilitate the exchange of models between groups of designers working with different simulators. Additionally, the techniques and the languages developed to model and simulate analogue electrical systems can also

be applied to systems which include non-electrical parts—e.g., sensors, actuators, acoustical transducers, etc.

2.3 Analogue Modelling

Traditionally, analogue component modelling was reserved to experts who developed very precise models of devices, particularly transistors. The models were written in general-purpose computer languages (e.g., C) and linked to the simulator. Nowadays, the promoters of AHDLs intend to make modelling more popular among designers and must therefore make it a less specialized task. However, the most important point remains unchanged: a good understanding of the system or device to model is essential to write a good model. The coding technique comes next. This section introduces general aspects of analogue modelling while the actual model coding task is detailed in the next chapters.

As already pointed out in the previous section, a model of an analogue system can be represented either as a list of nodes and a list of instances—both lists constituting a *netlist*—to describe its *structure* or as a list of equations and/or other statements to describe its *behaviour*. A structural description can be hierarchical, each inatance of a netlist being also described as a netlist, but, as the decomposition cannot be infinite, it always ends up with instances that are considered at the behavioural level. This hierarchical atructure can be seen as a tree, where the behavioural instances are the leaves.

Beside the structural or behavioural internal description, a model also has an external view, which describes its interface. It comprises pins and parameters. The pins are used to instantiate the component in a structural description. The parameters are used to set up the model to a particular application.

2.3.1 Modelling Levels

In addition to different description styles, a system under development can be described at various levels of abstraction. This can be represented in the *Y-chart* [Gaj87] of figure 2.4. The system can be described along three axes, each of which stands for a different view. The abstraction level is indicated on each axis starting from the centre where the abstraction is at its lowest.

As already pointed out previously, a description of a system comprises a structural part and behavioural descriptions of the component instantiated. On the structural axis we indicate the type of component used while on the behavioural axis we indicate how the component's model is described. On the physical axis, the corresponding physical cells are given. If we now consider the concentric circles, we can define several modelling levels, each of which will be described in more detail below, starting from the origin of the axes.

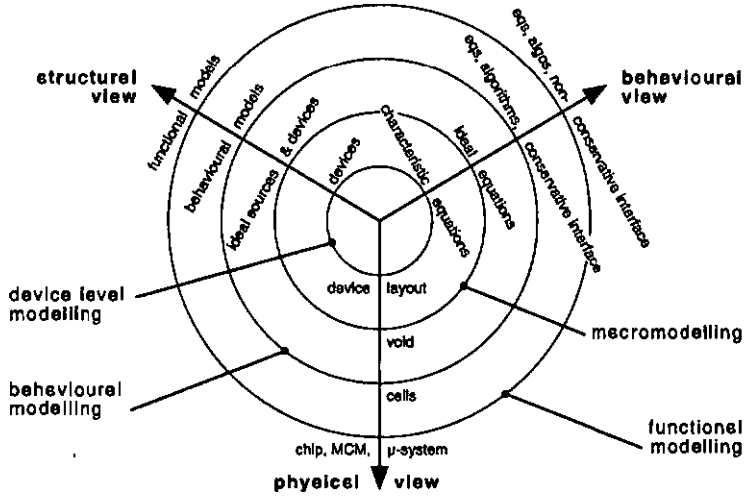


Fig. 2.4 Y-chart applied to analogue modelling.

In the area of digital design, the definition of 6 abstraction levels is widely accepted [Ram93]. In the analogue area, however, these modelling levels are defined more arbitrarily. Intermediate levels could also be defined. Furthermore, a system is usually described in a mixed-level manner, where a combination of the modelling levels is used. Also, note that the notion of abstraction level is related to the structure—the more detailed the structure, the less abstract the description. It is not always linked to the accuracy of the description, which depends on the level of detail of the associated behavioural descriptions. The modelling level classification given here is not the expression of a universal consensus. In particular, some authors define behavioural modelling as the most abstract level [San96].

Device Level Modelling

This level is also called primitive level. It is the least abstract description. The user describes the system structurally using standard devices (e.g., transistors, capacitors, resistors), whose behavioural models are included in the simulator code. These models are made of rather detailed characteristic equations of the devices whereas second-order effects are also taken into account. Various models can be used depending on the mode of operation of the device or on the degree of accuracy or simulation speed needed. In the structural description, the connection points are electrical—or other physical—nodes. They are described by KCL and KVL. Usually, device level modelling is used either graphically with circuit diagrams—as will be explained in chapter 4—or in a structural HDL like SPICE (see § 2.4.3).

Extensions of the model libraries are possible but this task is reserved to experts, who know the operation of the device very well. The modelling language is usually a conventional computer language (e.g., C). For each device and for each model available, parameters are extracted by measurement on real devices and supplied by foundries. Usually, little comfort is provided to the model expert, while more computer assistance is provided to the designer.

Macromodelling

Here, models of ideal components (e.g., resistors, capacitors, inductors, independent and dependent sources) are available. They are used to build a circuit which mimics the behaviour of the system instead of describing its actual structure [Boy74]. Again, the connection points are electrical nodes. Macromodels can be described in the SPICE language or as particular circuit diagrams as shown in figure 2.5 in the case of an operational amplifier.

This method is quite tricky—it uses electrical nodes to represent abstract variables—and the number of behavioural elements (i.e., ideal electrical devices) available is limited but it is well integrated in most simulation environments.

Macromodelling was used when HDLs were not available and it will probably be gradually replaced by behavioural modelling. This approach is similar to the prototyping of integrated filters using RLC constructs.

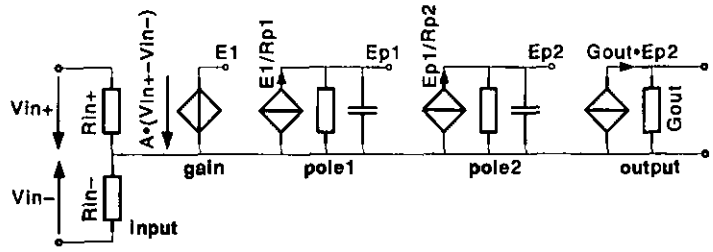


Fig. 2.5 Operational amplifier macromodel.

Behavioural Modelling

At this level, the user builds behavioural models in an HDL using differential equations, algorithmic sequences of statements or even tables of values. Any analogue, mixed-signal or even non-electrical behaviour can be described. The connection points of such models represent physical continuous-time signals, which are not limited to electrical ones but can be of any nature. They are governed by generalized conservation laws. For this reason, some typical behaviour like input impedance, output impedance and power supply can be included in the behavioural models.

Once the user has become familiar with the HDL, this method is very efficient since it is theoretically possible to describe any dynamic system. Furthermore, the modelling level of detail can vary from idealized models to very accurate ones. However, the current simulators are not optimized for behavioural simulation and can encounter convergence problems, especially in the presence of discontinuities.

The parameters of the models are either given by the user to express specifications or are extracted from existing circuits by lower level simulation or by measurement.

Functional Modelling

This most abstract modelling level is used to described complex syatems with little accuracy. Again, the models are described using an HDL and are connected together in order to form a block diagram. The connection points are not conservative but rather indicate a transfer

of information as in a signal-flow model. This can be realized either using a dimensionless variable (sometimes called a coupling) or using either the across or the through quantity of a physical interface. As they are not conservative, functional models must be handled carefully when they are integrated into a mixed-level description.

2.3.2 **Analogue Model Generation Tools**

Currently, several analogue modelling tools are available but most of them cover only the structural part of a system's description. The user places and connects symbols of components using a graphical editor, then the netlist is automatically extracted. The behavioural models of the components are either available in the libraries or they must be "hand-typed" by the user.

The behavioural modelling part, however, has not been extensively automated yet. Some interesting tools have been developed to generate behavioural models of s-domain and z-domain transfer functions. The tools *modgens* and *modgenz* [Vis88] convert transfer functions into state-space representations in the time-domain and generate the corresponding behavioural models in C. Similarly, *gensims* and *gensimz* [Ant92b] generate ABCDL behavioural models starting from transfer functions. ABCDL (Analogue Behaviour Circuit Description Language) is an HDL by AT&T Bell Laboratories.

2.3.3 **Desirable Evolution**

The behavioural model generation tools available are limited to a particular class of behaviour (e.g., transfer functions). Furthermore, they usually generate models coded in general-purpose programming languages (e.g., C) instead of AHDLs and linked to a simulator, which limits their portability.

In order to let the designers benefit from all the modern modelling possibilities, graphical-based analogue behavioural model generation tools would be welcome, especially such tools which allow the user to describe some arbitrary behaviour. The models should be coded in a standard hardware description language in order to get portable models which give the same simulation results on various simulators. Furthermore, the models generated should be as compact and optimized as user-written ones.

As we will see, the model generator ABSynth (Analogue Behavioural model Generator) developed in this project fulfils most of these requirements.

2.4 Analogue Hardware Description Languages

In the previous section, the concept of model was defined from a theoretical point of view. To be used in a simulation, a model must be formalized in a language that can be understood by a computer program. This section deals with specialized computer languages used to code analogue models.

2.4.1 Generalities

Historically, models of analogue hardware were written in general-purpose programming language (e.g., C). Users had to define their own functions and procedures to express the semantics of an analogue model. Furthermore, the coding style was influenced by the target simulator engine. This was acceptable as long as modelling was reserved to experts and as long as this task was limited to a few device models. In a modern top-down design process, however, many models of arbitrary systems and components are written every day by designers. Therefore, a more robust and comfortable modelling framework must be provided and a dedicated language is necessary. When the object of modelling is a piece of hardware, this dedicated language is called a Hardware Description Language (HDL).

In this section, as we focus on analogue systems, we will introduce the notion of Analogue Hardware Description Language (AHDL), of which Mantooth and Fiegenbaum give a good definition [Man95]:

> *An analogue HDL is a programming language that is specifically designed to allow the description (or modelling) of hardware that performs a continuous-time function.*

In other words, an AHDL is a computer language whose semantics is limited to the description of analogue hardware and whose syntax includes the corresponding constructs. Additionally, as AHDL models will be used in association with an analogue simulation kernel, some simulation semantics—e.g., Kirchhoff's laws—is implicitly

present. Furthermore, it is important to keep in mind that only the aspects which are relevant to simulation will be included in the model.

As we only consider lumped parameter systems, only structural and/or behavioural description facilities must be included in the description language, and the physical view is not treated. However, to describe *distributed parameter systems*—e.g., microwave applications described by Maxwell's equations—the size of the components should be taken into account and a physical description would then also be necessary.

AHDL Model Contents

In general, a model coded in an AHDL includes an external view composed of connection points (pins) and parameters and an internal view which describes the structure and/or the behaviour of the system.

The structure is described as a, possibly hierarchical, list of *instances* connected through *nodes*. This description can include a mix of different modelling levels. The nodes are physical connection points where energy conservation laws apply. The instances can be sized to a particular application through parameters.

The behaviour is described as a set of differential algebraic equations (DAEs) and/or a set of sequential statements. The description may be different for different analyses (DC, AC, transient). Several variable types are available, like physical (across-through) type, real type, and boolean type.

2.4.2 Mixed-Mode Representation

In many practical situations, the system to design is composed of analogue and digital parts. Therefore, it is desirable to have a unified language to model the whole system. Such a language must include the semantics of a digital HDL and the semantics of an analogue HDL but it must also include additional constructs that govern the communication between the two descriptions.

Two problems must be solved: first, an amplitude conversion must be placed between the continuous-amplitude analogue state variables and the quantified digital signals. Second, a time conversion must be provided between the continuous-time analogue description and the sampled-time digital model. This is out of the scope of this work and

will not be developed further. The reader can refer to [Man95], [Ber95] or [Sal94] for more details on this topic.

2.4.3 Existing AHDLs

Several AHDLs have existed for years. Most of them were designed by private companies for internal use. We will briefly present some of the publicly available ones.

SPICE

The SPICE language is the entry format to the well-known SPICE simulator [Nag75]. As most of the analogue simulators were based on the SPICE methodology, this format has been widely used, each vendor adding some extensions.

SPICE is a purely structural language and it is limited to device-level modelling and macromodelling. Hierarchical models can be described using sub-circuits. The behavioural models of the devices are all included in the simulator and cannot be accessed by the user. The device library includes common electronic and electrical components as well as ideal sources.

Basically, a SPICE file (also called SPICE netlist) consists of a list of instance statements, each of which begins with the name of the instance followed by a list of nodes to which it is connected and by the values of the parameters. The first character of the name indicates the device type. In the following example of an operational transconductance amplifier (OTA), MOS transistors are marked with a leading M, capacitor with a leading C and voltage source with a leading V.

```
m1 3 inn 1 vss n w = 4.3E-05 l = 2E-06
m2 2 inp 1 vss n w = 4.3E-05 l = 2E-06
m7 4 4 vss vss n w = 8E-06 l = 2.2E-05
m8 out 4 vss vss n w = 8E-06 l = 2.2E-05
m9 1 5 vss vss n w = 8E-06 l = 2.2E-05
m10 5 5 vss vss n w = 8E-06 l = 2.2E-05
m3 2 2 vdd vdd p w = 1.8E-05 l = 2.2E-05
m4 out 2 vdd vdd p w = 1.8E-05 l = 2.2E-05
m5 3 3 vdd vdd p w = 1.8E-05 l = 2.2E-05
m6 4 3 vdd vdd p w = 1.8E-05 l = 2.2E-05
```

```
mp1 vdd vdd p1 p1 n w = 2E-06 l = 3E-06
mp2 p1 p1 vss vss n w = 3E-06 l = 3E-06
mp3 5 p1 vdd vdd p w = 1E-06 l = 2E-06
vdd vdd 0 1.25
vss vss 0 -1.25
c1 out 0 2e-12
c1 in1 inn 24p
c2 inn out 1p
.end
```

FAS

The FAS language is a purely behavioural language developed by ANACAD [ANA92]. It is limited to the description of analogue systems. The behavioural models coded in FAS can be instantiated in an extended SPICE-like structural description of a circuit.

The following—incomplete—example of code illustrates the structure of a FAS description.

```
elliptic5
amodel elliptic5 (in,vdd,vss,pol,out) ;
   declare pin out : electrical ;     (* declaration part *)
   declare state youtv : real ;
   declare state limnflag,limpflag : boolean ;
   declare param gout : real ;
   ...
   initialize              (* default value of parameters and *)
      make youtv=0         (* initial values of variables *)
      make limnflag=0
      make limpflag=0
      make gout=1e-2
   ...
   endinitialize
   analog       (* begin of actual description of the behaviour *)
      if (mode=dc) then    (* particular statements for DC mode *)
         make iout = (youtv-yout-dcout)*gout
         make curr.on(out)=iout
      else  (* particular statements for AC and transient mode *)
         if (limpflag=1) then
            make iout=(youtv-yout-dcout)*goutlimp+ilimp
         endif
```

```
      endif
      make curr.on(out)=iout
   ...
   endanalog
endmodel
```

The interface (pin list) of the model is coded first, followed by a declaration part and an initialization part. Then, the actual description of the analogue behaviour is given as a series of statements. They can describe algorithmic parts, explicit equations as well as implicit equations. By default, the statements are valid for any analysis but some of them can be reserved to a particular analysis mode.

HDL-A

HDL-A is also a behavioural language by ANACAD [ANA94a]. Again, the structural descriptions are expressed by means of SPICE-like netlists. However, HDL-A is a mixed analogue digital HDL since it includes a *relation* block for analogue descriptions and a *process* block for digital descriptions. The *process* block is a sub-set of the behavioural modelling facility of VHDL. Unlike FAS, the HDL-A analogue part is clearly separated into analysis-specific code blocks. Similarly, equations are separated from procedural descriptions.

The following code is a partial example of an HDL-A model.

```
ENTITY rsd_atod_mix IS
    GENERIC (v_thr : real;
             ...);
    PIN (ANA_IN : electrical;
         ...);
END ENTITY;
ARCHITECTURE mixed OF rsd_atod_mix IS
    STATE vin, vclk : analog;
    VARIABLE tdel : real;
    ...
BEGIN
    RELATION
        PROCEDURAL FOR init =>
            v_thr := 0.625;
            vin := 0.0;
```

```
            ...
        PROCEDURAL FOR dc =>
            vin := ana_in.v;
            ana_in.i %= 0.0;
            ...
        PROCEDURAL FOR transient =>
            vin := ana_in.v;
            iin := cin*ddt(vin);
            ana_in.i %= iin;
            ...
    END RELATION;
    PROCESS
    BEGIN
        yout <= vin;
        ...
        LOOP
            WAIT ON rising(vclk,clk_thr);
                IF (count=n-1.0) THEN
                    wout := ymem;
                ELSE
                    wout := ymem;
                END IF;
                yout <= wout;
                ...
        END LOOP;
    END PROCESS;
END ARCHITECTURE mixed;
```

First, a description of the interface of the model is given as a list of parameters and a list of pins. Then the analogue part begins with an initialization block followed by several analysis-specific procedural or equation blocks. Finally, the digital part is described as a single process.

MAST

MAST is a language by Analogy. It is quite powerful since it covera behavioural and structural modelling as well as analogue and digital modelling. Analogue behaviour can be coded as sequential statements, equations or lookup tables. A complete presentation of the modelling possibilities offered is given in [Man95].

VHDL-AMS

VHDL-AMS is a mixed-mode, mixed-nature language currently (1996) under design by the 1076.1 working group of the IEEE, which aims at defining standard analogue extensions to the digital modelling language VHDL. The specifications for these extensions have been gathered into a Design Objective Document (DOD) [IEE95]. A commented description of these requirements is given by Shi and Vachoux [Shi95].

MHDL

MHDL (MIMIC Hardware Description Language) [Bar93], [Rho95] is a modelling language which allows the user to describe mixed analogue (lumped parameter) and microwave (distributed parameter) systems. To describe mixed analogue-digital systems, an MHDL model can be instantiated in a VHDL netlist.

CSSL

Another class of interesting languages is used in control engineering. Most of them originate from the Continuous System Simulation Language (CSSL). Unlike regular AHDLs, CSSL does not implicitly include Kirchhoff's law semantics. For this reason, it can not be considered strictly as an AHDL. However, CSSL can be used to describe and simulate a large class of dynamic systems.

Cellier [Cel91] explains that it is based on a state-space description of the system which is equivalent to a set of first order ordinary differential equations (ODEs). Therefore, CSSL is limited to the description of linear systems. Like in most AHDLs, these equations represent a number of statements which are simultaneously true. However, a CSSL compiler must first sort the equations into an executable sequence. This equation sorting process requires that every variable is declared only once and that no algebraic loops are present—i.e., if variable a is a function of variable b, variable b cannot be a function of variable a.

Cellier also introduces the derived languages ACSL, DARE-P, and DESIRE.

2.5 References

[All87] P. E. Allen and D. R. Holberg, *CMOS analog circuit design,* Holt, Rinehart and Wilson, New York, 1987.

[ANA92] *ELDO-FAS Dynamical System Modeling,* Version 4.1.x, ANACAD Computer Systems, 1992.

[ANA94a] *HDL-A User's Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

[Ant92a] B. A. A. Antao and A. J. Brodersen, "Techniques for Synthesis of Analog Integrated Circuits", *IEEE Design & Test of Computers,* March 1992, pp. 8-18, 1992.

[Ant92b] B. A. A. Antao and F. M. El-Turky, "Automatic Analog Model Generation for Behavioral Simulation", *IEEE 1992 Custom Integrated Circuits Conference,* pp. 12.2.1-12.2.4, 1992.

[Ant95] B. A. A. Antao and A. J. Brodersen, "A Framework for Synthesis and Verification of Analog Systems", *Analog Integrated Circuits and Signal Processing,* Vol. 8, No. 2, pp. 183-199, Kluwer Academic Publisher, Boston, 1995.

[Ast93] L. Astier et al., "Plan-Frame®: A Framework for Design Knowledge Capture", *The European Conference on Design Automation with The European Event in ASIC Design, USER FORUM,* pp. 97-101, 1993.

[Bar93] D. L. Barton and D. D. Dunlop, "An Introduction to MHDL", *1993 IEEE MTT-S Digest,* pp. 1487-1490, 1993.

[Ber95] J.-M. Bergé, O. Levia and J. Rouillard (eds.), *Modelling in Analog Design,* Kluwer Academic Publisher, Boston, 1995.

[Boy74] G. R. Boyle et al., "Macromodeling of Integrated Circuit Operational Amplifiers", *IEEE Journal of Solid-State Circuits,* Vol. SC-9, No. 6, pp. 353-363, 1974.

[Bra93] A. Brambilla and D. D'Amore, "The Simulation Errors Introduced by the Spice Transient Analysis", *IEEE Transaction on Circuits and Systems – I: Fundamental Theory and Applications,* Vol. 40, No. 1, pp. 57-60, 1993.

[Car87] C. H. Carlin and A. Vachoux, "MOSART: A Large Scale Time-Domain Simulator Based on Waveform Relaxation", *European Conference on Circuit Theory and Design,* pp. 227-232, 1987.

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

[Cha92] H. Chang et al., "A Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits", *IEEE 1992 Custom Integrated Circuits Conference*, pp. 8.4.1-8.4.6, 1992.

[Cot90] R. A. Cottrell, "Event-Driven Behavioural Simulation of Analogue Transfer Functions", *The European Design Automation Conference*, pp. 240-243, 1990.

[ElT91] H. El Tahawy et al., "Towards a VHDL Modeling and Simulation Language for Mixed (Analog/Digital) VLSI Circuits", *EURO-VHDL'91*, pp. 251-259, 1991.

[Gaj87] D. D. Gajski, "The Structure of a Silicon Compiler", *IEEE International Conference on Computer Design*, 1987.

[Get89] I. E. Getreu, "Behavioral Modeling of Analog Blocks Using the Saber Simulator", *32nd Midwest Symposium on Circuit and Systems*, pp. 977-980, 1989.

[Gie91] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*, Kluwer Academic Publisher, Boston, 1991.

[Hen87] B. Hennion et al., "ELDO: A General Purpose Third Generation Circuit Simulator Based on the O.S.R. Method", *European Conference on Circuit Theory and Design*, pp. 113-119, 1987.

[Hui93] J. H. Huijsing, R. J. van de Plassche and W. Sansen (eds.), *Analog Circuit Design*, Kluwer Academic Publisher, Boston, 1993.

[IEE93] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993, The Institute of Electrical and Electronics Engineers Inc., 1993.

[IEE95] IEEE PAR 1076.1 VHDL Analog Extensions, *VHDL-A Design Objective Document*, Version 2.3, 1995.

[Kur90] C. M. Kurker et al., "Development of an Analog Hardware Description Language", *IEEE 1990 Custom Integrated Circuits Conference*, pp. 5.4.1-5.4.6., 1990.

[Lon94] D. I. Long and S. Medhat, "Behavioural Modelling of Mixed-Signal ASICs: A New Multi-Level Approach", *IEEE International Symposium on Circuits and Systems*, pp. 331-334, 1994.

[Man95] H. A. Mantooth and M. Fiegenbaum, *Modeling with an Analog Hardware Description Language*, Kluwer Academic Publisher, Boston, 1995.

[Nag75] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL Memo ERL-M520, Univeraity of California, Berkeley, 1975.

[Och94] E. S. Ochotta et al., "Analog Circuit Synthesis for Large, Realistic Cells: Designing a Pipelined A/D Converter with ASTRX/OBLX", *IEEE 1994 Custom Integrated Circuits Conference,* pp. 15.4.1-15.4.4, 1994.

[Ram93] F. J. Rammig, "System Level Design", *Fundamentals and Standards in Hardware Description Languages,* pp. 109-151, J. P. Mermet (ed.), Kluwer Academic Publisher, Dordrecht, 1993.

[Rho95] D. L. Rhodes, "Analog Modeling Using MHDL", *Modelling in Analog Design,* pp. 47-92, J.-M. Bergé, O. Levia and J. Rouillard (eds.), Kluwer Academic Publisher, Boston, 1995.

[Rut93] R. A. Rutenbar, "Analog Design Automation: Where Are We? Where Are We Going?", *Proc. IEEE 1993 Custom Integrated Circuits Conference,* pp. 13.1.1-13.1.8, 1993.

[Sal94] R. Saleh, S.-J. Jou and A. R. Newton, *Mixed-Mode Simulation and Analog Multilevel Simulation,* Kluwer Academic Publisher, Boston, 1994.

[San94] W. Sansen, J. H. Huijsing and R. J. van de Plassche (eds.), *Analog Circuit Design,* Kluwer Academic Publisher, Boston, 1994.

[San96] A. Sangiovanni-Vincentelli, private conversation, March 1996.

[Sed93] S. J. Seda, *Symbolic Analysis for Analog Circuit Design Automation,* Hartung-Gorre Verlag, Konstanz, 1993.

[Shi95] C.-J. R. Shi and A. Vachoux, "VHDL-A Design Objectives and Rationale", *Modelling in Analog Design,* pp. 1-30, J.-M. Bergé, O. Levia and J. Rouillard (eds.), Kluwer Academic Publisher, Boston, 1995.

[Unb89] R. Unbehauen and A. Cichocki, *MOS Switched-Capacitor and Continuous-Time Integrated Circuits and Systems: Analysis and Design,* Springer-Verlag, Berlin, 1989.

[vdP95] R. J. van de Plassche, W. M. C. Sansen and J. H. Huijsing (eds.), *Analog Circuit Design,* Kluwer Academic Publisher, Boston, 1995.

[Vis88] C. Visweswariah et al., "Model Development and Verification for High Level Analog Blocks", *25th ACM/IEEE Design Automation Conference,* pp. 376-382, 1988.

[Vit93] E. A. Vittoz, "Micropower Techniques", *Design of Analog-Digital VLSI Circuits for Telecommunication and Signal Processing,* J. E. Franca and Y. P. Tsividis (eds.), Prentice Hall, 1993.

Chapter 3
Manual Coding

Before going towards modelling automation, it is important to master the process of modelling itself. The aim of this chapter is not to offer a complete modelling course but rather to provide some hints and some solutions to particular modelling problems. The modelling method presented here has been developed with the idea of automatic HDL code generation in mind. It is based on a "divide-and-conquer" approach where several partial behavioural units are combined to build a more complex behavioural model.

Thanks to the success of VHDL (VHSIC Hardware Description Language), digital modelling has become a well-established practice. We will therefore concentrate on analogue modelling with some extensions to mixed-signal modelling.

As exposed in the previous chapter, a model can be either behavioural or structural. Structural analogue modelling is already well-known for it has been applied since the 70s to describe electronic circuits using the SPICE formalism [Nag75]. Furthermore, as VHDL structural modelling is also quite common, we assume that structural modelling in VHDL-AMS—the mixed-mode extension of VHDL—will be straightforward. Consequently, we choose to concentrate on behavioural modelling, which, in any case, will always be necessary because structural descriptions have endings which must be described at behavioural level.

As the purpose of this research is the modelling and the simulation of physical continuous-time components and systems, we will also limit our investigations to models that have a *physical* (i.e., across-through)

interface. Signal flow graph modelling will not be treated in this chapter.

The interface, which allows the model to communicate with its environment, will be described first. Then, the major properties of the signal processing part, which describes the specific behaviour, will be treated, including the coding of s-domain transfer functions, z-domain transfer functions and the description of some typical building blocks.

The modelling method we propose could theoretically be implemented in any analogue hardware description language. Even if the terminology and the lexical definitions vary from one language to another, the principles remain. In this chapter, the code examples will be given in HDL-A and the terminology will be mostly the HDL-A terminology, although we will sometimes use the VHDL-AMS terminology [Shi95]. A complete description of HDL-A can be found in the Language Reference Manual [ANA94a]; the properties of HDL-A which are necessary to the understanding of this chapter are also briefly summarized in appendix C.

3.1 Model Interface

In the modelling style we have established, the interface of a model is composed of physical connection points (HDL-A `pin`) and of parameters (HDL-A `generic`). The pins serve as connections between models when they are instantiated in the structural description of a circuit. A pin carries a physical quantity represented as a couple of an *across* variable and a *through* variable as explained in chapter 2. The parameters allow the user to set up a model to a particular application.

3.1.1 Pins and Interface Stages

In an analogue circuit, we can define a direction of propagation of the information through a chain of signal processing elements. The physical quantities that carry this information, however, are directionless. Each model is influenced by the across quantities on all the nodes to which it is connected and by the through quantities on all its pins. Reciprocally, each node is influenced by all the components connected to it. Consequently, a model can read the value of the across and of the through quantity on any pin but it can also impose its own *contributions* to these values. The final value of the across and

through quantities are then computed at simulation time according to the contributions of all the components.

In practice, the four read/contribute possibilities are rarely used simultaneously. According to our experience, most of the models can be realized with a restrictive pin usage: tbe model uses the value of the across quantity on all the surrounding nodes to calculate the value of the through quantity on each pin. In the code example given below, the same pin identifier appears on the left-hand side and in the right-hand side of the same assignment statement. The across quantity is read, then the value of the through quantity is computed and imposed as a contribution to the pin.

```
sig_out.i %= (sig_out.v - v0)/rout;
```

Theoretically, the dual way—i.e., read the through quantity and calculate the across quantity—can also be followed but it seems that some simulators hardly accept it.

As another convention, we attribute a positive sign to any through quantity that enters the model. According to this choice, we can define a complete model interface which does not only include pins but also builds a complete input stage, a complete output stage or a power supply block.

The across and through quantities on a model pin are node and branch quantities of the surrounding circuit. For this reason, we choose not to initialize them inside the model but rather in the set-up of the simulation.

Note that when we impose a quantity to a pin, it means that this pin is implicitly used as a source. Much care must then be taken when the model is instantiated into a circuit description so that ideal sources are never connected together, which could lead to incompatible connections as explained in [Boi83].

Input Stage

Some pins can be considered as passive accesses where either the across or the through value is read and the dual value is imposed. Note that the value imposed may be the result of complex non-linear computations. Functionally, this access can be seen as an *input stage*.

We can define an input stage as an input impedance Z_{in}, which is placed between the considered pin and an implicit reference node. We calculate the input current according to Ohm's law.

$$i_{in} = V_{in}/Z_{in}$$

In some cases, this impedance is composed of a resistance R_{in} and a capacitance C_{in} connected in parallel as shown in figure 3.1.

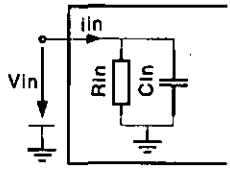


Fig. 3.1 Parallel RC input stage.

The corresponding equation is

$$i_{in} = C_{in}\frac{dV_{in}}{dt} + V_{in}/R_{in} \ ,$$

which gives the following HDL-A code:

```
IN_PIN.i %= Cin*ddt(IN_PIN.v) + IN_PIN.v/Rin
```

Output Stage

Some other pins can be considered as ideal sources: one quantity is imposed, the dual one can have any value. If the source is non-ideal, an internal equivalent impedance can be defined. This source-type behaviour characterizes an *output stage.*

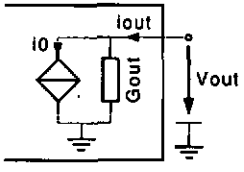


Fig. 3.2 Non-ideal resistive output stage.

Using our conventions, an output stage will basically be defined as a current source. If the current source is ideal, the calculated value of the current is directly imposed to the pin. If the source is non-ideal,

however—as shown in figure 3.2—the current on the actual output pin is

$$i_{out} = i_0 + V_{out} \cdot G_{out}$$

where i_{out} is the value of the actual current, i_0 is the value of the equivalent ideal current, V_{out} the voltage read on the pin and G_{out} the output conductance of the source. The corresponding HDL-A code is

```
OUT_PIN.i %= i0 + OUT_PIN.v*Gout;
```

Similarly, a non-ideal voltage source can also be defined. The characteristic equation of the source is

$$V_{out} = V_0 + i_{out} \cdot R_{out}$$

where V_{out} is the voltage on the pin, V_0 is the ideal voltage, i_{out} is the current on the pin and R_{out} is the internal resistance of the source. As we prefer to read a voltage and impose a current, we rewrite this equation as

$$i_{out} = (V_{out} - V_0)/R_{out}$$

and the HDL-A statement as

```
OUT_PIN.i %= (OUT_PIN.v - v0)/Rout;
```

Theoretically, an ideal voltage source cannot be described using this method since we want to impose a current which value would be undefined. However, this method must not be considered as absolute and we admit that, in some applications, it is still better to describe a pin the other way round—i.e., reading the current and imposing the voltage.

Polarization Pin

Some circuits (e.g., operational amplifiers) must be connected to an external current source in order to polarize the transistors. To model this particularity, a *polarization* pin is included in the model interface. One modelling solution is to consider it as a regular input pin. An alternative solution is to read the current directly on the pin.

Power Supply Stage

As a corollary of Kirchhoff's current law (KCL), the sum of the currents on all the pins of the model is always zero. A power supply block, where VDD is the positive power supply pin and VSS the negative one, must then be added to perform a balance-sheet of the currents according to the following principle:

> *All the current contributions that flow into the model go out of it through VSS, all the current contributions that flow out of the model come from VDD.*

Additionally, a loss current can be defined, which flows directly from VDD to VSS. This is illustrated in figure 3.3.

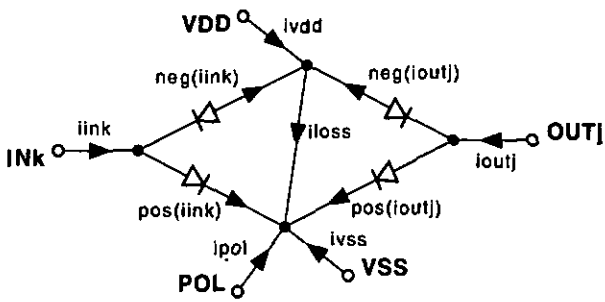


Fig. 3.3 Current balance-sheet.

We get the following equations

$$i_{vdd} = i_{loss} - \sum_j neg\left(i_{out_j}\right) - \sum_k neg\left(i_{in_k}\right)$$

$$i_{vss} = -i_{loss} - \sum_j pos\left(i_{out_j}\right) - \sum_k pos\left(i_{in_k}\right) - i_{pol}$$

where i_{loss} is the loss current, $neg(i_{out_j})$ is the negative part of the current on the output pin j, $neg(i_{in_k})$ is the negative part of the current on the input pin k, $pos(i_{out_j})$ is the positive part of the current on the output pin j, $pos(i_{in_k})$ is the positive part of the current on the input pin k and i_{pol} is the polarization current. An example of the resulting HDL-A code is given below

```
PROCEDURAL FOR dc, ac, transient =>
...
   if (iin>0.0) then
      n_iin := 0.0;
      p_iin := iin;
   else
      n_iin := iin;
      p_iin := 0.0;
   endif
   if (iout>0.0) then
      n_iout := 0.0;
      p_iout := iout;
   else
      n_iout := iout;
      p_iout := 0.0;
   endif
   VDD.i := iloss - n_iout - n_iin;
   VSS.i := -1.0*iloss - p_iout - p_iin - ipol;
```

Note that the modelling engineer is free to define models which do not follow Kirchhoff's laws or any other conservation law.

3.1.2 Model Parameters

In order for a model to be usable in various applications, the designer can define a set of parameters called `generics` in HDL-A. When an instance of a model is placed in a circuit description, particular values must be assigned to the different generics. Alternatively, default values, specified in the model code, are used. In any case, the value of a generic remains constant during a simulation. In HDL-A, a generic is declared in the entity part and is initialized to its default value in the initialization part of the architecture. In VHDL, however, the default value is set directly in the declaration statement. In the model code, a generic can appear in an equation or in the right hand side of an expression. It can also be used in a test clause. It cannot be used as the left-hand side of an assignment statement or as unknown in an equation.

As an example, the real parameter `rout` is declared, initialized and used in an assignment statement.

```
ENTITY ...
   GENERIC (rout : real);
...
END ENTITY...
ARCHITECTURE...
    PROCEDURAL FOR init =>
        rout := 100.0;
...
    PROCEDURAL FOR dc, ac, transient =>
        sig_out.i %= (sig_out.v - v0)/rout;
...
END ARCHITECTURE
```

3.2 Continuous Signal Processing

We have seen how to code the interface of the model. We now look at the heart of the model, where the desired behaviour is implemented. In this section, we will first show how to code models specified by transfer functions in the *s*-domain, expressed either using differential equations or pole-zero decomposition. Then, some typical continuous-time building blocks will be presented.

3.2.1 Generalities

First, the behaviour to implement is specified in an informal way. It can be a set of characteristic equations derived analytically from an existing component structure or extracted from tables of simulation results. It can also be described as an algorithm or given as a specification for a signal processing element. Lastly, it can be composed of a primary behaviour and a list of standard secondary effects—e.g., a slew-rate limitation. Sometimes, even if the model has a purely analogue interface, its behaviour can be considered as partly digital. Let us now look at the various possibilities offered to formalize the given behaviour.

In HDL-A, the behavioural code is split into two distinct parts—an analogue part and a digital part. The digital part is a single-process sub-set of VHDL. The analogue part is original; it is divided into `procedural` blocks and `equation` blocks, each block being valid for a particular analysis domain (i.e., DC, AC or transient analysis). A

digital behaviour can be described either in the digital part of the model, which is straightforward, or in the analogue part if the model code must remain purely analogue. In this text, we concentrate on analogue descriptions but, in some cases, the digital part will be used advantageously for the purpose of doing some comparisons.

According to the structure of HDL-A, our target HDL, the different specification styles enumerated above will lead to different code styles. Characteristic equations can be coded directly in the equation part of the model. If they are explicit, however, they may also be coded in the procedural part. An algorithmic description is coded in a procedural part using conditional or iterative statements. When the model description can be split into a primary behaviour and several secondary effects, the corresponding partial descriptions can be combined into a single model. In this context the terms *explicit* and *implicit* are used in their mathematical meaning. In a system of n variables x_1 to x_n, an explicit equation expresses x_i as a function of all the other variables except x_i itself while an implicit equation expresses x_i as a function of all the variables including x_i. Sometimes, however, modelling experts use the term *explicit* to define equations which are explicitly coded in the model and the term *implicit* to denote equations which are part of the simulation kernel.

The granularity of a hardware description language, which is basically given by the mathematical operators or by the standard functions defined in the Language Reference Manual (LRM), may be too fine. Therefore, it may be convenient to define reusable building blocks of higher level, which can be cascaded in order to build a complete model.

3.2.2 S-Domain Modelling

Very often, continuous systems are specified in the frequency domain by means of transfer functions. However, HDL-A, and presumably VHDL-AMS, offer time-domain description facilities only. Transfer functions must then be transformed into differential equations using the inverse Laplace transform as explained in, e.g., [Bah90].

In this discussion, we will concentrate on the coding of rational transfer functions with real coefficients. Furthermore, stability requires that the degree of the denominator is at least equal to the degree of the numerator. Two approaches will be presented. First, an equivalent differential equation is derived and coded in the model.

Secondly, the transfer function is decomposed into a product of simpler functions, each of which corresponds to either poles or zeros. The equivalent basic differential equations can then be coded and cascaded in order to build the complete behaviour model.

Differential Equations

The general form of a rational transfer function is

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\sum_{m=0}^{M} a_m s^m}{1 + \sum_{n=1}^{N} b_n s^n} \quad , \quad M \le N$$

which can be developed as

$$Y(s) + b_1 sY(s) + b_2 s^2 Y(s) + \ldots + b_{N-1} s^{N-1} Y(s) + b_N s^N Y(s)$$
$$= a_0 X(s) + a_1 sX(s) + a_2 s^2 X(s) + \ldots + a_{M-1} s^{M-1} X(s) + a_M s^M X(s)$$

and if we apply the inverse Laplace transform, we obtain

$$y(t) + b_1 \frac{dy(t)}{dt} + b_2 \frac{d^2 y(t)}{dt^2} + \ldots + b_{N-1} \frac{d^{N-1} y(t)}{dt^{N-1}} + b_N \frac{d^N y(t)}{dt^N}$$
$$= a_0 x(t) + a_1 \frac{dx(t)}{dt} + a_2 \frac{d^2 x(t)}{dt^2} + \ldots + a_{M-1} \frac{d^{M-1} x(t)}{dt^{M-1}} + a_M \frac{d^M x(t)}{dt^M}$$

which can be coded in HDL-A as an implicit equation according to the following pseudo-code.

```
PROCEDURAL FOR implicit_domain_name
   dx := ddt(x);
   d2x := ddt(dx);
   dm_1x := ddt(dm_2x);
   dmx := ddt(dm_1x);
   dy := ddt(y);
   d2y := ddt(dy);
   dn_1y := ddt(dn_2y);
   dny := ddt(dn_1y);
EQUATION (y) FOR implicit_domain_name
   y + b1*dy + b2*d2y + ... + bn_1*dn_1y + bn*dny
   == a0*x + a1*dx + a2*d2x + ... + am_1*dm_1x + am*dmx;
```

As only the first derivative operator is available, we use internal state variables to store the derivatives of $x(t)$ and $y(t)$. This leads to intermediate explicit equations, which can be coded in the procedural part of the model.

The same behaviour can also be coded in the integral form where initial conditions can be specified too. The resulting equation is as follows:

$$I_N[y(t)] + b_1 \cdot I_{N-1}[y(t)] + \ldots + b_{N-1} \cdot I_1[y(t)] + b_N \cdot y(t) =$$
$$a_0 \cdot I_N[x(t)] + a_1 \cdot I_{N-1}[x(t)] + \ldots + a_{M-1} \cdot I_{N-M+1}[x(t)] + a_M \cdot I_{N-M}[x(t)]$$

where

$$I_K[f(t)] = \int_{t=0}^{\infty} I_{K-1}[f(t)]dt + I_{K-1}[f(t)]\Big|_{t=0}$$

and

$$I_1[f(t)] = \int_{t=0}^{\infty} f(t)dt + f(0)$$

This representation is basically equivalent to the differential form but it is a less common modelling practice. Note that the output $y(t)$ depends on the integrals of $y(t)$ and of $x(t)$ only. With $M > N$, some derivatives of $x(t)$ would appear in the equation and the system could be unstable.

Poles and Zeros

A rational transfer function can also be decomposed into a product of poles and zeros as

$$H(s) = H_0 \frac{\prod_{m=1}^{M}(s - \zeta_m)}{\prod_{n=1}^{N}(s - p_n)} \quad , \quad M \le N$$

where ζ_m are the coordinates of the zeros and p_n the coordinates of the poles. For single zeros or poles, ζ_m, respectively p_n are real. Complex poles and zeros, however, are grouped into pairs of complex conjugates. Each member of those products can be modelled separately and the whole system is built by cascading all the sub-systems. For

each sub-system—i.e., single pole, single zero, pair of complex conjugate poles, pair of complex conjugate zeros—we will now start from the transfer function to establish the characteristic differential equation and the corresponding HDL-A code. In the following tables we give solutions for low-pass normalized sub-systems—i.e., $H(s=0)=1$.

Single real pole located at coordinates p in the s-plane	
Transfer function	$H(s)=\dfrac{-p}{s-p}$
Differential equation	$x(t)+\dfrac{1}{p}\cdot\dfrac{dy(t)}{dt}-y(t)=0$
HDL-A code	`x + 1.0/p*dy - y == 0.0;`

Table 3.1 HDL-A modelling of a aingle s-domain pole.

Pair of complex conjugate poles p, p^* at $\alpha_p \pm j\beta_p$	
Tranafar function	$\dfrac{p\cdot p^*}{(s-p)\cdot(s-p^*)}$
Differential equation	$x(t)-\dfrac{1}{\alpha_p{}^2+\beta_p{}^2}\cdot\dfrac{d^2y(t)}{dt^2}$ $+\dfrac{2\alpha_p}{\alpha_p{}^2+\beta_p{}^2}\cdot\dfrac{dy(t)}{dt}-y(t)=0$
HDL-A code	`x - 1.0/(ap*ap+bp*bp)*d2y` `+ 2.0*ap/(ap*ap+bp*bp)*dy - y == 0.0;`

Table 3.2 HDL-A modelling of a pair of s-domain poles.

Single real zero at ζ	
Transfer function	$H(s) = -\frac{1}{\zeta}(s-\zeta)$
Differential equation	$y(t) = x(t) - \frac{1}{\zeta} \cdot \frac{dx(t)}{dt}$
HDL-A code	`y := x - 1.0/z*dx;`

Table 3.3 HDL-A modelling of a single *s*-domain zero.

Pair of complex conjugate zeros ζ, ζ^* at $\alpha_z \pm j\beta_z$	
Transfer function	$H(s) = \frac{1}{\zeta \cdot \zeta^*}(s-\zeta) \cdot (s-\zeta^*)$
Differential equation	$y(t) = x(t) - \frac{2\alpha_z}{\alpha_z^2 + \beta_z^2} \cdot \frac{dx(t)}{dt} + \frac{1}{\alpha_z^2 + \beta_z^2} \cdot \frac{d^2x(t)}{dt^2}$
HDL-A code	`y := x - 2.0*az/(az*az+bz*bz)*dx` `+ 1.0/(az*az+bz*bz)*d2x;`

Table 3.4 HDL-A modelling of a pair of *s*-domain zeros.

Note that the poles are described by implicit differential equations because the derivatives of the output appear in the equation. They must be coded in the equation part of the HDL-A model. The equations which describe the zeros, however, depend only on the input and its derivatives. Thus, they are of explicit form and can be coded in a procedural block. However, as the degree of the numerator is higher than the degree of the denominator, zero-only sub-systems are not stable. Consequently, they should theoretically be always combined with other sub-systems (i.e., poles) in order to finally obtain a stable system because the simulator sees the combination of sub-systems as one single system. However, as some simulators do not accept input signals with infinite slope—i.e., input signals are always

bounded—systems with $M > N$ can still be simulated under specific conditions.

Additionally, it is possible to define a cell composed of one pole p and one zero ζ. The corresponding characteristic equations are given in table 3.5. If p, respectively ζ, tends to infinite, this cell becomes a single zero cell, respectively a single pole cell.

One-pole-one-zero cell; the coordinates are p and ζ respectively	
Transfer function	$H(s) = \frac{p}{\zeta} \cdot \frac{s-\zeta}{s-p}$
Differential equation	$y(t) - \frac{1}{p} \cdot \frac{dy(t)}{dt} = x(t) - \frac{1}{\zeta} \cdot \frac{dx(t)}{dt}$
HDL-A code	`y - 1.0/p*dy == x - 1.0/z*dx`

Table 3.5 HDL-A modelling of a s-domain one-pole-one-zero cell.

Similarly, table 3.6 describes a bi-quadratic cell composed of a pair of poles and a pair of zeros. Again, if p and p^*, respectively ζ and ζ^* tend to infinite, we obtain the characteristic equation developed for a pair of zeros or a pair of poles respectively.

Bi-quadratic cell; the coordinates are p, p^* and ζ, ζ^*	
Transfer function	$H(s) = \frac{p \cdot p^*}{\zeta \cdot \zeta^*} \cdot \frac{(s-\zeta) \cdot (s-\zeta^*)}{(s-p) \cdot (s-p^*)}$
Differential equation	$\frac{1}{\alpha_p^2 + \beta_p^2} \cdot \frac{d^2y(t)}{dt^2} - \frac{2\alpha_p}{\alpha_p^2 + \beta_p^2} \cdot \frac{dy(t)}{dt} + y(t) = \frac{1}{\alpha_z^2 + \beta_z^2} \cdot \frac{d^2x(t)}{dt^2} - \frac{2\alpha_z}{\alpha_z^2 + \beta_z^2} \cdot \frac{dx(t)}{dt} + x(t)$

HDL-A code	`1.0/(ap*ap+bp*bp)*d2y`
	`- 2.0*ap/(ap*ap+bp*bp)*dy + y ==`
	`1.0/(az*az+bz*bz)*d2x`
	`- 2.0*az/(az*az+bz*bz)*dx + x;`

Table 3.6 HDL-A modelling of a s-domain bi-quadratic cell.

3.2.3 Miscellaneous Continuous-Time Building Blocks

Besides poles and zeros, other building blocks can be defined. As an illustration of the use of an analogue HDL, we will give some code solutions for limitations and delays. These particular elements of behaviour are mainly considered as secondary effects, that complement the primary, ideal description of a system.

Limitations

Procedural languages such as HDL-A or VHDL are very convenient to model a limitation defined as a linear (unity gain) behaviour if the input is between two bounds and as a saturation farther as shown in figure 3.4.

The first solution that comes to mind is to use a simple conditional statement. If `x` is the input variable, `y` the output variable and `inf` and `sup` are the lower, respectively upper bounds, the pseudo-code can be expressed as

```
IF (x > sup) THEN y := sup;
ELSIF (x < inf) THEN y := inf;
ELSE y := x;
END IF;
```

In many cases, however, this solution poses a problem to the simulator because the characteristic response—i.e., the output expressed as a function of the input—is continuous but its first derivative is not (figure 3.4a). This can induce some oscillations in the iterative solving algorithm used by the simulator when `x` is near to one limit. To by-pass this problem, another variant can be proposed which exhibits a smooth characteristic (figure 3.4b). This model is based on a hyperbolic tangent centred between two asymptotes located at *sup* and *inf*, and which has a slope equal to 1 at x=0. This behaviour can be expressed as the following expression

$$y(t) = \frac{sup - inf}{2} \tanh\left(\frac{2x(t)}{sup - inf}\right) + \frac{sup + inf}{2}$$

and as the following pseudo-code

```
arg := 2.0*x/(sup-inf);
y := ((sup-inf)/2) * ((exp(2.0*arg)-1.0)/(exp(2.0*arg)+1.0)
   + ((sup+inf)/2);
```

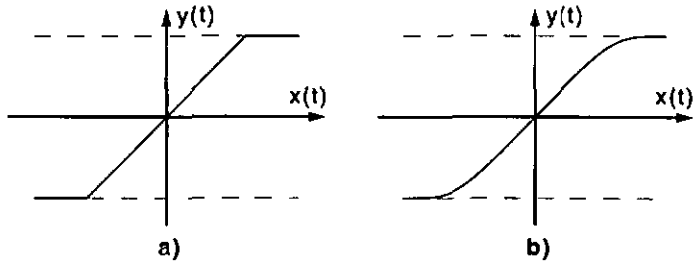


Fig. 3.4 Limitation unit: a) discontinuous b) smooth.

This shows how we can take advantage of the various aspects of an analogue HDL. In a first simple version, the behaviour is expressed using conditional statements. In a second version, an equation is coded. Many other versions of a limitation function can be developed. Some of them are for general use while some others are more specialized. In a practical example, a double limitation was introduced to model at the same time the output current limitation and the maximum swing of an operational transconductance amplifier (OTA) [EPF96].

Delay Elements

Using the same differential equation as was used to describe a pole, an exponential delay element can be modelled.

$$x(t) - A \cdot \frac{dy(t)}{dt} - y(t) = 0 \quad , \quad A > 0$$

If the input of this element is a unit step at time T

$$x(t) = u(t - T)$$

the output will be of the form

$$y(t) = u(t-T)\left(1 - e^{-\frac{1}{A}(t-T)}\right)$$

If we now define the delay τ as the time interval between T and the cross point of the tangent of the curve

$$1 - e^{-\frac{1}{A}(t-T)}$$

at T with the straight line y=1, we have τ=A (figure 3.5).

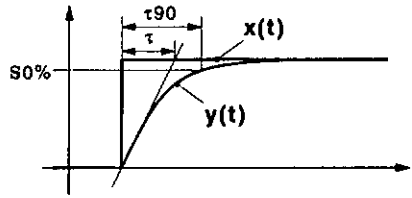


Fig. 3.5 Exponential delay.

This behaviour can be implemented as

```
x - tau * ddt(y) - y == 0.0;
```

We can also define the delay τ_{90} as the time needed by the output signal to reach 90% of its final amplitude. In this case we have τ_{90}=A*ln10, which leads to the following pseudo-code.

```
x - (tau90/ln(10.0)) * ddt(y) - y == 0.0;
```

3.3 Sampled-Data Modelling

As explained in the introduction, this work is basically limited to the description of analogue models. This means that, first, the interface of the model is composed of analogue pins only—there are no digital ports—second, the model will be simulated using analogue simulation algorithms. This type of representation is well suited to actual analogue systems, which process signals that can vary continuously in time and in amplitude. Digital systems, however, which process digital signals that have a quantified amplitude and that vary only at particular points in time, are better described by digital models and are better simulated using an event-driven simulator. Therefore, they are not covered in this work.

In this section, however, we will concentrate on an intermediate class of systems called *sampled-data systems*. They process sampled-

data signals that have a continuous amplitude but that are only defined at particular points in time. Such signals can be described as series of real numbers called *samples*. Sampled-data systems are typically used to describe switched-capacitor circuits.

Here, the interface of the sampled models will still be considered analogue, but the signal processing part will be either coded in the analogue part of the model and simulated by an analogue simulator, or coded in the digital part of the model and simulated using an event-driven algorithm.

If the sampling is periodic in time and synchronous for all signals, the system can be described by a difference equation or by a z-domain transfer function. We will first see how to code a difference equation in the analogue part of the model code, then in the digital part. Second, we will derive the difference equations that correspond to elementary z-domain cells.

Another way of describing a sampled-data system is to define an equivalent continuous-time system and to post-process the output with a sample-delay-hold (S/D/H) element. This will be described too. Finally, we will see how to model systems described by algorithms.

3.3.1 Sampling and Shifting Operations

Let us see now how to implement two operations that are key issues in sampled-data modelling: the sampling operation and the shifting operation. Some solutions will be developed to code them in both the digital and the analogue domain. In any case, we assume that the input signal $x(t)$ is band-limited according to the sampling theorem.

In the Digital Part

In the digital part of an HDL-A model, like in VHDL, the information is carried by objects called `signals`. A signal is defined at discrete points on the time axis. Its amplitude can be either quantified or not. A signal is computed according to an algorithm coded in a `process` block and controlled by a `wait` statement. A signal takes its actual value after the wait statement and the corresponding algorithm have been completely executed.

In other words, we have the following sequence:

(1) The wait statement waits for an event or a time-out.

(2) When the event or time-out occurs, the algorithm is executed and the new value of each signal involved is computed using old signal values and optional additional variables.

(3) After the algorithm has been completely executed, the newly computed values are assigned to the signals.

These values will then be held until the next execution of the wait statement. The new signal values can also trigger some other processes, which are executed again until all the signal values are stable.

In addition to signals, auxiliary variables can also be used but, unlike signals, variable assignments take effect immediately. The value of an analogue state variable can also be used in an algorithm, but it cannot be assigned.

As our aim is to model sampled-data systems, we make the following choices:

- In order for the signals to match our definition of sample-data signals, they will not be quantified.
- The wait statement will depend only on the clock and not on the results of any other process. Note that each HDL-A model can include only one process block.

We are then sure that the whole system is open-loop and that each wait statement will be executed only once per clock cycle. As a corollary, if we read an analogue state variable inside the wait loop, it will be read only once per cycle, realizing hence the sampling operation. Furthermore, the value of the previous sample of each signal is available during the computation of the algorithm. This way, a shifted copy of a signal can be obtained just by assigning its value to another signal.

The following code example illustrates the sampling of the input state variable $x(t)$ and the shifting of the intermediate signal $x(k)$.

```
PROCESS
...
LOOP
   WAIT ON ...;
      x1 <= x;     -- sampling of the state x into the signal x1
      x2 <= x1;    -- shifting of the signal x1 into the signal x2
      ...
```

```
END LOOP;
END PROCESS;
```

In the Analogue Part

The analogue part of a model will be evaluated using the mechanisms of analogue simulation. Any equation or piece of procedural code is considered as being always satisfied and can be evaluated at any time depending on the way the simulator manages the time. Unlike in the digital part, there are neither trivial sampling nor shifting mechanism in the analogue part of HDL-A. We will have to implement both of them.

The sampling and shifting operations must be synchronized to the clock signal. A conditional statement controls the execution of a particular block in which the analogue input signal will be sampled and some sampled signals will be shifted by an integer number of clock periods. For example, the condition can be set to the rising edge of the clock signal.

```
PROCEDURAL FOR transient =>
...
if rising(vclk,threshold) then
```

The sampling operation is simply carried out as an assignment statement. The instantaneous value of the continuous input variable $x(t)$ is copied into another state variable $x_0(t)$ at sampling time.

```
x0 := x;
```

The sampling period is then calculated as the difference between the current time and the last sampling time, which is stored in a state variable tmem.

```
tsam := (current_time - previous(tmem));
tmem := current_time;
```

Then, some sampled signals must be shifted by one or more sampling periods. This is done using the function previous of HDL-A.

```
xm := previous(x0,(m*tsam));
```

These sampling and shifting mechanisms can now be used to model sampled-data behaviour given either by difference equations or by algorithms.

3.3.2 Coding of a Difference Equation

In time-domain description of sampled-data systems, the current value of a signal is computed as a function of its past values and of the current and past values of other signals. In order to simplify the representation, we will limit the discussion to linear systems described by difference equations with constant coefficients of the form

$$y(k) = \sum_{m=0}^{M} a_m x(k-m) - \sum_{n=1}^{N} b_n y(k-n)$$

where k is the current sample. As $m \geq 0$, the system is always causal.

In HDL-A, which is a mixed analogue-digital language, such an equation can be coded either in the analogue part or in the digital part of a model.

Coding in the Digital Part

According to the sampling and shifting facilities defined above, we now code the difference equation in the digital part of an HDL-A model.

Terminology: k is the index that identifies the k-th sample of a signal. t_k is the value of the time when the sample k is computed. For a sampled signal the k-th sample is expressed as $x(k)$, for an analogue state variable we write $x(t_k)$ the value of $x(t)$ at time t_k.

Terminology: at time t_k, we distinguish $x^-(k)$ which is the value of the signal x at the beginning of the wait loop and $x^+(k)$, which is the value of x after the loop has been computed. To simplify the notation, we write $x(k)$ for $x^+(k)$ and $x(k-1)$ for $x^-(k)$ because $x^-(k)$ is the value $x^+(k-1)$, which has not been modified since time t_{k-1}.

We will now have to express each term of the difference equation as an HDL-A object.

$x(k)$ — We take the value $x(t_k)$ of the input analogue state variable $x(t)$, thus realizing the sampling of the entry.

$x(k-1)$ — We assign the value of the state variable $x(t_k)$ to a signal $x_1(k)$. At time t_l, $l=k+1$, during the evaluation of $y(l)$, we can access $x_1(k)$ which is actually $x(t_{l-1})$.

$x(k-2)$ We assign the value of the signal $x_1(k-1)$ to a signal $x_2(k)$. At time t_l, $l=k+1$, during the evaluation of $y(l)$, we can access $x_2(k)$ which is actually $x_1(k-1)$ and also $x(t_{l-2})$.

$x(k-m)$ The same way, we assign the value of the signal $x_{m-1}(k-1)$ to a signal $x_m(k)$.

$y(k)$ This is the output signal that must be computed.

$y(k-1)$ This value is clearly available. During the evaluation of $y(l)$, $l=k+1$, we can access $y(k)$ which is also $y(l-1)$.

$y(k-2)$ We assign the value of the signal $y(k-1)$ to a signal $y_2(k)$. During the evaluation of $y(l)$, $l=k+1$, we have $y_2(k)$ which is also $y(k-1)$ and also $y(l-2)$.

$y(k-3)$ We assign the value of the signal $y_2(k-1)$ to a signal $y_3(k)$. During the evaluation of $y(l)$, $l=k+1$, we have $y_3(k)$ which is also $y_2(k-1)$ and also $y(k-2)$ and also $y(l-3)$.

$y(k-n)$ The same way, we assign the value of the signal $y_{n-1}(k-1)$ to a signal $y_n(k)$.

We get the following pseudo-code.

```
PROCESS
...
LOOP
   WAIT ON rising(vclk,threshold);
      x1 <= x;
      x2 <= x1;
      ...
      xm <= x[m-1];
      y <= a0*x + a1*x1 + ... + am*xm - b1*y - ... - bn*yn;
      y2 <= y;
      y3 <= y2;
      ...
      yn <= y[n-1];
END LOOP;
END PROCESS;
```

The output signal $y(k)$ can be used as input of an analogue output stage or in any other part of the model. Sometimes, $y(k)$ must be available immediately in the same computation cycle. This is the case, for instance, when we cascade two difference equations: the input of the second equation is the output of the first one. We must then

introduce a new intermediate variable $w(k)$, which immediately takes the computed value and which is available inside the model for further computations. The pseudo-code becomes

```
w := a0*x + a1*x1 + ... + am*xm - b1*y - ... - bn*yn;
y <= w;
```

Thanks to this mechanism, a difference equation can also be decomposed into elementary equations as we will see later on.

Coding in the Analogue Part

Here, we evaluate the response $y(t_k)$, which is now an analogue state variable, in an analogue `relation` block. The different variables of the equation are represented by `state` variables. The sampling and shifting operations are implemented as already explained above.

$x(k)$	$x(t)$ is sampled, its value being copied into a state variable x_0.
$x(k-1)$	x_0 is shifted by one sampling period, and copied into a state variable x_1.
$x(k-m)$	x_0 is shifted by m sampling periods, and copied into a state variable x_1.
$y(k)$	Output state variable to compute.
$y(k-1)$	y is shifted by one sampling period, and copied into a state variable y_1.
$y(k-n)$	y is shifted by n sampling periods, and copied into a state variable y_n.

As all the sampled and shifted state variables hold a constant value during one clock cycle, the actual equation can be coded outside the clock conditional block.

We get the following pseudo-code.

```
PROCEDURAL FOR transient =>
...
if rising(vclk,threshold) then
   x0 := x;
   tmem := current_time;
   tsam := (current_time - previous(tmem));
```

```
    tdel := real(tsam);
    x1 := previous(x0,tdel);
    ...
    xm := previous(x0,(m*tdel));
    y1 := previous(y,tdel);
    ...
    yn := previous(y,(n*tdel));
end if;
y := a0*x0 + a1*x1 + ... + am*xm - b1*y1 - ... - bn*yn;
```

3.3.3 *Z*-Domain Elementary Cells

We have seen above how to code a difference equation either in the digital part or in the analogue part of a model. The same specification expressed as a rational z-transfer function can also be decomposed as a product of elementary cells. To obtain the same behaviour, the corresponding elementary difference equations will be cascaded. In this section, the z^{-1} formalism has been chosen because it is commonly used in the digital signal processing community and because it leads to elementary cells which can be realized easily. More informations on the z-transform and on the description of digital systems can be found in [Kuc88].

The z-transform of the difference equation coded previously gives

$$Y(z) = X(z)\sum_{m=0}^{M} a_m z^{-m} - Y(z)\sum_{n=1}^{N} b_n z^{-n}$$

The corresponding transfer function is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{m=0}^{M} a_m z^{-m}}{1 + \sum_{n=1}^{N} b_n z^{-n}}$$

which can be developed as

$$H(z) = H_0 \frac{\prod_{m=1}^{M} \left(1 - \zeta_m z^{-1}\right)}{\prod_{n=1}^{N} \left(1 - p_n z^{-1}\right)}$$

ζ_m are the coordinates of the M zeros of $H(z)$ located at $z\neq 0$ in the z-plane and p_n are the coordinates of the N poles of $H(z)$ located at $z\neq 0$ in the z-plane. In such an expression, the number of poles always equals the number of zeros. This means that the system also includes $M-N$ poles at $z=0$ if $M>N$ or $N-M$ zeros at $z=0$ if $M<N$.

We now develop elementary cells that can be cascaded in order to build such a transfer function. In each case H_0 has been set in such a way that we have $H(z=1)=1$. The difference equation is then coded in HDL-A in both the digital and the analogue versions.

Elementary cell $\dfrac{1}{1-pz^{-1}}$.

This corresponds to a pole located at the real coordinates p in the z-plane coupled with a zero on the origin. We have:

Transfer function	$H(z)=(1-p)\dfrac{1}{1-pz^{-1}}=(1-p)\dfrac{z}{z-p}$
Difference equation	$y(k)=(1-p)x(k)+py(k-1)$
Digital HDL-A code	`y <= (1.0-p)*x + p*y`
Analogue HDL-A code	`y := (1.0-p)*x0 + p*y1`

Table 3.7 Elementary z-domain cell: pole combined with a zero on the origin.

Elementary cell $\dfrac{1}{(1-pz^{-1})(1-p^{*}z^{-1})}$

p, p^{*} are a pair of complex conjugate poles located at $\alpha_p \pm j\beta_p$. These poles are combined with a double zero on the origin.

Tranafer function	$H(z)=(1-p)(1-p^*)\dfrac{1}{(1-pz^{-1})(1-p^*z^{-1})}$ $=(1-p)(1-p^*)\dfrac{z^2}{(z-p)(z-p^*)}$
Difference equation	$y(k)=(1-2\alpha_p+\alpha_p{}^2+\beta_p{}^2)x(k)$ $+2\alpha_p y(k-1)-(\alpha_p{}^2+\beta_p{}^2)y(k-2)$
Digital HDL-A code	`y <= (1.0-2.0*ap+ap**2+bp**2)*x + 2.0*ap*y - (ap**2+bp**2)*y2;`
Analogue HDL-A code	`y := (1.0-2.0*ap+ap**2+bp**2)*x0 + 2.0*ap*y1 - (ap**2+bp**2)*y2;`

Table 3.8 Elementary z-domain cell: pair of complex conjugate poles combined with a double zero on the origin.

Elementary cell $(1-\zeta z^{-1})$

This transfer function represents a single zero at real coordinates ζ coupled with a pole on the origin.

Transfer function	$H(z)=\dfrac{1}{1-\zeta}(1-\zeta z^{-1})=\dfrac{1}{1-\zeta}\cdot\dfrac{z-\zeta}{z}$
Difference equation	$y(k)=\dfrac{1}{1-\zeta}(x(k)-\zeta x(k-1))$
Digital HDL-A code	`y <= (x - z*x1)/(1.0-z)`
Analogue HDL-A code	`y := (x0 - z*x1)/(1.0-z)`

Table 3.9 Elementary z-domain cell: zero combined with a pole on the origin.

Elementary cell $(1-\zeta z^{-1})(1-\zeta^* z^{-1})$

ζ, ζ^* are a pair of complex conjugate zeros located at $\alpha_z \pm j\beta_z$. These zeros are combined with a double pole on the origin.

Transfer function	$H(z)=\frac{1}{(1-\zeta)(1-\zeta^*)}(1-\zeta z^{-1})(1-\zeta^* z^{-1})$ $=\frac{1}{(1-\zeta)(1-\zeta^*)}\cdot\frac{(z-\zeta)(z-\zeta^*)}{z^2}$
Difference equation	$y(k)=\frac{1}{(1-2\alpha_z+\alpha_z{}^2+\beta_z{}^2)}\cdot$ $(x(k)-2\alpha_z x(k-1)+(\alpha_z{}^2+\beta_z{}^2)x(k-2))$
Digital HDL-A code	`y <= (x-2.0*az*x1+(az**2+bz**2)*x2)/ (1.0-2.0*az+az**2+bz**2);`
Analogue HDL-A code	`y := (x0-2.0*az*x1+(az**2+bz**2)*x2)/ (1.0-2.0*az+az**2+bz**2);`

Table 3.10 Elementary z-domain cell: pair of complex conjugate zeros combined with a double pole on the origin.

These cells are either poles combined with zeros at z=0 or zeros combined with poles at z=0. If we connect them in cascade to build the transfer function of a more complex system, the exceeding singularities we introduced at z=0 cancel each other.

If we now want to model a z-domain transfer function of the form

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{m=0}^{M}\alpha_m z^m}{\sum_{n=0}^{N}\beta_n z^n}\quad,\quad M\le N$$

it is possible to rewrite it as

$$H(z)=\frac{z^{-N+M}\cdot\sum_{m=0}^{M}\alpha_m z^{-M+m}}{\sum_{n=0}^{N}\beta_n z^{-N+n}}=H_0\cdot z^{-L}\frac{\prod_{m=1}^{M}(1-\zeta_m z^{-1})}{\prod_{n=1}^{N}(1-p_n z^{-1})}\quad,\quad L=N-M$$

In this formulation, M must be smaller than N to ensure that the system is causal.

The corresponding time-domain model can then be built as a cascade of difference equations according to the elementary cells

developed above. In order to also include the z^{-L} contribution, the input signal $x(k)$ has to be shifted by L clock cycles implementing

$$H(z) = z^{-L} \Rightarrow w(k) = x(k-L)$$

in HDL-A.

In other words, we added N zeros and M poles on the origin while implementing the N poles p_n and the M zeros ζ_m of the transfer function using our modelling method. M pole-zero pairs cancel each other. The supplementary term z^{-L} adds L poles on the origin to cancel the N–M remaining zeros in order to actually implement the desired transfer function $H(z)$.

The decomposition of a transfer function in elementary pole-zero cells is very useful when the location of the singularities is known. However, filters given by their coefficients a_m and b_n can be better modelled directly with a difference equation as described in sect. 3.3.2. This is particularly true for FIR filters, because all b_n are null.

3.3.4 Modelling with a Sample-Delay-Hold Element

In addition to the methods presented above a sampled-data system can also be modelled using a continuous-time system followed by an *sample-delay-hold* element (S/D/H) as proposed by Tsividis [Tsi83]. This is shown in figure 3.6. In order for this system to model a sampled-data system correctly, the input signal must be bound in frequency according to the sampling theorem.

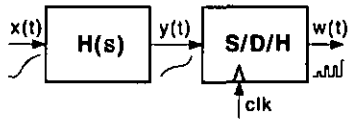


Fig. 3.6 Signal processing chain with an S/D/H.

The Sample-Delay-Hold

The S/D/H element is characterized by three pins—the input, the output and the clock—and by four parameters—a threshold for the evaluation of the clock edge, a delay τ_a between this edge and the actual sampling, a delay τ_b between the sampling and the appearance

of the sampled value on the output and a hold time τ_c. Figure 3.7 illustrates the behaviour of this element.

To implement the S/D/H, we first create shifted copies of the clock signal. These state variables `clka`, `clkb` and `clkc`, are normalized between 0 and 1 and shifted respectively by τ_a, $\tau_a+\tau_b$ and $\tau_a+\tau_b+\tau_c$. First, the input state variable is sampled at each rising edge of `clka`, in order to realize the delayed sampling. Then the resulting state variable is sampled with `clkb` in order to shift the sampled state variable by τ_b. The delayed state variable is finally delivered to the output when `clkb` is high and `clkc` is low.

```
PROCEDURAL FOR transient =>
...
clka := previous(clk,taua);    -- clock delayed by taua
clkb := previous(clk,tauab);   -- clock delayed by taua+taub
clkc := previous(clk,tauabc); -- clock delayed by taua+taub+tauc
if rising(clka,0.5) then sama := in;     -- sampling of the input
end if;                                  -- after taua
if rising(clkb,0.5) then samb := sama;   -- sampling of sama
end if;                                  -- after taub
if (tauc=0.0) then cond := 1.0;
else cond := clkb * (1.0 - clkc);
end if;
if (cond>0.5) then out := samb * cond;
else out := 0.0;
end if;
```

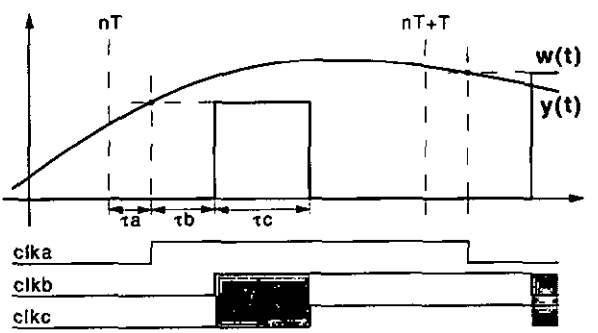


Fig. 3.7 Sample-delay-hold behaviour.

3.3.5 Coding of an Algorithm

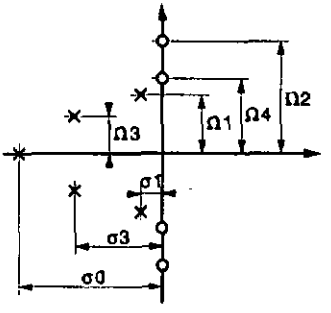
Obviously, a sampled-data system whose behaviour is given as an algorithm can be easily coded in HDL-A either using the digital code facilities or in the procedural part of the analogue portion. In both cases, the sampling of the input signal and the shift operation are realized as explained for the coding of difference equations. The algorithm itself is then coded using conditional or loop statements, making advantageously use of the facilities offered by the language. An example of algorithm coding is provided in chapter 7.

3.4 Coding Example

To illustrate the coding method exposed in this chapter, a low-pass filter is modelled in HDL-A and simulated. We start from the well-known CCITT G712 PCM specification summarized in table 3.11 [Cor92].

Frequency band	Frequency	Attenuation
Passband	$< f_p = 3$ kHz	$\alpha_p = \pm 0.125$ dB
Stopband	$> f_{s1} = 4$ kHz	$\alpha_{s1} = -14$ dB
	$> f_{s2} = 4.6$ kHz	$\alpha_{s2} = -32$ dB

Table 3.11 Low-pass filter specifications.



σ0 = -12.836E3
σ1 = -1.9600E3
σ3 = -7.7402E3
Ω1 = 20.037E3
Ω2 = 39.937E3
Ω3 = 15.102E3
Ω4 = 27.136E3

Fig. 3.8 Pole-zero location of a fifth order elliptic transfer function.

In [Cal94], a solution is given in the form of a fifth order elliptic filter. The transfer function comprises 5 poles and 4 zeros as represented in figure 3.8.

The complete HDL-A code is given in appendix B but we explain some code extracts with more details here. To implement this transfer function in HDL-A, we code the differential equations that correspond to the single pole, to the two pairs of poles and to the two pairs of zeros according to § 3.2.1 and we place them in series as

```
PROCEDURAL FOR dc, ac, transient =>
    dint2 := ddt(int2);
    dint3 := ddt(int3);
    int4 := int3 + a24 * ddt(dint3);
    dint4 := ddt(int4);
    int5 := int4 + a25 * ddt(dint4);
    ...
EQUATION (int1,int2,int3) FOR dc, ac, transient =>
    vin - b11 * ddt(int1) - int1 == 0.0;
    int1 - b12 * dint2 - b22 * ddt(dint2) - int2 == 0.0;
    int2 - b13 * dint3 - b23 * ddt(dint3) - int3 == 0.0;
    ...
```

where the intermediate variables `b11`, `b12`, `b22`, `b13`, `b23`, `a24` and `a25` were introduced to save some computation time. They are functions of the generics `sigma0`, `sigma1`, `omega1`, `sigma3`, `omega3`, `omega2` and `omega4`. For example, we have

$$b_{12} = \frac{-2 \cdot 2\pi \cdot \sigma_3}{(2\pi \cdot \sigma_3)^2 + (2\pi \cdot \omega_3)^2}$$

which gives

```
PROCEDURAL FOR init =>
...
   b12 := (-2.0*sigma3)/(twopi*(sigma3*sigma3+omega3*omega3));
```

The remainder of the model is composed of an interface which includes a capacitive input stage—the input resistance is assumed to be infinite—a resistive output stage and a power supply block.

The model is then placed in a simulation environment and an AC analysis is performed to verify the frequency behaviour. In order for our model to match the specifications of the filter, the coordinates of the poles and zeros are denormalized. In our example, a multiplying 2π factor has been included in the model so that the coordinates must

be multiplied by the cut-off frequency only—3 kHz in this case. The result of the simulation is displayed in figure 3.9 as a plot of the amplitude and phase of the output signal vs. frequency.

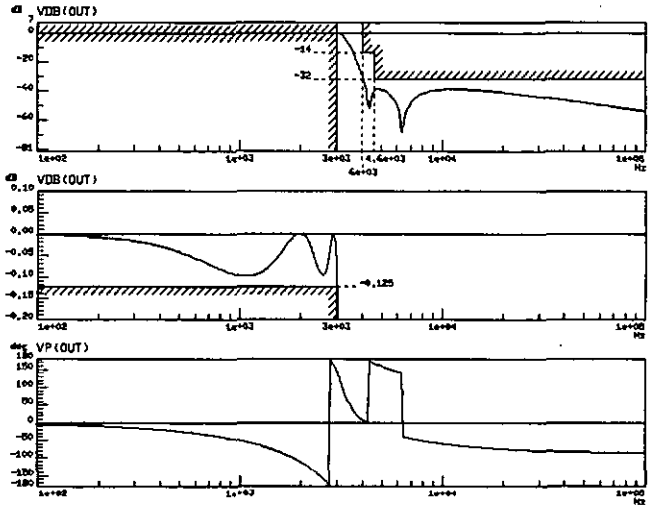


Fig. 3.9 Simulation of a fifth order elliptic filter.

3.5 Conclusions

With the aim of introducing the user to the field of analogue behavioural modelling and to the coding of AHDL models, we presented here a modelling approach based on a *divide-and-conquer* strategy. We claim that the behaviour to model can be split into several parts, which can be handled separately and we propose AHDL code solutions for the most classical ones—input and output stages, continuous-time and sampled-data transfer functions, delays and limitations.

This method can be used very successfully in many practical cases. However, when the elements of behaviour are too tightly coupled, a more global approach should be adopted, which is more difficult to formalize and which depends on each particular application.

Whatever the method, we notice that the coding process requires a good knowledge of the modelling language syntax. It also requires a good understanding of the operation of an analogue or mixed

analogue-digital simulator. Therefore we consider that some modelling support must be provided to the user. This will be the subject of the two next chapters that deal with graphical modelling and with automatic AHDL code generation.

3.6 References

[ANA94a] *HDL-A User's Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

[Bah90] H. Baher, *Analog & Digital Signal Processing,* John Wiley & Sons, Chichester, 1990.

[Boi83] R. Boite and J. Neirynck, *Théorie des Réseaux de Kirchhoff,* (in French), Traité d'Électricité, Vol. IV, Presses polytechniques romandes, Lausanne, 1983.

[Cal94] C. Calame, *Modélisation Comportementale de Filtres Analogiques à Temps Continu,* (in French), Projet de semestre hiver 1993/94, Ecole polytechnique fédérale de Lausanne, Laboratoire de microtechnique EPFL - UNI NE, Neuchâtel, 1994.

[Cor92] F. Corthay, *Oversampled digital leapfrog filters,* Thèse, Université de Neuchâtel, Institut de microtechnique, Neuchâtel, 1992.

[EPF96] *Mixed-Mode Modelling and Simulation,* MICROSWISS IEEE Workshop, EPFL IMT CSEM, Lausanne, 1996.

[Knu88] R. Knu, *Introduction to Digital Signal Processing,* McGraw-Hill Book Company, New York, 1988.

[Nag75] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* ERL Memo ERL-M520, University of California, Berkeley, 1975.

[Shi95] C.-J. R. Shi and A. Vachoux, "VHDL-A Design Objectives and Rationale", *Modelling in Analog Design,* pp. 1-30, J.-M. Bergé, O. Levia and J. Rouillard (eds.), Kluwer Academic Publisher, Boston, 1995.

[Tsi83] Y. Tsividis, "Principles of Operation and Analysis of Switched-Capacitor Circuits", *Proceedings of the IEEE,* Vol. 71, pp. 926-940, 1983.

Chapter 4
Graphical Description

In the previous chapter, we saw how to model analogue behaviour by means of an analogue hardware description language. Here, we propose to formalize the same knowledge in a graphical form. This representation can be seen as a first step towards an automatic code generation process, which will be exposed in chapter 5. It can also serve documentation purposes.

In the introduction, the specifications for such a graphical description of analogue behaviour are established, the existing graphical description methods for continuous-time systems are presented and our particular approach is briefly exposed. Then, the different elements of our description method are explained with more details in the subsequent sections. Finally, a short example is presented.

This method takes into account the modelling knowledge established previously. However, it has been designed to remain of general use and independent of any particular hardware description language.

4.1 Introduction

As seen in chapter 3, modelling with an Analogue Hardware Description Language (AHDL) requires a very good knowledge of the language syntax. As such, this task is unfortunately reserved to specialists. If we want to promote analogue behavioural modelling among designers, computer-aided design tools with convenient user

interface must be provided. Ideally, such tools should be integrated in the design environment that is already used for circuit design. The same requirements were also valid for digital modelling. VHDL and Verilog have been accepted and used successfully by the digital design community when computer-aided modelling tools have been available, although automatic logic synthesis tools have played a decisive role in this too. Obviously, these tools have significantly contributed to the wider acceptance of high-level modelling and of top-down design in general. Furthermore, it has been shown that the overall design productivity is improved by the use of graphical HDL-based design tools [Jos95].

As some analogue designers may be unwilling to learn yet another computer language, it is necessary to provide them with a graphical modelling interface. A graphical description offers additional advantages: it is easier to understand than a text file and can therefore serve as a better communication channel among designers. *A good drawing says more than a thousand words.* In other respects, the readability of a graphical model description can also be improved using hierarchy. The modelling know-how can also be stored directly in the graphical form and reused easily.

4.1.1 Specifications of the Graphical Description Set

In order to model dynamic systems accurately, a graphical description set must include the following features:

- Description of the structure of a system as the interconnection of components through a physical—i.e., across-through—interface.
- Description of the behaviour of a system or component, using non-physical variables for the information propagation and processing.
- Description of generic models including a set of parameters.
- Aptitude to an implementation as a front-end to a computer program.

4.1.2 Existing Graphical Description Methods

Before designing a new graphical description set, we review several existing graphical modelling techniques, each of which is discussed

and compared with our specifications. More information on these description techniques can be found in [Cel91].

Circuit Diagram Modelling

Circuit diagram modelling is well-known; it has been widely used by electrical and electronic engineers for years. The basic components of electrical engineering (i.e., resistors, transistors, sources, etc.) are available as symbols, each of which is composed of a body, pins and a set of parameters. These basic components are linked to an implicit model, which can be as simple as Ohm's law for a resistor or as complex as a full equation set to describe a MOS transistor. To model a system, the symbols, which correspond to the constitutive components, are placed on a sheet and their pins are interconnected by wires. The value of the parameters of each instance—e.g., the resistance of a resistor, the size of a transistor—are adjusted according to the actual component this instance represents. The connection points between the instances are defined as electrical nodes, implicitly described by Kirchhoff's current and voltage laws (KCL, KVL).

As an example of a circuit diagram, the structural description of a Operational Transconductance Amplifier (OTA) is shown in figure 4.1.

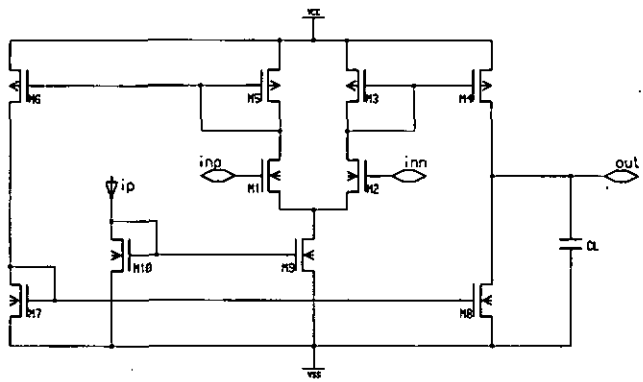


Fig. 4.1 Example of a circuit diagram.

Circuit diagram modelling is basically a structural technique. The structure of the diagram is the same as the structure of the actual

circuit it represents. However, the topological aspect—the placement of the components on a board or on a chip—of the circuit is not described. The symbols are just placed in such a way that it will be easy for an engineer to understand the function of the circuit. The behaviour of the circuit is given by the diagram structure and by the implicit models of all the components. To analyse a whole circuit, we must list the characteristic equations of all the components involved and list the equations that correspond to KCL on each node and KVL on each mesh.

This representation has also been used to describe the behaviour of systems in a macromodelling approach, but the limitations of this technique, as exposed in chapter 2, remain.

As explained above, the nets of a circuit diagram represent both the across and the through quantities and there are neither input nor output ports. For this reason, it is not suitable for the description of signal propagation or signal processing elements and, consequently, circuit diagram modelling cannot be applied to the behavioural description of analogue systems.

Block Diagram Modelling

Block diagrams have been widely used by engineers in many different application domains. Blocks, which have input and output ports are connected by oriented paths. The paths represent signals, which can be of either continuous-time, sampled-data or digital type; a signal can be either an across or a through quantity or even an abstract variable. A block represents a transducer which calculates the value of each output signal according to the respective values of all the input signals. The functionality of a block is indicated either graphically or with an equation. Basically, a path originates at one output of a block and ends at one input of another block. However, if two blocks need to compute the same input signal, a take-off point is introduced in order to bring the signal to both blocks. As summing blocks appear very frequently in this representation, a particular symbol is used. As nearly any block can be defined, block diagrams can be used to describe linear as well as non-linear systems.

As an illustration, figure 4.2 shows the block diagram of a stereo FM modulator [deC84].

As the across and through variables appear separately in the block diagram, it can neither represent the structure of a physical

system—i.e., an electrical circuit—nor describe any physical interface. The behaviour of a system, however, can be represented conveniently.

Moreover, this description technique is very intuitive and can therefore be understood easily by inexperienced users.

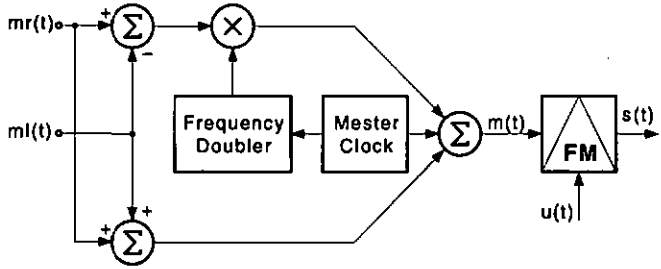


Fig. 4.2 Example of a block diagram.

Signal-Flow Graph Modelling

The Signal-Flow Graph (SFG) representation is very similar to the block diagram. A signal, instead of being represented by a path, is now represented as a node, which can be used directly as a summer node or as a take-off point, whilst a path now represents a transducer. This definition limits the description to single-input single-output transducers. The functionality of a transducer is indicated as an multiplying factor placed near to the corresponding path. Therefore, a SFG is limited to the description of linear systems. As an example, figure 4.3 shows the SFG of a fifth order elliptic continuous-time ladder filter [Ban85].

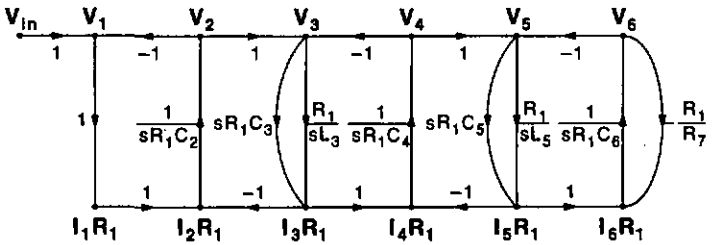


Fig. 4.3 Example of a signsl-flow graph.

This description technique can also represent a behaviour but it is less intuitive than block diagrams. However, as the number of graphical element is reduced (path and nodes), it can be more easily implemented as a front-end to a computer program.

Bond Graph Modelling

Another graphical description method is the bond graph. A bond is represented as an harpoon, which carries both through and across variables. The bonds are connected together using two types of junctions. In a 0-junction, all across variables are equal and all through variables add up to zero: it is a generalization of KCL. In contrast, in a 1-junction, all through variables are equal and all across variables add up to zero: it is a generalization of KVL. Additionally, a conventional character—R for a resistor, C for a capacitor, etc.—placed on each terminating harpoon indicates the type of element it represents. This builds a set of implicit constitutive equations of the system. Figure 4.4 shows the bond graph of a first order RC passive low-pass filter.

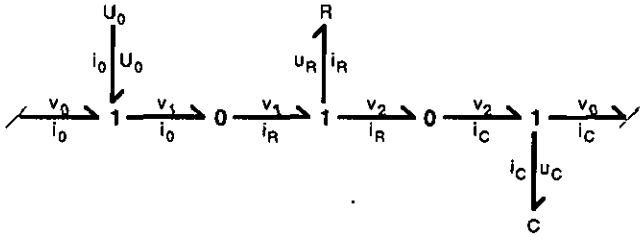


Fig. 4.4 Example of a bond graph.

This method is not very intuitive for the novice user, but it is very powerful for the structural description of physical systems. However, it is not well suited to behavioural descriptions where non-physical variables are used instead of across-through couples. The semantics of the description is similar to the semantics of a circuit diagram but it is generalized to non-electrical physical domains.

4.1.3 Analogue Behavioural Description Method

None of the description techniques exposed in the previous section fully satisfies all the requirements. The main problem is to find a method which supports both a structural description using across and through variables and a behavioural signal processing description using abstract variables. For this reason, different methods are used for different purposes.

Circuit diagrams will be used for the structural description of a system as the interconnection of components. As this classical representation is already available in most design framework, it will be left aside in the rest of this text. The emphasis will now be on the behavioural description of components and systems. However, the physical interface of the components must also be described to allow them to be instantiated in circuit diagrams.

The *interface* will be represented as an *icon*, which gives an external view of the model.

The *behaviour* will be described using *extended block diagrams*. The extensions are the following:

- Blocks with more than one output can be defined.
- The summer type block is extended to the four basic mathematical operators.
- Some specialized blocks are added to describe the interface of the components.

We introduce now a new type of extended block diagram called a *Functional Diagrams* (FD) which is built using standard blocks called *Graphical Building Symbols* (GBS). This new description technique, previously published in [Mos94], will be explained in the following sections.

4.2 The Icon of a Component

The icon is the graphical object that can be used to instantiate the behavioural model in a circuit description. It is then equivalent to any standard component symbol—i.e., resistor symbol, transistor symbol, etc. More formally:

> *The* icon *describes graphically the* interface *of the model with its environment.*

Practically, the icon consists of:

- A *body* which should give a first visual idea of the component's function. For some usual components (e.g., comparators, filters), conventional body representations exist.
- *Pins* used to interconnect the component with other elements in a higher level circuit diagram. Basically, an analogue component is influenced by the quantities on all the surrounding nodes and reciprocally influences them. Consequently, analogue quantities—of both across and through type—can be read or written as contributions to pins. For this reason, the pins have to be defined as bi-directional. Furthermore, they can be electrical or of any other physical nature (i.e., fluid, mechanical, etc.). The pins form the interface of the model with other components placed in the same circuit or system description.
- Optional *properties*, which are the parameters of the model. They allow the user to set up a generic model for a particular application. Each property is defined with a default value. The properties are the interface of the model with the user.
- Optional *textual comments* to better describe the function of the model.

As an example, we go back to our low-pass filter already treated in the previous chapter. A model icon (figure 4.5) has been defined with four pins and ten properties—the coordinates of the poles and zeros of the transfer function, the input capacitance, the output resistance and the value of the static supply current.

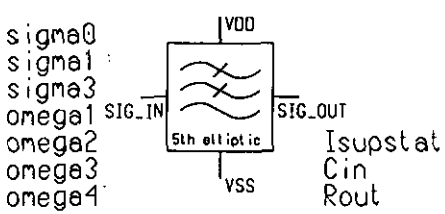


Fig. 4.5 Icon of a fifth order elliptic low-pass continuous-time filter.

4.3 The Functional Diagram

The functional diagram is a schematic representation of an analogue behaviour given in the form of an extended block diagram. Some graphical building symbols, which stand for elements of behaviour are put together and interconnected in order to describe a more complex behaviour. Expressed more formally:

> *The* Functional Diagram (*FD*), *built as an assembly of interconnected* Graphical Building Symbols (*GBSs*), *describes the* behaviour *of the model.*

Note: The structure of the FD does not match the structure of any particular physical (e.g., electronic) realization of the component. Only the behaviour is described.

The functional diagram contains the *semantics* of the model. This semantics is determined firstly by the interconnection scheme of the GBSs in the diagram and secondly by the semantics of each GBS.

The functional diagram is drawn with a graphical editor, that allows GBSs to be chosen from a library, placed in a schematic and connected by wires. As in the block diagram formalism, a wire represents a one-dimensional, non-physical, continuous-time state variable. Connection points on a GBS are oriented and thus indicate the direction of propagation of the information throughout the model. The following connection rule applies: the output pin of a GBS can be connected to an unlimited number of input pins but cannot be connected to another output pin (conflict).

Once the internal behaviour has been described, bi-directional pins, which correspond to the pins on the model's icon, are placed around the diagram. Particular GBSs are added to compose the interface between the physical quantities on the pins and the internal variables of the model.

In order to obtain the desired behaviour, the user can adjust the properties of some GBSs. The value attributed to the properties can be either numerical or it can be an expression in which the external parameters of the model—as defined on the icon—appear.

If we now go back to our continuous-time filter example, we can draw a functional diagram (figure 4.6) to describe its behaviour. We can first recognize an RC-parallel input stage where the resistance is set to a nearly infinite value and the capacitance corresponds to the property *cin* defined on the icon. The value of the voltage on the input

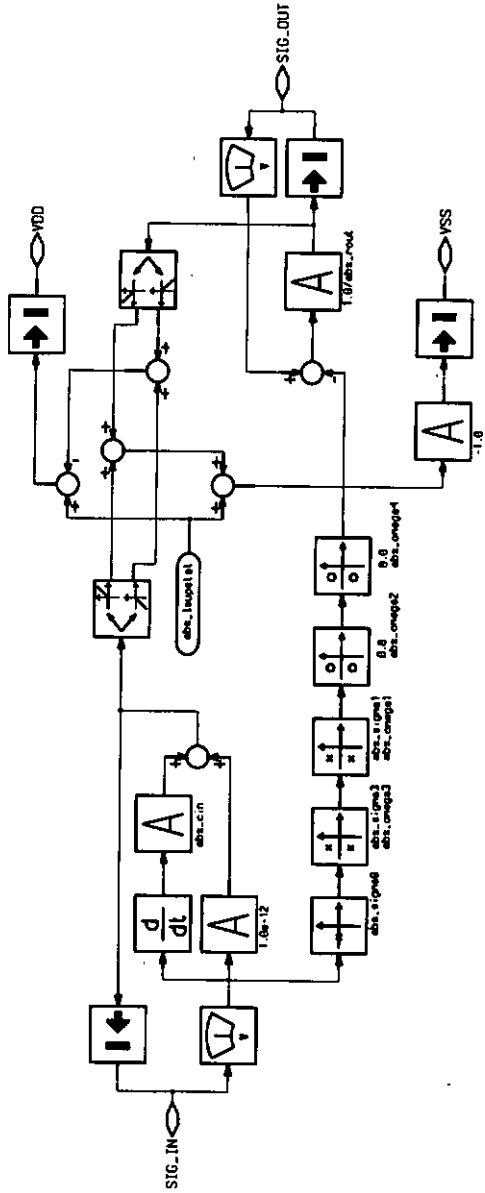


Fig. 4.6 Functional diagram of a fifth order elliptic low-pass continuous-time filter.

pin is read using a *voltage probe* GBS. This way the physical *across* quantity *voltage* is translated into an abstract variable. This value is multiplied by the constant which represents the input conductance using a *gain* GBS. In a parallel path, its derivative is multiplied by the input capacitance *cin*—a parameter of the model. The sum of these two terms is written as a current contribution to the input pin by means of a *current generator* GBS. Now, the abstract variable is translated into a physical *through* quantity of type *current*.

Then, the transfer function is modelled as a series of GBSs, which represent a single pole (*pole1* GBS), two pairs of poles (*pole2* GBS) and two pairs of zeros (*zero2* GBS) respectively. Again, the parameters of the model—icon properties—appear on the various GBSs.

The output of the transfer function could be used directly as the ideal output voltage value. Here, we rather feed it to a resistive output stage in order to model a non-ideal output. The ideal value is compared to the actual value of the voltage on the output pin and divided by the output resistance *rout*. The resulting value is imposed as a current contribution to the output pin.

Finally, a power supply with static supply current *isupstat* has also been defined. The positive and negative current terms are split up using *separator* GBSs and imposed as current contributions to VSS and VDD respectively. The input, output and power supply stages, described graphically here, correspond one-to-one to the HDL versions given in the previous chapter.

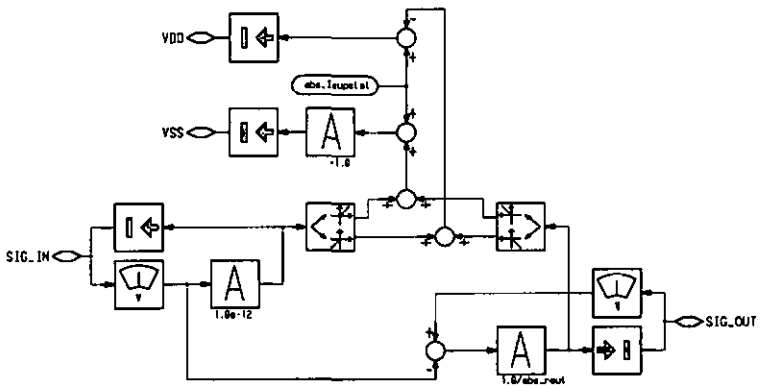


Fig. 4.7 DC-mode FD of a fifth order elliptic low-pass continuous-time filter.

If a component must be modelled differently depending on the analysis type (DC, AC, transient), the behaviour can be described in several analysis-specific functional diagrams. This facility will usually be referred to as multi-FD modelling. Figure 4.7 shows a functional diagram of our low-pass filter valid only in DC mode. All the derivatives are removed from the original FD. The new input stage is now purely resistive and the transfer function is replaced by a wire.

4.4 The Graphical Building Symbols

The graphical building symbols are spare parts which represent elements of behaviour and which are used to build a more complex behavioural description. In other words:

> *A* Graphical Building Symbol (*GBS*) *is the graphical representation of an analogue function.*

A GBS is composed of:

- A graphical *body* drawn so that it gives a good idea of the corresponding behaviour. The body and the name of the GBS together represent the semantics of the GBS in a graphical way. If the body is not meaningful enough, an additional textual description must be provided to the user.
- Optional input and/or output *pins*. In a functional diagram, the information is transformed in the GBS and transits in the form of a variable from one GBS to another through pins. Therefore, they must be oriented. Input pins are bound to the symbol with an incoming arrow so that they can be visually identified. As we will see in sect. 4.5, any pin which is on the physical side of a read/write conversion GBS, however, must be declared bi-directional and marked with a bi-directional arrow. Each pin must also have a distinct name.
- Optional *properties*, which allow the user to adjust a GBS instance to a particular need. For example, the *gain* GBS has a property, also called *gain*, which can be set to an arbitrary value to determine the multiplying factor between the input and the output.

The range of behaviour that can be represented by a GBS varies from elementary mathematical operators to arbitrarily complex sets of

equations. However, a set of basic GBSs is defined, which can be classified in the following categories:

- *Operator GBSs:* they mainly represent current mathematical operators. Beside the four basic operators, time differentiation and integration are defined as well as gain, sign, poles, zeros and some others. Figure 4.8a shows an 2-input adder GBS characterized by

$$OUT = IN_1 + IN_2$$

where IN_1 and IN_2 are the two input pins of the GBS and OUT is the output pin. Figure 4.8b shows a gain GBS characterized by

$$OUT = gain \cdot IN$$

where IN is the input pin of the GBS, OUT is the output pin and *gain* is a property of the GBS.

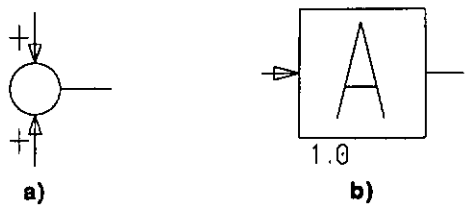


Fig. 4.9 Examples of operator GBSs: a) 2-input adder b) gain.

- *Function generation GBSs:* functions like sine or cosine can be generated to be used in the model. Figure 4.9 shows a sine GBS characterized by

$$OUT = \sin(IN)$$

where IN is the input pin of the GBS and OUT is the output pin.

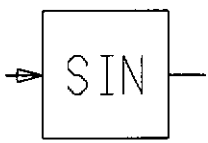


Fig. 4.9 Example of function generator GBSs: sine function.

- *Parameter GBSs:* most of the model parameters (icon's properties) correspond, in the functional diagram, to properties of GBS instances—e.g., the property *gain* of a gain GBS. However some free parameters must also be available directly in the FD. For this reason an additional parameter GBS has been defined. It has one property—the actual parameter—and one output pin which carries the value of the parameter. Additionally, some GBSs access simulation parameters like time or model temperature. Figure 4.10a shows a free parameter GBS characterized by

$$OUT = parameter$$

and figure 4.10b shows a temperature GBS characterized by

$$OUT = temperature$$

Fig. 4.10 Examples of parameter GBSs: a) free parameter b) model temperature.

- *Conversion GBSs:* inside the FD, the wires which connect the GBSs are oriented and they represent non-physical variables only. When the model is used in a circuit diagram, however, its pins are connected to physical nodes. As explained above, these pins are not oriented and they carry a pair of quantities—an across quantity and a through quantity. Some conversion elements are then necessary to interface the two description modes. In the FD, the pins of the model, which are represented by particular connector symbols to indicate the boundary of the FD, can exclusively be connected to a particular class of GBSs, called conversion GBSs. These are used to access the physical quantities on the pins and translate them into variables, a form that can be used inside the FD. Conversion GBSs can be either of *probe* type and represent a read functionality, or of *generator* type and represent a contribution functionality or even both at a time. Figure 4.11a shows a pressure probe GBS. The value of the pressure read on the *fluid* input pin is transmitted to the output

pin of the GBS and can therefore be used as a variable in the FD. Figure 4.11b shows a current generator GBS. The value of the variable present on the input pin of the GBS is imposed as a *contribution* to the current on the *electrical* output pin.

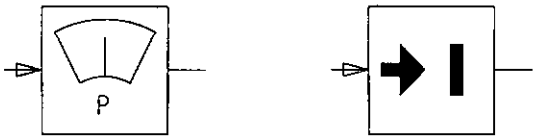


Fig. 4.11 Examples of conversion GBSs: a) pressure probe b) current generator.

As our purpose is to describe analogue—i.e., continuous-time—models, all the GBSs have to be understood as analogue building elements. However, some of them, called "sampled-data GBSs" represent some sampled-data behaviour—like a pole in the z-plane (see § 3.3.3)—described in an analogue way. Additionally, "digital GBSs" could also be defined to represent some digital behaviour—like a flip-flop or a boolean operator—in an analogue way. This is possible because any physical sampled-data or digital system is analogue by nature and can always be described at this level even if the resulting model is optimum. This way, some models could be described using "sample-data GBSs" for the signal processing part, analogue GBS for the interfaces and power supply parts, and "digital GBSs" for an optional command part. In any case—as explained in the previous section—the wires used to interconnect the GBSs always carry continuous-time variables.

A list of the standard GBSs which are included in ABSynth V1.1 [Mos96] can be found in Appendix D.

4.5 Hierarchical Design

In practice, functional diagrams can be quite complex and it is obvious that a diagram with more than a few tens of GBSs becomes difficult to read. For this reason, hierarchy has been introduced.

The idea is as follows: a new *hierarchical GBS* is designed to replace part of the original functional diagram.

Firat the new GBS ia drawn according to the rules given in sect. 4.4. The orientation of the different pina ia defined in auch a way that the new GBS can be directly placed in the original FD.

Then, a *sub- functional diagram* (sub-FD) is drawn, which contains the portion of the original FD replaced by the new GBS. This core is surrounded by oriented connectors, which correapond to the pina of the new GBS. The sub-FD expresses the semantics of the hierarchical GBS.

The hierarchical GBS and the associated sub-FD form a new generic object that can then be placed in a library for later reuse. When a hierarchical GBS is inatantiated in an FD, usera can go down the hierarchy to get a better understanding of the represented behaviour but they are allowed to modify neither the atructure of the sub-FD nor the value of the properties of the conatituent GBSs. However, the hierarchical GBS can be provided with high-level properties which value can be aet by the user at inatantiation time. To enaure the link with the sub-FD, these high-level properties will appear in expressiona attributed to the properties of the lower-level GBSs.

As an example, a hierarchical variant of our filter model is shown in figure 4.12. The input atage, the transfer function, the output stage and the power supply are now represented by hierarchical GBSa. The icon's properties alao appear on the varioua GBSs.

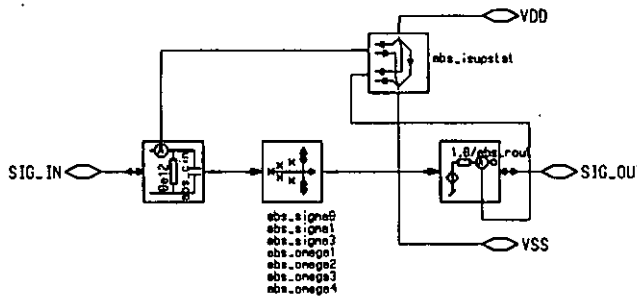


Fig. 4.12 Hierarchical FD of a fifth order elliptic low-pass continuous-time filter.

If we look in more detail at the output stage (figure 4.13), the property R ia defined on the hierarchical GBS and it appears in the expression of the gain property of a gain GBS inside the aub-FD. The sub-FD also includes conversion GBSs (a current generator and a

voltage probe). For this reason, the hierarchical GBS is considered as a conversion GBS with read/write functionality and the corresponding pin is defined bi-directional. The other pins are an input pin that carries the variable V_0—as defined in § 3.1.1—and an output pin which allows the user to access the variable i_{out}.

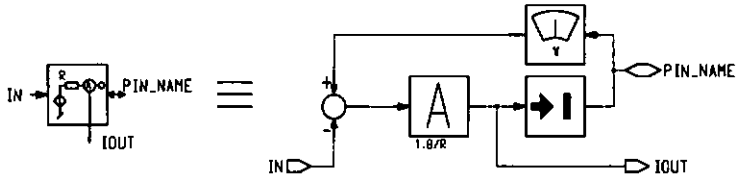


Fig. 4.13 Hierarchical GBS and sub-FD of a resistive output stage.

A sub-FD can contain other hierarchical GBSs without limitations on the number of hierarchy levels. However, recursive description—i.e., the use of a hierarchical GBS in the sub-FD associated with itself—is not allowed.

It is also possible to combine multi-FD description with hierarchical modelling. In this case, the component is described by three different analysis-specific functional diagrams, each of which contains hierarchical GBSs. Furthermore, one can define a hierarchical GBS associated with three analysis-specific sub-FDs.

4.6 Conclusions

After a quick review of some continuous graphical representations, a new graphical method for the modelling of dynamic systems has been presented. The most important concepts introduced are an *icon,* which represents the interface of the system with its environment and a *functional diagram,* which describes the behaviour of the system. The icon can be used at a higher level in a circuit diagram. The functional diagram representation is an extended block diagram representation, many specialized blocks called—*graphical building symbols*—being available.

The communication channels of the icon are limited to the use of properties and physical pins. Signal-flow connection points between components are not included in this description.

The functional diagram has the intuitive and powerful description capabilities of the block diagram and additionally provides ways to link the description of the behaviour to the interface of the component. Furthermore, as the library of GBSs can be easily extended, either using hierarchy or not, a large range of behaviour can be described. However, this graphical modelling method is also based on a divide-and-conquer approach and the limitations discussed in the previous chapter remain valid.

Moreover, due to the infinite number of possible GBSs, a functional diagram will be harder to compute than a signal-flow graph but the well-defined structure of the description ensures that this computation is possible.

4.7 References

[Ban85] M. Banu and Y. Taividis, "An Elliptic Continuous-Time CMOS Filter with On-Chip Automatic Tuning" *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 6, pp. 1114-1121, 1985.

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

[deC84] F. de Coulon, *Théorie et Traitement de Signaux*, (in French), Traité d'Électricité, Vol. VI, Presses polytechniques romandes, Lausanne, 1984.

[Jos95] M. Joshi and H. Kobayashi, "Quantifying Design Productivity: An Effort Distribution Analysis", *EURO-DAC'95 European Design Automation Conference with EURO-VHDL'95*, pp. 476-481, 1995.

[Mos94] V. Moser et al., "A Graphical Approach to Analogue Behavioural Modelling", *The European Design and Test Conference*, pp. 535-539, 1994.

[Mos96] V. Moser and P. Nussbaum, *ABSynth User's guide*, IMT Report 380 PE 01/95, Version 1.1, Université de Neuchâtel, Institut de Microtechnique, 1996.

Chapter 5
Automatic Code Generation

The modelling experience gathered in this project and described in chapter 3 has led to the development of a graphical method for the description of analogue behaviour, as exposed in chapter 4. In a second step towards computer-aided modelling, the equivalent hardware description language model code will be generated starting from this graphical description. The whole process will be considered as successful if the resulting HDL model can be compiled without syntax errors and simulated. Obviously, the simulation results must also match the specifications.

After a general introduction, two strategies for the generation of an HDL architecture will be compared. Then, an entity generator and an architecture generator will be described in detail. These two code generators—already briefly discussed in [Mos95]—constitute the core of the program ABSynth (Analogue Behavioural model Synthesizer) which actual implementation is the subject of chapter 6.

5.1 Introduction

Before going on to the description of the code generation strategies developed, we expose the specifications of the code generation tool. Then, a target Hardware Description Language (HDL) must be chosen and the consequences of this choice with respect to the specifications must be discussed.

Specifications

First we define the general objectives of the code generator:

- The code generator to develop must automatically translate the semantics of an analogue model given graphically as an icon and a functional diagram into an HDL file.
- The generated code must contain the complete description of the model and it must be syntactically right by construction.
- After compilation of the generated code, it must be possible to instantiate the model in a circuit description for simulation.
- The generated models should be coded in a standard HDL.

Target Hardware Description Language

The hardware description language we choose for the long term is VHDL-AMS—the IEEE standard mixed-mode extension of VHDL. As VHDL-AMS has not been completely defined yet, our automatic code generator must be developed for a proprietary modelling language of the VHDL family. Consequently, it must be designed so that it can be easily updated to the standard language. For this reason, the code generation approach must be as independent as possible of a particular language syntax. It is obvious, however, that the definitions of the target HDL syntax must be included in one form or another in the program.

In order to determine the level of language independence we want to achieve, we make the difference between some general language constructs that are common to all the VHDL-like languages and some more specialized language constructs—i.e., keywords, standard functions, etc. The information relative to the general language structure can be part of the code generation program. The more specialized definitions, however, should be stored out of the main program so that they can be updated easily.

From the widely approved VHDL-AMS Design Objective Document (DOD) summarized in [Shi95], we emphasize some properties of VHDL-AMS that help determine the general aspect of the future language. They will be briefly listed here.

- VHDL-AMS must be a "super-set" of VHDL'93 (...) (DO3).

- VHDL-AMS must be capable of describing (...) systems by structural composition of components (DO13).
- Structural description consisting of analogue components of analogue behaviour must observe conservation-law and/or signal-flow semantics (DO14).
- VHDL-AMS must support behaviour specification of an analogue system by a set of linear/non-linear differential/algebraic equations and/or by a sequence of assignments (DO18).

From DO3, we assume that VHDL-AMS models will have the same *entity-architecture* structure as VHDL models. Furthermore, some entities can have several associated architectures. Consequently, we split up the code generation task into two parts:

- An *entity generator,* which translates the information relative to the *icon* into an *entity* declaration. This generator is described in more details below.
- An *architecture generator,* which translates the semantics of a *functional diagram* into an *architecture* description. According to DO13 and DO18, two strategies can be defined to generate the architecture part. It can be either a *behavioural* description made of a collection of statements or a *structural* description based on signal-flow semantics (DO14).

To be considered right by construction, the generated models must strictly follow the target language syntax, but we can limit our approach to a sub-set of the language. Here, we implemented a sub-set of ANACAD's HDL-A which seems compatible with the spirit of the VHDL-AMS DOD. For instance, we only generate purely analogue models since the digital VHDL part is already covered by other tools. Some other limitations will be indicated along this chapter.

5.2 Code Generation Strategies

In a functional diagram, the semantics of the model is given by the structure of the FD and by the semantics of each GBS. In this graphical form, however, the semantics of the GBSs can only be understood by a human user since it is only available as a symbol, a name and an optional textual description. This semantic information must then also be provided in a form that can be understood by a

computer system, usually in the form of a library of HDL code segments. An architecture generation process consists mostly in gathering this information according to the structure of the FD.

In this section, two strategies of architecture generation are described. The first one describes the semantics of the FD using structural constructs of the target language, the second generates a behavioural description.

5.2.1 **Structural Code Generation**

The aim of this first strategy is to translate a functional diagram into a *structural* architecture description, which consists basically of a list of instances and a list of nodes—also called a *netlist*. The instances correspond to the GBSs present in the FD, while the nodes describe the connections.

The semantics of the FD is simply extracted from the graphical database in a netlist form. The semantics associated with the GBSs, however, must be provided as a library of Elementary Behavioural Models (EBMs) [Kla94] coded in the target language. This netlist of interconnected EBMs is basically the same kind of description as a circuit netlist where models of physical components are instantiated. However, the nodes have a different meaning: in a circuit netlist, they represent physical—i.e., across-through—connection points, while in our description they represent signal-flow variables. Alternatively, the EBM instances can also be connected using a degenerated physical interface, the across variable carrying the signal and the through variable being always set to zero.

The main advantage of this method is its simplicity; it is actually just a particular kind of netlisting, a well-known method. However it also has a drawback: the resulting model depends on the EBM library and, therefore, cannot be ported to another system without this library.

In the current version of HDL-A [ANA94a], structural descriptions are not possible and consequently this approach was left aside for future work.

5.2.2 Behavioural Code Generation

In this second approach, a purely *behavioural* architecture will be generated as a collection of behavioural statements.

The semantics of the GBSs is now given in code templates that are written in the target language syntax and stored in a library. The two-step code generation process can be briefly summarized as follows: first, the code segments, which correspond to all the GBS instances present in the FD, are gathered into a single behavioural description; second, the connections of the GBSs are translated into assignment statements.

Compared to the alternative structural strategy, this approach generates models that are independent of any library and therefore more portable. However, the generation process may be more complicated to implement.

An architecture generator based on this strategy will be described further below in this text.

5.3 Entity Generator

We describe now the way an entity clause can be generated. First, our target syntax is specified, followed by a description of the actual entity code generation process. An entity clause describes the interface of a model, which, in our graphical description, corresponds to the icon. The entity must have a name and can have several connection points and parameters. The following correspondences have been defined:

- The name of the entity is the same as the name of the graphical component.
- Each bi-directional pin of the icon is expressed as an analogue `pin` declaration of the same nature.
- Each property of the icon is expressed as a `generic` declaration with its default value.

5.3.1 Reduced Entity Syntax

According to these specifications, we can define a reduced target syntax as a sub-set of the HDL-A entity syntax. The complete HDL-A entity clause includes the declaration of the entity itself and the

declarations of the different input/output elements. The I/O elements can be of four different types, declared in the following order:

- *Generic.* A generic is a parameter of the model. In a circuit description, the user can set the generics of each model instance to particular values independently. The value of a generic remains constant during a simulation. This concept is inherited from VHDL and will therefore remain in VHDL-AMS. The definition of another type of parameter, which value can change during the simulation, however, is foreseen in VHDL-AMS but is not defined in HDL-A and, consequently, was not implemented here.
- *Port.* A port is a input/output object used for the connection of digital signals. As our approach is purely analogue, the port clause has not been implemented.
- *Coupling.* A coupling is a signal-flow-graph input object, which allows a model to use quantities that belong to another model. It has not been implemented either.
- *Pin.* A pin is the analogue connector used to interconnect model instances in a circuit description. It has naturally been implemented.

As we decided not to implement all the I/O types, a reduced entity syntax must be defined to describe the code we actually intend to generate. It is limited to generics of type real and to pins that can be of various *nature* type. This syntax is described below in the Bachus-Naur-Format (BNF). As it is a subset of HDL-A, only our particular definitions are given here. The basic concepts and the specific limitations of our syntax sub-set are briefly commented as they appear in the text. The remaining definitions—as well as additional explanations about the language itself—can be found in the HDL-A language reference manual (LRM) [ANA94b].

```
abs_entity_declaration ::=
ENTITY model_name IS
   abs_entity_header
END ENTITY model_name ;
```

The entity declaration statement declares a new model with a name and an interface described in the entity header. In the last line, the keyword `entity` and the model name, although optional in HDL-A, will always be included in our models because they improve the readability of the final code.

```
abs_entity_header ::=
   [ abs_formal_generic_clause ]
   abs_formal_pin_clause
```

As explained before the `entity_header` does not include port clause or coupling clause. The generic clause is optional but the pin clause is mandatory.

```
abs_formal_generic_clause ::= GENERIC ( abs_generic_list ) ;

abs_generic_list ::= abs_interface_constant_element
   { ; abs_interface_constant_element }

abs_interface_constant_element ::=
   identifier : real
```

The only implemented generic sub-type indication is *real*. In HDL-A, the default value of the generics cannot be set in the entity declaration part. Each generic is declared in a single identifier declaration statement. There is no identifier list.

```
abs_formal_pin_clause ::= PIN ( abs_pin_list ) ;

abs_pin_list ::= abs_interface_pin_element
   { ; abs_interface_pin_element }

abs_interface_pin_element ::=
   identifier : abs_nature_indication
```

Each pin is declared in a single identifier declaration statement. There is no pin identifier list either.

```
abs_nature_indication : abs_single_nature_mark
```

Vector and matrix nature marks are not implemented.

```
abs_single_nature_mark ::=
   electrical | mechanical | mechanical2 | thermal |
   rotational | fluid
```

Theoretically, the approach is not limited to these nature types. More nature types could be added easily.

5.3.2 Entity Generation Process

We now describe the entity generation process that translates an icon description into an entity clause expressed according to our reduced

syntax. First, the information is read from the graphical database. The model name, the property names and the corresponding default values are stored in memory. The default values cannot be used in the HDL-A entity declaration, but this will be possible in VHDL-AMS. The pin names are stored too, together with the corresponding values of the property *nature*. If no nature property is found, the pins are assumed to be *electrical*.

In a second step, the actual entity declaration code is written in a file. The first line of code is built according to the syntax and using the name of the model as stored previously. Then, if generics are present, the generic clause is written. For each generic, a `generic_list` line is written. In fact, there is only one generic identifier per line. A sample `generic_list` code line is copied from a template file, the word `generic` is replaced by the actual generic name and the resulting customized line is copied to the output file. Each line but the last one is followed by a ";".

The generic template line is:

```
generic : real
```

Then, the pin declarations are issued in a similar way. The difference is that both the *pin* name and the *nature* attribute must be set according to the pin description. The actual pin name takes the place of the word `pin`, while the value of the optional property nature replaces the word `electrical` in the following template:

```
pin : electrical
```

Once the generic and pin declarations are complete, the remainder of the entity declaration clause is written.

As an illustration, we go back to the filter example of the previous chapter. The model name is `ellip5`, the icon has ten properties—`sigma0` to `1`, `omega1` to `4`, `cin`, `rout` and `isupstat`—and four electrical pins—SIG_IN, VDD, VSS and SIG_OUT. The entity declaration begins with

```
ENTITY ellip5 IS
```

Then, the properties are extracted from the property tables and expressed as real generic declarations.

```
GENERIC (
        sigma0 : real;
        sigma1 : real;
        omega1 : real;
```

```
        omega2 : real;
        sigma3 : real
        omega3 : real;
        omega4 : real;
        cin : real;
        rout : real;
        isupstat : real
        );
```

The pin declarations are added.

```
PIN (
    SIG_IN : electrical;
    SIG_OUT : electrical;
    VDD : electrical;
    VSS : electrical
    );
```

Finally, the entity declaration is terminated.

```
END ENTITY ellip5;
```

5.4 Behavioural Architecture Generator

In addition to the entity generator, an architecture generator has been developed based on the behavioural code generation strategy described previously. The semantics of the functional diagram is translated into an architecture body using behavioural statements.

First, we give some general information on the architecture generation process. Then, as we only implement a sub-set of HDL-A, the corresponding reduced architecture syntax is defined. Finally, the different parts of the architecture generator are described.

5.4.1 Generalities

In order to link the graphical model description and the code to generate, the following correspondences have been established between the FD and the HDL-A architecture block.

- The name of the architecture is identical to the name of the FD.
- A GBS instance is translated into a piece of HDL code.

- A GBS property becomes a real variable, which must be initialized according to the value of the property.
- A GBS pin becomes an analogue state variable.
- A net becomes a state variable assignment statement.

To be able to interpret the functional diagram, the semantics must be extracted from the graphical database and made available to the code generation program. This information includes:

- The name of the component and the name of the functional diagram which are used in the architecture declaration.
- The list of the properties of the icon, which are used to set the default value of each generic in the initialization block.
- The list of all the nets of the FD.
- A list of the GBS instances, which are sorted according to their respective precedence in order for the final code to reflect the information propagation scheme of the FD. Algebraic loops are kept unchanged for we assume that they will be solved later on at simulation time.

As explained above, the basic idea behind the architecture generator is that each GBS is associated with a code template written in the target syntax. Based on this information, the final code is generated in a two-step sequence:

(1) For each GBS instance, a copy of the corresponding code template is made and the identifiers are modified according to the instance name in order to ensure their uniqueness and the information that corresponds to the interconnections between the GBSs is added to the code as simple assignment statements. This is called the *code customization* process.

(2) All those customized code segments are gathered into a single code file. This is called the *gathering* process.

5.4.2 Reduced Behavioural Architecture Syntax

Again, we only need a reduced syntax to express the semantics of the functional diagram. This architecture syntax is a subset of the architecture syntax of HDL-A, which is compatible with the spirit of

VHDL-AMS. However, as the syntax of an architecture body is much more complex than the syntax of the entity, the current work of the VHDL-AMS language design committee can lead, in the future, to other choices. Nevertheless, we assume that even if the keywords or some language constructs change, the basic ideas will remain correct.

As described in the LRM, the complete architecture of an HDL-A model is composed of the architecture declaration, a block declaration statement and a body statement. The body statement is made of an analogue concurrent statement and a digital concurrent statement, which is not considered here.

The BNF description of the architecture statement syntax we implement is given below with some comments. We only give the definitions which have been modified for our purpose. The remaining definitions can be found in the LRM.

```
abs_architecture_body ::=
   ARCHITECTURE architecture_name OF entity_name IS
      { abs_block_declaration }
   BEGIN
      { abs_body_statement }
   END ARCHITECTURE architecture_name ;
```

The architecture body statement first declares the actual architecture with a particular name and with a reference to the entity to which it belongs. Then a block declaration statement declares all the variables of different types that are used inside the model. The body statement, which includes the actual description of the architecture, comes next. The very last line, which defines the end of the architecture description, is always followed by the architecture name in order to improve the readability of the final code.

```
abs_block_declaration ::=
     VARIABLE name_list : subtype_indication ;
   | STATE name_list : analog_indication ;
```

The block declaration statement has been simplified because our implementation includes neither subprograms nor constants object nor digital signals.

```
abs_body_statement ::=
     abs_concurrent_analog_statement
```

This body statement implementation does not include digital statements. Furthermore, HDL-A itself is limited to a single analogue statement.

```
abs_concurrent_analog_statement ::=
   RELATION
      abs_relational_block { abs_relational_block }
   END RELATION
```

Just as in HDL-A, such a analogue statement can be composed of several relational blocks.

```
abs_relational_block ::=
     abs_explicit_block
   | abs_implicit_block
```

The relational blocks will be of *explicit* type when the relation is given as a procedural description or as an explicit equation—i.e., when the unknowns are functions of independent variables only. The relational block is *implicit* when the relation is an implicit equation—i.e., the unknowns are functions of independent variables and of the unknowns themselves.

```
abs_explicit_block ::=
   PROCEDURAL
      FOR abs_explicit_domain_name =>
         abs_analog_sequence_of_statements

abs_explicit_domain_name ::=
   AC | DC | INIT | TRANSIENT
```

The domain name keywords—i.e., `ac`, `dc` and `transient`—denote for which analysis type a relational block is valid. In HDL-A, a block can be valid for several analysis domain. In our implementation, however, each relational block, though optional, is specific to one and only one domain. Domain lists are not allowed. The `init` keyword specifies a particular explicit block which is used first to define the default values of the generics and second to initialize the other variables. Note that, formally, a generic belongs to the entity. Its default value—as in VHDL—should then be given in the entity part and not in the architecture.

```
abs_analog_sequence_of_statements ::=
   abs_analog_statement { abs_analog_statement }
```

A relational block is composed of at least one statement.

```
abs_analog_statement ::=
   abs_pure_analog_statement ::=
   abs_other_statement
```

The analogue statements are of different types defined in detail in the HDL-A LRM.

```
abs_pure_analog_statement ::=
     state_assignment_statement
   | pin_assignment_statement
   | analog_if_statement
   | analog_loop_statement
```

There are no coupling assignment statements.

```
abs_other_statement ::=
     assert_statement
   | variable_assignment_statement
   | null_statement
```

Procedure call statements have not been implemented either.

```
abs_implicit_block ::=
   EQUATION ( basic_state_name { , basic_state_name } )
      FOR abs_implicit_domain_name =>
         equation { equation }
abs_implicit_domain_name ::=
   AC | DC | TRANSIENT
```

The unknowns of an implicit block must be listed as basic state names after the keyword `equation`. Again, the domain specific implicit blocks must be described separately. As there is no implicit initialization block, the unknowns must be initialized explicitly. In HDL-A, the set of unknowns must be the same for all the analysis domains. Consequently, there must be the same number of equations in each implicit block. Semantically speaking, this limitation has no justification, and we hope that it will disappear in VHDL-AMS.

The remainder of the syntax corresponds to the HDL-A syntax.

5.4.3 Code Customization

As explained above, code customization is the first stage of our behavioural architecture generator. This process transforms a generic code template related to a GBS into a customized code segment

according to the name and to the properties of a particular instance of this GBS. It must fulfil three tasks:

- In the generic code template, the identifiers are generic; in the final code, they must be unique. To achieve this, the name of the GBS instance is used as a prefix to each identifier.
- In the generic code template, the properties of the GBS are initialized to a default value; in the final code, they must be initialized to the value of the corresponding property of the GBS instance. This value is changed accordingly in the initialization part.
- In an FD, each non-interface input of a GBS is connected to the output of another GBS. This information is expressed in the model code as a new statement: the value of the output `state` of the preceding GBS is assigned to the corresponding input `state`.

For example (figure 5.1), the input pin `in` of a gain GBS instance `I2` is connected to the output pin `out` of the GBS instance `I1`. The value of the gain is set to 1000.

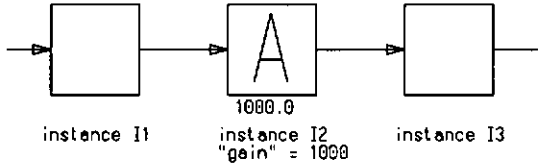


Fig. 5.1 GBS customization example.

The primitive code template of the gain GBS is

```
ARCHITECTURE imt_hdla OF gain IS
    STATE in, out : analog;
    VARIABLE gain : real;
BEGIN
    RELATION
        PROCEDURAL FOR init =>
            in := 0.0;
            out := 0.0;
            gain := 1.0;
        PROCEDURAL FOR dc =>
            out := gain * in;
```

```
        PROCEDURAL POR ac =>
            out := gain * in;
        PROCEDURAL FOR transient =>
            out := gain * in;
    END RELATION;
END ARCHITECTURE imt_hdla;
```

The resulting code is as follows

```
ARCHITECTURE imt_hdla OF gain IS
    STATE I2in, I2out : analog;
    VARIABLE I2gain : real;
```

The identifiers have been modified.

```
BEGIN
    RELATION
        PROCEDURAL FOR init =>
            I2in := 0.0;
            I2out := 0.0;
            I2gain := 1000.0;
```

The value of the gain is set to 1000.

```
        PROCEDURAL FOR dc =>
            I2in := I1out;
```

The value of the variable `I1out` is assigned to `I2in`.

```
            I2out := I2gain * I2in;
        PROCEDURAL FOR ac =>
            I2in := I1out;
            I2out := I2gain * I2in;
        PROCEDURAL FOR transient =>
            I2in := I1out;
            I2out := I2gain * I2in;
    END RELATION;
END ARCHITECTURE imt_hdla;
```

5.4.4 Single FD Architecture Code Gathering

Code gathering is the second stage of the behavioural architecture generator. It aims at gathering all the previously customized code segments into a single model. If the behaviour of the model in the

three analysis domains is described in the same functional diagram, graphical information reading and code customization are done only once. Then the code can be gathered for each block successively. Note that the resulting code can be different in the three analysis domains because the GBS code templates may describe a different behaviour in the different analysis domains.

The gathering task is divided into the following steps.

a. Declaration of the architecture. The architecture name is the name of the functional diagram, the entity name is the name of the component.

```
ARCHITECTURE name_of_the_FD OF name_of_the_component IS
```

b. Declaration block. The declarations of the states and variables are gathered from the customized GBS code segments. Even if a GBS code template has different relational blocks in the different analysis domain, all the variables must be declared in the same block.

```
BEGIN
   RELATION
      -- Declaration of STATEs and VARIABLEs
```

c. Initialization block. The generics are set to their default value. The initialization statements for the states and the variables are gathered from the customized GBS code segments.

```
PROCEDURAL FOR INIT =>
   -- Default values of all the GENERICs
   -- Initialization of the STATEs and VARIABLEs
```

d. DC explicit behaviour description. Gathering of the DC analogue sequence of statements from the customized GBS code segments.

```
PROCEDURAL FOR DC =>
   -- DC explicit behaviour description
```

e. AC explicit behaviour description. Gathering of the AC analogue sequence of statements from the customized GBS code segments.

```
PROCEDURAL FOR AC =>
   -- AC explicit behaviour description
```

f. Transient explicit behaviour description. Gathering of the transient analogue sequence of statements from the customized GBS code segments.

```
PROCEDURAL FOR TRANSIENT =>
   -- Transient explicit behaviour description
```

g. Unknowns of the DC implicit behaviour description. Gathering of the DC unknowns from the customized GBS code segments.

```
EQUATION (
         -- basic state name of all the unknowns
         ) FOR DC =>
```

h. DC implicit behaviour description. Gathering of the DC equations from the customized GBS code segments.

```
      -- DC implicit behaviour description
```

i. Unknowns of the AC implicit behaviour description. Gathering of the AC unknowns from the customized GBS code segments.

```
EQUATION (
         -- basic state name of all the unknowns
         ) FOR AC =>
```

j. AC implicit behaviour description. Gathering of the AC equations from the customized GBS code segments.

```
      -- AC implicit behaviour description
```

k. Unknowns of the transient implicit behaviour description. Gathering of the transient unknowns from the customized GBS code segments.

```
EQUATION (
         -- basic state name of all the unknowns
         ) FOR TRANSIENT =>
```

l. Transient implicit behaviour description. Gathering of the transient equations from the customized GBS code segments.

```
      -- Transient implicit behaviour description
```

m. End of the architecture description.

```
   END RELATION
END ARCHITECTURE name_of_the_FD ;
```

Example

As an example of generated code, we go back to the example of the elliptic low-paas filter treated in the previous chapters. We will not give the whole model code here, but we will rather point out some interesting extracts. The complete generated model code is given in appendix B. In the functional diagram given in figure 4.6, the transfer function is given as a series of GBSs, each of which stands for either poles or zeros. As explained in chapter 3, a single pole and a pair of poles are modelled in HDL-A using the following explicit equations, respectively

```
in - (-1.0/(twopi*p)) * ddt(out) - out == 0.0;
in - b1 * dout - b2 * ddt(dout) - out == 0.0;
```

where `b1` and `b2` are functions of the coordinates of the poles and where `dout` is the first derivative of the output state `out`. A pair of zeros, however, is modelled using an implicit equation. Again, `a1` and `a2` are functions of the coordinates of the zeros.

```
out := in + a1 * din + a2 * ddt(din);
```

The single pole GBS is instance `I706`, the pair of poles are respectively `I707` and `I708`, the pairs of zeros `I709` and `I710`. After code customization—instance names are prefixed to the identifiers and assignment statements are added—and gathering we get the following code, for instance for the AC part of the model:

```
PROCEDURAL FOR ac =>
                                -- GBS I706 to GBS I707 connection
    I707_in := I706_OUT;
                                  -- 1st derivative of I707 output
    I707_dout := ddt(I707_out);
    I708_in := I707_OUT;
    I708_dout := ddt(I708_out);
    I709_in := I708_OUT;
    I709_din := ddt(I709_in);
                                                -- 1st pair of zeros
    I709_out := I709_in + I709_a1 * I709_din
              + I709_a2 * ddt(I709_din);
    I710_in := I709_OUT;
    I710_din := ddt(I710_in);
                                                -- 2nd pair of zeros
    I710_out := I710_in + I710_a1 * I710_din
```

```
            + I710_a2 * ddt(I710_din);
...
EQUATION (
        I706_out,
        I707_out,
        I708_out
        ) FOR ac =>
                                                    -- single pole
    I706_in - (-1.0/(twopi*I706_p)) * ddt(I706_out)
        - I706_out == 0.0;
                                                -- 1st pair of poles
    I707_in - I707_b1 * I707_dout - I707_b2 * ddt(I707_dout)
        - I707_out == 0.0;
                                                -- 2nd pair of poles
    I708_in - I708_b1 * I708_dout - I708_b2 * ddt(I708_dout)
        - I708_out == 0.0;
```

The DC and the transient parts are based on the same FD as the AC description. Furthermore, the code templates corresponding to all the GBSs involved in this example are described the same way in all the three analysis domains. Consequently, the resulting DC and transient code is the same as the extract shown above.

5.4.5 **Multi-FD Architecture Code Gathering**

If the graphical model includes different FDs for different analysis types—i.e., DC, AC and transient—they will be processed successively. In the following detailed description, the points *a* to *m* refer to the code generation steps described in the previous sub-section.

First, the three analysis-domain-specific architectures are generated from the corresponding functional diagrams.

- An intermediate DC architecture description, if specified, is generated according to steps *b*, c, *d*, *g* and *h*. As the description is specific to DC analysis, the states and variable also differ from the states and variable used in the other analysis-specific descriptions. Therefore, the string *DC* is prefixed to all the identifiers to ensure their uniqueness. Additionally, dummy AC and transient equation blocks must also be generated. Indeed, the syntax requires that the same unknowns must be declared in all the three equation blocks. To satisfy this requirement, each unknown that appears in

the DC part must also be declared in the AC and transient parts. Moreover, as the number of unknowns must match the number of equations, dummy equations—i.e., `yy == 0.0;`—must be added.

- An intermediate AC architecture description, if specified, is generated according to steps *b*, c, *e*, *i* and *j*. The identifiers are prefixed with *AC*. Dummy DC and transient equations are added if necessary.
- An intermediate transient architecture description, if specified, is generated according to steps *b*, *c*, *f*, *k* and *l*. The identifiers are prefixed with *TR*. Dummy DC and AC equations are added if necessary.

Then, this information is gathered to form the final model code according to the following steps:

a'. The name of the architecture is formed as a concatenation of the names of all the FDs.

b'. The declarations are gathered from the analysis domain specific architectures.

c'. The initialization statements are gathered. Note that the assignment of the default values of the GBS is repeated three times. This is syntactically incorrect but the model can still be compiled. As this is considered temporary syntax, this side-effect has not been fixed.

d', *e'* and *f'*.
The explicit DC, AC and transient architecture descriptions are copied successively.

g'. The unknowns for the DC implicit relational block are gathered. The dummy declarations are copied too.

h'. The DC equations are gathered, including the dummy ones.

i'. The unknowns for the AC implicit relational block are gathered. The dummy declarations are copied too.

j'. The AC equations are gathered, including the dummy ones.

k'. The unknowns for the transient implicit relational block are gathered. The dummy declarations are copied too.

l'. The transient equations are gathered, including the dummy ones.

m'. End of the architecture description.

Example

Our low-pass elliptic filter has now been modelled in three analysis-specific functional diagrams. The AC and transient modes are modelled with the same FD as in the previous example (figure 4.6) but the DC model is now much simpler—the poles and zeros are just replaced by a wire. We show below the resulting code extracts that correspond to poles and zeros.

The DC procedural block does not contain poles or zero equations and will therefore not be shown here. The AC and transient procedural blocks, as they have been generated starting from the same FD, are very similar to the single-FD case. We only show the AC part since the transient part only differs by the identifier prefix—the prefix AC is replaced by TR. In a general case, however, the AC and transient blocks could be based on different FDs and consequently the corresponding code parts would be completely different too.

```
PROCEDURAL FOR ac =>
    ACI707_in := ACI706_OUT;
    ACI707_dout := ddt(ACI707_out);
    ACI708_in := ACI707_OUT;
    ACI708_dout := ddt(ACI708_out);
    ACI709_in := ACI708_OUT;
    ACI709_din := ddt(ACI709_in);
    ACI709_out := ACI709_in + ACI709_a1 * ACI709_din
                + ACI709_a2 * ddt(ACI709_din);
    ACI710_in := ACI709_OUT;
    ACI710_din := ddt(ACI710_in);
    ACI710_out := ACI710_in + ACI710_a1 * ACI710_din
                + ACI710_a2 * ddt(ACI710_din);
...
```

No pole and zero GBSs are present in the DC functional diagram but the DC equation block still contains the declarations of the unknowns used in the other analysis domains and the corresponding dummy equations.

```
EQUATION (
        ACI706_out,
        ACI707_out,
```

```
        ACI708_out,
        TRI706_out,
        TRI707_out,
        TRI708_out
        ) FOR dc =>
    ACI706_out == 0.0;
    ACI707_out == 0.0;
    ACI708_out == 0.0;
    TRI706_out == 0.0;
    TRI707_out == 0.0;
    TRI708_out == 0.0;
```

The AC equation block contains, again, the declarations of all the unknowns used. The equations are the actual AC equations plus dummy ones that correspond to the transient-mode unknowns.

```
EQUATION (
        ACI706_out,
        ACI707_out,
        ACI708_out,
        TRI706_out,
        TRI707_out,
        TRI708_out
        ) FOR ac =>
    ACI706_in - (-1.0/(twopi*ACI706_p)) * ddt(ACI706_out)
        - ACI706_out == 0.0;
    ACI707_in - ACI707_b1 * ACI707_dout
        - ACI707_b2 * ddt(ACI707_dout) - ACI707_out == 0.0;
    ACI708_in - ACI708_b1 * ACI708_dout
        - ACI708_b2 * ddt(ACI708_dout) - ACI708_out == 0.0;
    TRI706_out == 0.0;
    TRI707_out == 0.0;
    TRI708_out == 0.0;
```

Once more, we do not show the transient-mode equation block since it is very similar to the AC-mode one. All the unknowns must also be declared. The equations are then the actual transient equations plus dummy ones for the AC-mode unknowns.

5.4.6 Hierarchical Code Generation

Instead of being stored as a code template, the semantics of a hierarchical GBS is expressed as a sub-FD. We show here how this semantics is translated into a new code template so that it can be used by the top-level architecture generator.

This process is applied recursively to all the hierarchical GBSs until all the code templates are available. A newly generated code template can be stored in a library for later reuse so that the corresponding GBS can be considered as a flat GBS the next time it appears in an FD. The associated code template can then be used directly and much CPU time is saved.

If the hierarchical GBS is a *conversion* GBS, an entity—with the same name as the GBS—must be generated. It only includes a declaration of the bi-directional pins that are on the physical side of the interface.

Then, in all cases, the architecture part is generated according to the following steps:

a". Declaration of the architecture. The name of the architecture is always `imt_hdla`. The name of the entity to which it belongs is the name of the hierarchical GBS.

b". Declaration block. The properties of the hierarchical GBS are declared as `variables`, the oriented connection points are declared as `states`. In addition, the declarations of all the customized GBS code segments which correspond to the GBS instances of the sub-FD, are added to the code.

c". Initialization. The `variables`, which correspond to the properties of the hierarchical GBS, are initialized according to their respective default values. The `states`, which correspond to the oriented connection points, are initialized to zero. Then, the initialization part is completed with the information from the GBS instances of the sub-FD.

d" to *l"*.
The architecture code is built from the sub-FD the same way as for a single FD—points *d* to *l*.

m".
End of the `imt_hdla` architecture description.

As already pointed out in the previous chapter, it is possible to combine multi-FD modelling with hierarchical modelling. If the model is described by analysis-specific functional diagrams that contain hierarchical GBSs, the hierarchy is solved first—i.e., new code templates are generated according to points *a"* to *m"* above—and then the final code is generated using the newly generated code templates according to points *a'* to *m'* as explained in § 5.4.4.

On the other hand, if a hierarchical GBS is described by three analysis-specific sub-FDs, the corresponding code template must be generated according to points *a"'* to *m"'* given below. Then the new code template can be used as usual.

a"' to c"'.

Declarations and initializations as in *a"* to c".

d"' to *l"'*.

The architecture code is built from the different analysis-specific sub-FDs the same way as for a multiple FD—points *d'* to *l'*.

m"'.

End of the `imt_hdla` architecture description.

Example ∘

We saw in the previous chapter (figure 4.7) that the transfer function of our elliptic low-pass filter can be described as a hierarchical GBS. Here we first give an extract of the code template generated for the transfer function. It is very similar to the code generated in § 5.4.4 in the case of a single flat FD. The indexes `I1`, `I2` and `I3` correspond to the names of the GBS instances in the sub-FD.

```
PROCEDURAL FOR ac =>
    I2_in := I1_OUT;
    I2_dout := ddt(I2_out);
    I3_in := I2_OUT;
    I3_dout := ddt(I3_out);
    I4_in := I3_OUT;
    I4_din := ddt(I4_in);
    I4_out := I4_in + I4_a1 * I4_din + I4_a2 * ddt(I4_din);
    I5_in := I4_OUT;
    I5_din := ddt(I5_in);
    OUT := I5_in + I5_a1 * I5_din + I5_a2 * ddt(I5_din);
```

```
...
EQUATION (
        I1_out,
        I2_out,
        I3_out
        ) FOR ac =>
    IN - (-1.0/(twopi*I1_p)) * ddt(I1_out) - I1_out == 0.0;
    I2_in - I2_b1 * I2_dout - I2_b2 * ddt(I2_dout) - I2_out
        == 0.0;
    I3_in - I3_b1 * I3_dout - I3_b2 * ddt(I3_dout) - I3_out
        == 0.0;
```

Then, this new code template is customized and placed as part of the final model code. Now, the identifiers have a double prefix: the name of the fifth-order elliptic transfer function GBS instance (I15) has been prefixed to the identifiers of the code template. First, we see the assignment statement that stands for the connection of the transfer function GBS to another one. Then, some extracts of procedural code are given, followed by the AC equation block, which in this particular case entirely comes from the transfer function code template.

```
PROCEDURAL FOR ac =>
                                    -- GBS I15 to GBS I17 connection
    I15_IN := I17_OUT;
...
                        -- double-prefix identifier procedural code
    I15_I5_in := I15_I4_out;
    I15_I5_din := ddt(I15_I5_in);
    I15_OUT := I15_I5_in + I15_I5_a1 * I15_I5_din
             + I15_I5_a2 * ddt(I15_I5_din);
    I2_in := I17_II;
...
                                     -- GBS I6 to GBS I15 connection
    I16_IN := I15_OUT;
...
                          -- double-prefix identifier equation code
EQUATION (
        I15_I1_out,
        I15_I2_out,
        I15_I3_out
        ) FOR ac =>
```

```
I15_IN - (-1.0/(twopi*I15_I1_p)) * ddt(I15_I1_out)
    - I15_I1_out == 0.0;
I15_I2_in - I15_I2_b1 * I15_I2_dout
    - I15_I2_b2 * ddt(I15_I2_dout) - I15_I2_out == 0.0;
I15_I3_in - I15_I3_b1 * I15_I3_dout
    - I15_I3_b2 * ddt(I15_I3_dout) - I15_I3_out == 0.0;
```

5.5 Conclusions

In this chapter, we showed how the graphical description of an analogue behavioural model—expressed as a functional diagram and an icon—can be translated into hardware description language (HDL) code. We first discussed the choice of a target language—HDL-A was chosen until VHDL-AMS is available—and the influence of this choice on the code generation process. This led to the definition of two separate generators—one for the *entity* and one for the *architecture*. Then, two strategies for the architecture generation were compared and, due to the limitations of HDL-A, only the behavioural approach has been retained.

Next, the entity and architecture generation processes were described in more detail, including a definition of a reduced target language syntax. The architecture generation was separated into two main tasks: the *code customization* task and the *gathering* task. The second one was described in different versions depending on the type of functional diagram to handle—i.e., flat FD vs. hierarchical FD, single FD vs. analysis-specific FDs. Note that, as code gathering is done according to the respective precedence of the GBS instances, algebraic loops drawn in the FD are kept unchanged in the generated code. We assume that they can be solved at simulation time.

We only showed here the code generation methods. The actual implementation in the form of the computer program ABSynth is the subject of the next chapter.

In order to remain flexible with respect to hardware description languages, one of our objectives was to develop code generation methods that are independent of the target language syntax. In practice, however, this was possible only to a limited extent. We discuss below three situations where the language syntax still influences the structure of the solutions proposed and we try to evaluate the cost and risk of a corresponding modification, particularly in the perspective of a transition from HDL-A to VHDL-AMS.

- The entity-architecture structure of HDL-A led to the design of two separate generators. A change of this definition in VHDL-AMS would require deep modifications but, as VHDL is also based on this definition, the risk is very low.
- The architecture of an HDL-A model consists of several distinct parts: procedural and equation blocks as well as lists of unknowns. This directly influences the structure of the architecture generators. The risk is considered high that the structure of an architecture clause is defined differently in VHDL-AMS and this will require quite deep modifications.
- The details of the target language syntax are stored in separate code templates which must obviously be completely rewritten in case of a change in the target language. However, it has only little influence on the generators. The code template parser is the only program part that should be modified. Here, the risk of a modification can be considered as very high.

Consequently, we claim that our code generation approach is quite robust, but that the transition to VHDL-AMS will still require some modifications in the structure of the architecture generators.

5.6 References

[ANA94a] *HDL-A User's Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

[ANA94b] *HDL-A Language Reference Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

[IEE95] IEEE PAR 1076.1 VHDL Analog Extensions, *VHDL-A Design Objective Document,* Version 2.3, 1995.

[Kla94] M. Klaarwater, *Netlisting Analog Behavioral Models from Graphical Capture,* IMT Report 360 PE 03/94, Université de Neuchâtel, Institut de microtechnique, 1994.

[Mos95] V. Moser et al., "Generating VHDL-A-like Models Using ABSynth", *EURO-DAC'95 European Design Automation Conference with EURO-VHDL'95,* pp. 522-527, 1995.

[Shi95] C.-J. R. Shi and A. Vachoux, "VHDL-A Design Objectives and Rationale", *Modelling in Analog Design,* pp. 1-30, J.-M. Bergé, O. Levia and J. Rouillard (eds.), Kluwer Academic Publisher, Boston, 1995.

Chapter 6
Implementation

As a first step towards implementing the graphical modelling method exposed in chapter 4 and the code generators of chapter 5 as a computer program, a FAS [ANA92] generator has been developed in 1993 and was demonstrated to the public at the European Design and Test Conference in March 1994 in Paris. This first version included basic features only and a restricted GBS library. Its aim was to establish the feasibility of such a model generation tool and to validate the associated graphical description method presented at the same conference [Mos94]. The positive feedback this tool received from potential users and from CAD tool vendors was encouraging and we decided to continue on this track. At that time the HDL-A language reference manual [ANA94b] became available and we developed a second version of the model generator with the same functionality but with HDL-A as a target language. This second tool was demonstrated at the FSRM meeting in Bern in November 1994 under the name ABSynth (Analogue Behavioural model Synthesizer). Since then, ABSynth has been further developed. New features, like multi-FD specification and hierarchical code generation, have been added. The GBS library has been completed with more analogue GBSs and with sampled-data GBSs. The tool has also been extended to the description of non-electrical systems (mixed-nature modelling). This implementation was described in a publication presented at EURO-VHDL'95 in September 1995 in Brighton [Mos95].

In this chapter we will first describe the tool ABSynth, then, we will see how to include ABSynth into a complete analogue design flow.

6.1 The Tool ABSynth

The computer-aided modelling tool ABSynth (Analogue Behavioural model Synthesizer) is a software implementation of the graphical behaviour description of chapter 4 and of the code generators of chapter 5. We first give the global structure of the tool, then we explain the most important modules. Finally, two versions of the user interface are described.

6.1.1 Tool Structure

Here, we explain the global structure of ABSynth. The different parts are listed according to a logical review of the functions of the whole system, as illustrated in figure 6.1.

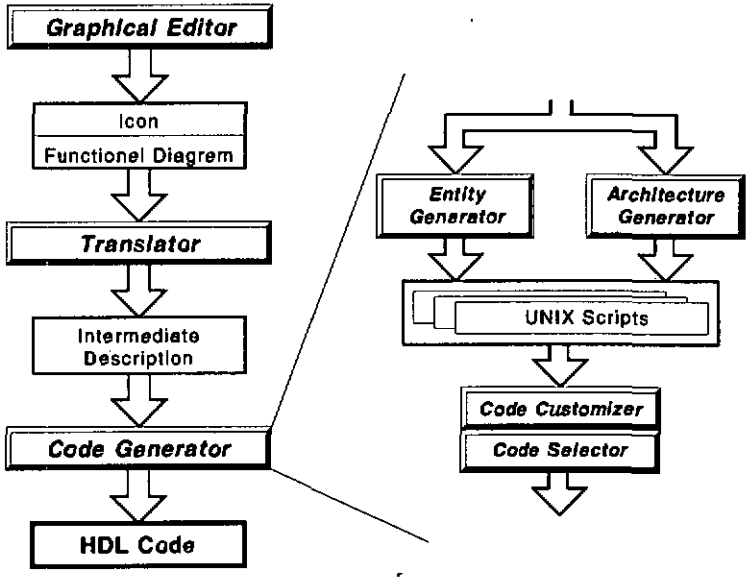


Fig. 6.1 ABSynth tool structure.

First, a *graphical capture tool* is necessary to draw the functional diagram and the icon of the component to model. Then the graphical information, which corresponds to the whole component model is

translated into an *intermediate description form*. Next, this information is used by the actual *code generator*, which calls the *entity generator* and the *architecture generator* with the respective options in order to build the final model code.

The entity and architecture code generators produce intermediate UNIX scripts that call the *code customizer* and the *code selector* used to perform the gathering task. These last two programs access the libraries of primitive code templates.

6.1.2 Intermediate Description Format

An intermediate description format is used to store the semantics of the functional diagram and the icon in a non-graphical from. This way, the graphical part of the tool and the code generation part are well separated. Therefore, a change in the graphical tool database does not influence the code generation process. Reciprocally, a change in the target language syntax does not influence the graphical part. However, this tool structure necessitates more computation steps to go through the whole process.

Specifications

The specification of the intermediate description is a key issue in the design of our tool because this description ensures the communication between the graphical part and the code generation part. Therefore, it has to be defined first. Once the syntax of this language is specified, two programs must be provided: a code generator used to translate the graphical information into the intermediate format and a parser to read the descriptions given in the intermediate format and load the corresponding information into memory so that further program units can work on it. This is illustrated in figure 6.2.

According to the correspondence between the graphical objects and the target HDL code listed in chapter 5, the intermediate description must contain the following information:

- The name of the component.
- A list of pins and a list of the properties with their default values.
- The name of the functional diagram.

- A list of the GBS instances present in the FD with the respective value of all their properties.
- A description of the GBS instances interconnections.
- The hierarchical description of each GBS instantiated in the FD including the interface (pins, properties) and, possibly, a sub-FD.

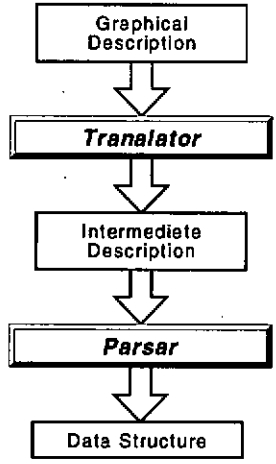


Fig. 6.2 Intermediate description generating and reading.

The EDIF Language

Instead of defining yet another netlist description format, we decided to use the EIA/ANSI standard language EDIF (Electronic Design Interchange Format). EDIF is a very powerful language that offers various description views from netlist level down to layout level. We will not present the whole EDIF language here but just point out the particular facilities we need. The version we implemented is EDIF 2 0 0 [Kah92]. This language offers many interesting features:

- The language syntax and semantics have been validated theoretically [Lau93] and implemented in many practical applications.
- As EDIF is a widely spread standard, it provides a loose link between the graphical part and the code generation part. This

way, the code generators can remain independent from the particular graphical entry tool used.

- Another advantage of choosing an existing format like EDIF is that parsers and other utilities are available on the market.

Sub-set of EDIF

We describe here how a component modelled with ABSynth can be described in EDIF. Only a sub-set of the whole language is used. The definitions provided here have been derived from the official EDIF definitions [EIA87]. In the syntax description, the keywords printed in italics refer to objects of our graphical description.

The modelled component is described as a `design` which refers to a `cell` called the top cell of the description and to a `library` (the top library), which corresponds to the path of the component in the database of the graphical tool.

```
absynthdesign ::=
     '(''design' ComponentName
          '(''cellRef' ComponentName
               '(''libraryRef' ComponentPath ')' ')' ')'
```

The top library is defined in a very restrictive way: the EDIF level is `0` and the technology statement is minimal.

```
toplibrary ::=
     '(''library' ComponentPath
          '(''edifLevel' '0' ')'
          '(''technology'
               '(''numberDefinition'
                    '(''scale' '1' '(''e' '1' '-6' ')'
                         '(''unit' 'distance'')' ')' ')' ')'
          topcell
     ')'
```

The name of the top cell is, again, the name of the component, the cell type is `generic`.

```
topcell ::=
     '(''cell' ComponentName '(''cellType' 'generic'')'
          topview
     ')'
```

The top cell's view carries the name of the functional diagram, its type is `netlist` and it includes an `interface` declaration and a `contents` declaration.

```
topview ::=
     '(''view' FunctionalDiagramName '(''viewType' 'netlist'')'
          topcellinterface
          topcellcontents
     ')'
```

The interface of the top cell must have at least one `port` and may have `properties`.

```
topcellinterface ::=
     '(''interface'
          topcellport
          { topcellport | topcellproperty }
     ')'
```

The directionless connection points of the icon are declared as ports with direction `INOUT`. A property `nature` can be added to indicate a non-electrical pin.

```
topcellport ::=
     '(''port' PortName (''direction' 'INOUT'')'
        [ '(''property' 'nature' '(''string' "NatureType"')''')' ]
          '(''property' 'pin' '(''string' "PortName" ')' ')'
          '(''property' 'pintype' '(''string' "'io'" ')' ')' ')'
```

The icon's properties are declared with their respective default values, given as `strings`.

```
topcellproperty ::=
     '(''property' PropertyName
          '(''string' "DefaultPropertyValue" ')' ')'
```

The contents of the top cell describes the functional diagram as a list of GBS `instances` and a list of `nets`.

```
topcellcontents ::=
     '(''contents' { gbsinstance | fdnet } ')'
```

The GBS `instance` listing includes a reference to a GBS cell and to the corresponding library. The value set to a GBS instance property in the FD appears as a string in this GBS instantiation statement.

```
gbsinstance ::=
     '(''instance'
          GBSInstanceName
          ( leafgbsviewRef | hierarchicalgbsviewRef )
          { 'property' GBSPropertyName
               '(''string' "GBSInstancePropertyValue"')' } ')'

leafgbsviewRef ::=
     ( '(''viewRef' LeafGBSName 'cellRef' LeafGBSName
          'libraryRef' LeafGBSLibraryName ')'

hierarchicalgbsviewRef ::=
     '(''viewRef' SubFDName 'cellRef' HierarchicalGBSName
          'libraryRef' HierarchicalGBSLibraryName ')' )
```

The nodes of the functional diagram are listed here as `joined` statements each of which lists a number of port references. A `port` can be either a port of the icon—in this case, there is no further reference statement—or a port of a GBS referred to by an instance reference statement.

```
fdnet ::=
     '(''net' NetName
          '(''joined' {
               '(''portRef' PortName ')'
               | '(''portRef' GBSPortName
                    '(''instanceRef' GBSInstanceName ')' ')'
               } ')' ')'
```

The description of the top cell is not sufficient. Each GBS instantiated must also be declared. In the final EDIF file, this declaration must take place before the GBS can be instantiated. Hierarchical GBS descriptions are composed of `contents` and an `interface` and are very similar to the description of the top cell. Leaf GBS descriptions, on the other hand, only contain an `interface`. The GBS declarations are also grouped in libraries, which precisely correspond to the GBS libraries defined in the graphical database.

```
gbslibrary ::=
     '(''library' GBSLibraryPath
          '(''edifLevel' '0' ')'
          '(''technology'
               '(''numberDefinition'
                    '(''scale' '1' '(''e' '1' '-6' ')'
```

```
                        '(''unit' 'distance'')' ')' ')' ')'
          { gbscell }
     ')'

gbscell ::=
     '(''cell' GBSName '(''cellType' 'generic'')'
          ( leafgbsview | hierarchicalgbsview )=
     ')'

leafgbsview ::=
     '(''view' GBSName '(''viewType' 'netlist'')'
          gbsinterface
     ')'

hierarchicalgbsview ::=
     '(''view' SubFDName '(''viewType' 'netlist'')'
          gbsinterface
          gbscontents
     ')'

gbsinterface ::=
     '(''interface'
          gbsport
          { gbsport | gbsproperty }
     ')'
```

The connection points of a GBS are described as input, output or bi-directional ports.

```
gbsport ::=
     '(''port' GBSPortName
          '(''direction' ('INPUT' | 'OUTPUT' | 'INOUT') ')'
          '(''property' 'pin' '(''string' "GBSPortName" ')' ')'
          '(''property' 'pintype'
                '(''string' "('IN' | 'OUT' | 'IXO')" ')' ')' ')'
```

The property declaration includes a string field for the default value specified on the GBS.

```
gbsproperty ::=
     '(''property' PropertyName
          '(''string' "DefaultPropertyValue" ')' ')'
```

The contents of a hierarchical GBS are also described as a list of instances and a list of nets.

```
gbscontents ::=
     '('contents' { gbsinstance | fdnet } ')'
```

6.1.3 Graphical Capture Tool

Once the intermediate description format has been defined, a front-end editor must be specified too.

Specifications

In the context of ABSynth, the graphical editor must be able to provide all the features of our graphical modelling method, as described in chapter 4. The following specifications stem from it:

- Icon editor functionality: it must be possible to draw a new icon, place bi-directional (or directionless) pins on its border and attach properties to the icon as well as to the pins.
- Schematic editor functionality: it must be possible to select symbols from a library, place them and connect them in order to form a functional diagram. Single wire connections are sufficient.
- Graphical building symbol editor functionality: it must be possible to draw new graphical building symbols, place oriented pins on their border and attach properties to them. It must be possible for the user to define new symbol libraries and to extend existing ones with new symbols.
- Hierarchical design must be possible.
- It must also be possible to link several schematics to a single icon.
- An EDIF output must be provided.

Mentor Graphics Design Architect

As many very good schematic entry tools have existed for years, we decided not to develop our own editor but to use an existing product. The tool Design Architect by Mentor Graphics, which offers all the required features [Men94a] has been chosen. However, since the interface between the graphical editor and the code generator is based

on a standard format (EDIF), it would be easy to implement ABSynth using another graphical editor: only the GBSs must be re-drawn.

The EDIF generator which goes with Design Architect is *enwrite*, also by Mentor Graphics [Men95a]. In our application, enwrite is called in batch mode with the default settings. Additionally, Design Viewpoint Editor—another tool by Mentor Graphics—[Men94b] is used to select the sheet which corresponds to the specified functional diagram.

A new menu *ABSynth* and some dialogue boxes have been designed to extend the functionality of Design Architect for our application.

6.1.4 Executable Program Modules

The various executable program modules we developed will be shortly described here. The respective command line syntax and the corresponding options are given in appendix A. The modules interact as shown previously in figure 6.1.

A code generation script calls the entity generator and the architecture generator. Both of them read the EDIF description of the design using a parser and generate shell script programs. These scripts, which contain calls to the code customizer and to the code selector are then executed to obtain the final HDL-A code.

The EDIF parser is a public domain program, the four remaining modules have been implemented in C and they communicate with each other using the UNIX *pipe* mechanism.

Entity Generation Program

The HDL-A entity generator translates the EDIF top cell *interface* description into an HDL-A *entity clause*. An EDIF port declaration becomes an HDL-A pin clause while an EDIF property declaration becomes a formal generic clause in HDL-A.

Architecture Generation Program

The HDL-A architecture generator translates the EDIF top cell *contents* description into a behavioural HDL-A *architecture clause*. In

case of a hierarchical GBS, the program is executed recursively to generate the corresponding primitive code template based on the EDIF *view* description of the GBS.

In case of multi-FD specification, the architecture code generator is called for each analysis-specific FD to generate the corresponding intermediate architecture code. Then, the final code is built using the code selection program described below.

In the current version (version 1.1) of ABSynth, multi-sub-FD modelling—i.e., the description of the behaviour of a hierarchical GBS by means of analysis-specific sub-diagrams—has not been implemented.

Code Customization Program

The code customizer is able to parse a primitive code template and to modify its contents. The identifiers and the values of the parameters are changed according to the command line.

Code Selection Program

The code selector parses a customized code template and selects a particular part of the code. It is used to gather the code relative to all the GBS instances into one single file. In multi-FD mode, it is also used to gather the three analysis-specific architecture descriptions into one single model.

6.1.5 User Interface

The main task of the ABSynth users is to draw the functional diagrams and the icons of the components they want to model. This is done by means of a graphical editor. A set of standard graphical building symbols—including hierarchical ones—is available in a library but the user is free to define new libraries and new GBSs. The icons and the functional diagrams are also stored in the database of the graphical editor so that parts of FDs can be easily reused.

Once the graphical model is complete, the user can launch the code generator. This can be done either in a shell or directly from within the graphical tool using a particular menu. The code generator

needs primitive code templates, which describe the behaviour of each GBS. They are stored in a three-level library structure.

- The *standard library* contains the descriptions of all the standard GBSs in the form of code templates.
- An optional *system library* is used to store the code templates corresponding to custom GBSs common to several users.
- An optional *user library* is used to store the description of a set of user-specific GBSs.

For each GBS present in a functional diagram, ABSynth searches for a behavioural description, first in the user library, then in the system library, and finally in the standard library. If no code template exists, ABSynth looks for a sub-FD and generates a new code template accordingly. If required, this new code template can then be saved in the user library.

Once the model code has been synthesized, it can be compiled and used in a simulation.

Code Synthesis Script

The complete code generation process can be executed as a single command in a UNIX shell. The full component path must be provided as a parameter. Various options have been implemented.

If several functional diagrams exist for the same component, users can specify which one must be considered or they can assign a different functional diagram to each analysis domain. They can also decide to generate only the entity.

It is also possible to specify the user library path, the system library path and the output file name.

A model can be considered either as flat or as hierarchical. In the flat mode, a primitive code template must be available before synthesis for each GBS. In the hierarchical mode, the code template of a hierarchical GBS can be generated based on the sub-FD and, optionally, saved in the user library. The complete usage syntax—with the description of the options—is presented in appendix A.

Mentor Menu

A new menu has also been defined in the graphical environment of Mentor Graphics. It offers the same functionality as the shell command, but allows the user to set the options and generate the code in a more convenient way without leaving the design environment. Figure 6.3 shows the aspect of the ABSynth dedicated menu and the synthesizer options dialogue box. The *Synthesize* button is used to generate the model code, the *Compile* button is just a shortcut used to call the built-in HDL-A compiler. Both actions can also be launched in one single run, if preferred. The user can set up the synthesis and compilation environment using the *Setup* button or the various *Options...* sub-menus. For instance, the synthesis dialogue box allows the user to specify the component path, the name of the FD(s) to consider, the user code template library path and a number of other options.

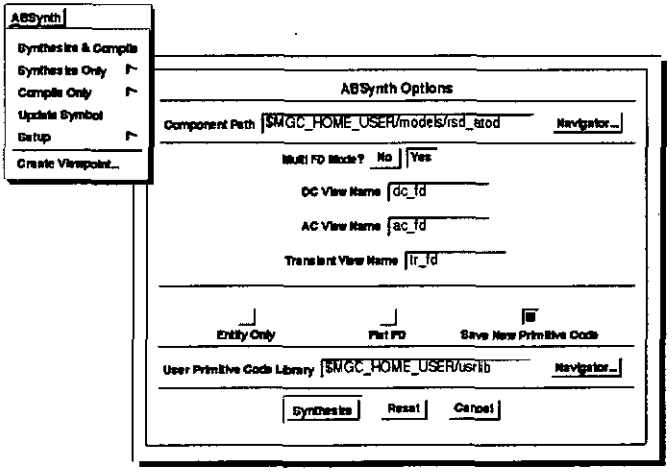


Fig. 6.3 ABSynth menu and associated synthesis dialogue box.

6.2 Analogue Design Flow with ABSynth

ABSynth, with HDL-A as a target language, can be integrated either in the design flow proposed by ANACAD or in the Mentor Graphics environment.

In ANACAD's Esim tool [ANA94a], the model can be compiled using *amc*, the component is then instantiated in a circuit netlist and, finally, the circuit is simulated with Eldo and the resulting curves are displayed using the graphical tool Xelga.

In the Mentor environment, most of the tasks are implemented inside Design Architect. The compiler can be called from the ABSynth menu or using the regular procedure. Some back-annotation functions have been integrated in the ABSynth menu so that the icon of the component can be used directly to instantiate the component into a circuit diagram. Finally, the circuit can be simulated with Accusim II [Men95b], which can also display the result curves.

6.3 Results

In order to benchmark the tool ABSynth, we first measure the code generation time in various situations. Then we compare two ABSynth models with an HDL-A model coded manually. Our criterion will be code size, simulation results and simulation time. The benchmark example used is the elliptic low-pass filter already treated in the previous chapters. Code generation time and simulation time are CPU time measured on a Sun Sparc 20/712 workstation.

Code generation time

We generate HDL-A models of the low-pass filter in three different ways:

- Starting from the flat functional diagram of figure 4.6. The resulting code is given in Appendix B.
- Starting from the hierarchical functional diagram of figure 4.12. In a first run the generic code templates which correspond to the hierarchical GBSs are not available. They must be generated and can be saved in the user library. The complete code is not given here but the differences with respect to the flat model are explained in chapter 5.
- Starting once more from the hierarchical FD. In this second run, the previously generated generic code templates of the hierarchical GBSs are taken from the user library. The resulting code is the same as in the previous case.

The code generation time is displayed in table 6.1.

Model	Flat	Hierarchical (1st run)	Hierarchical (2nd run)
CPU time	45 s	49 s	16 s

Table 6.1 Code generation time.

We first see that code generation starting from a hierarchical description is somewhat slower than from a flat description. However, when generic code templates of hierarchical GBSs can be reused, the CPU time is reduced in an important manner—in this case by a factor of 3.

Code size

Table 6.2 displays the size of the different models generated as well as the size of the manually coded model given in Appendix B. We compare the number of code lines, the number of `states` and the number of `variables`.

Model	Manual	Flat	Hierarchical
# Lines	103	453	467
# States	8	55	63
# Variables	15	22	33

Table 6.2 Model code size.

Obviously, automatically generated code is less compact than code written directly by a modelling engineer. In this case, the biggest difference lies by the number of state variables necessary. This is not surprising because ABSynth uses a state variable for the output of a GBS and another one for the input of the GBS to which it is connected even if they both carry the same information. However, the flat description and the hierarchical description lead to code of the same complexity.

Simulation results

The different models are then simulated and we obtain in all cases exactly the same curves. The results of an AC simulation of the low-pass filter were already displayed in figure 3.9.

Simulation time

Simulation time, however can vary much from one model to another. Table 6.3 shows the simulation time needed by each model in the case of an AC simulation and in the case of a transient simulation.

Model	Manual	Flat	Hierarchical
AC simulation time	10 s	16 s	16 s
Transient simulation time	2.5 s	3.6 s	3.6 s

Table 6.3 Simulation time.

As the manually written model is much faster than the ones we generated with ABSynth, the generated code should be optimized. A first idea is to reduce the number of state variables and the number of lines of code. This could be done by modifying the code customization process in such a way that the output of a GBS would be represented by the same state variable as the input of the next GBS. This way one state variable and one assignment statement could be saved for each inter-GBS connection. This point has not been investigated further and it is left for future work.

6.4 Conclusions

In this chapter the actual implementation of ABSynth has been described. We saw that the tool consists of several independent programs, which can be called sequentially in a UNIX script. The modules have been implemented in C and they generate UNIX scripts to call the different executable modules and also external programs.

Furthermore many primitive code segments have been written in a sub-set of HDL-A as described in chapter 5. As we use a standard intermediate format (EDIF), the graphical modelling method could theoretically be implemented using any graphical entry tool. We chose to implement it inside Mentor Graphics Design Architect. After a model has been generated with ABSynth, it can be compiled and simulated either inside the ANACAD Eldo environment or inside the Mentor Graphics Framework using the compiler included in Design Architect and the Accusim II simulator. This way, our computer-aided modelling method is completely integrated in a commercial design environment; it is currently available on Sun Sparc workstations under Solaris 2.5 or SunOs 4.1.3. A user's guide is also available [Mos96].

Finally some results have been displayed. As the simulation of the ABSynth models produces the same curves as the simulation of models written by hand, the main goal is achieved. However the resulting code should be optimized for better simulation performance.

6.5 References

[ANA92] *ELDO-FAS Dynamical System Modeling,* Version 4.1.x, ANACAD Computer Systems, 1992.

[ANA94a] *HDL-A User's Manual,* Isaue 1.0, ANACAD Electrical Engineering Software, 1994.

[ANA94b] *HDL-A Language Reference Manual,* Iaaue 1.0, ANACAD Electrical Engineering Software, 1994.

[EIA87] *EDIF Electronic Design Interchange Format,* Versaion 2 0 0, EIA/ANSI Standard 548, Electronic Indnatries Association, 1987.

[Kah92] H. J. Kahn and R. F. Goldman, "The Electronic Design Interchange Format EDIF: Present and Future", *29th ACM/IEEE Design Automation Conference,* pp. 666-671, 1992.

[Lau93] R. Y. W Lau and H. J. Kahn, "Information Modelling of EDIF", *30th ACM/IEEE Design Automation Conference,* pp. 278-283, 1993.

[Men94a] *Design Architect User's Manual,* Software Veraion 8.4_1, 10/94, Mentor Graphics Corporation, 1994.

[Men94b] *Design Viewpoint Editor User's and Reference Manual,* Software Version 8.4_1, Mentor Graphics Corporation, 1994.

[Men95a] *EDIF Netlist User's and Reference Manual,* Software Version 8.4_2, Mentor Graphics Corporation, 1995.

[Men95b] *Analogue Simulators User's Manual,* Software Version 8.4_2, Mentor Graphica Corporation, 1995.

[Mos94] V. Moser et al., "A Graphical Approach to Analogue Behavioural Modelling", *The European Design and Test Conference,* pp. 535-539, 1994.

[Mos95] V. Moser et al., "Generating VHDL-A-like Models Using ABSynth", *EURO-DAC'95 European Design Automation Conference with EURO-VHDL'95,* pp. 522-527, 1995.

[Mos96] V. Moser and P. Nussbaum, *ABSynth User's guide,* IMT Report 380 PE 01/95, Version 1.1, Université de Neuchâtel, Institut de microtechnique, 1996.

Chapter 7
Application: Modelling of an RSD A/D Converter

As a more complete application example of behavioural modelling in general and of ABSynth in particular, a cyclic RSD (redundant signed digit) analogue to digital converter is modelled in HDL-A. Different variants of the converter model will be coded and compared with a MOS implementation in terms of accuracy and simulation time.

7.1 Principle of Operation

The Redundant Signed Digit (RSD) converter performs a successive approximation conversion based on a two-level comparison and produces a ternary output signal V_{out}. This A/D converter is discussed in detail in [Gin92]. Here, we only describe the RSD algorithm represented in figure 7.1.

The input signal V_{in} is compared with two threshold voltages $+V_{th}$ and $-V_{th}$. According to the result of the comparison, the first bit is expressed in a ternary code: it takes the value 1 for $V_{in}>V_{th}$, 0 for $V_{th}>V_{in}>-V_{th}$ and -1 for $-V_{th}>V_{in}$. Note that the result of the comparison is in fact a ternary moment, but we still use the term *bit* for the sake of readability. Additionally, the input signal is multiplied by 2 and a reference voltage V_r is either added or subtracted to obtain a new intermediate signal V_x. If $V_{in}>V_{th}$ then $V_x=2V_{in}-V_r$, if $V_{th}>V_{in}>-V_{th}$ then $V_x=2V_{in}$ and if $-V_{th}>V_{in}$ then $V_x=2V_{in}+V_r$. The same comparison algorithm is then applied recursively to V_x in order to

calculate the remaining output bits. As the first bit is used simultaneously as a sign bit and as the most significant bit, n cycles suffice to convert an input sample into a $(n+1)$-bit word.

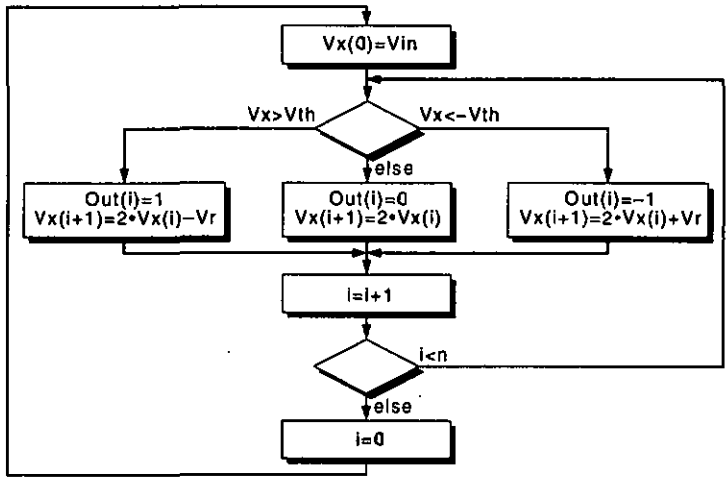


Fig. 7.1 Cyclic RSD conversion algorithm.

The convergence condition of this algorithm is $V_r > V_x > -V_r$. Hence, the reference voltage is chosen as the maximum possible value of the input signal. Accordingly, V_{th} must lie in the interval $[0, V_r/2]$. In practice, V_{th} is chosen equal to $V_r/4$, which provides a large margin for the inaccuracy of the comparators.

7.2 Behavioural Modelling

The RSD converter is now modelled in HDL-A. The ternary result is expressed using two binary signals: the positive one, V_{outp}, is set to high when V_x is 1 and to low otherwise, the negative one, V_{outn}, is set to high when V_x is –1 and to low otherwise. Different model variants are proposed and marked I to IV:

- The first one (I) is purely analogue—i.e., the core algorithm is expressed as analogue code, the interface connection points are analogue pins. This model will be referred to as the *analogue* model.

- Then, (model II) the algorithm is coded using digital constructs of HDL-A (i.e., in a `process` block), the interface connection points remaining analogue pins. This model will be referred to as the *mixed* model.
- Next (model III), the algorithm is coded in the digital part again but it is associated to a nearly completely digital interface. All the pins except the input pin are digital. This model will be referred to as the *digital* model, although it is actually a mixed-mode one, due to the analogue input pin.
- Finally, the converter is modelled with ABSynth, which obviously gives, again, an analogue model (IV).

To interpret the value of the output sample, the n-bit binary number that corresponds to V_{outn} is subtracted from the one that corresponds to V_{outp}. This difference must then be divided by the signal dynamics (2^n) and multiplied by the reference voltage V_r.

7.2.1 Analogue HDL-A Coding

The purely analogue version of the RSD converter model (model I) is composed of a purely analogue interface placed around an analogue implementation of the conversion algorithm.

Analogue Interface

This interface is composed of six analogue pins:

```
PIN (
    ANA_IN : electrical;
    OUTP : electrical;
    OUTN : electrical;
    CLK : electrical;
    VDD : electrical;
    VSS : electrical
    );
```

The input pin (`ana_in`) and the clock pin (`clk`) are modelled with a capacitive input impedance as shown for the input pin in the following code extract.

```
vin := ana_in.v;
iin := cin*ddt(vin);
ana_in.i %= iin;
```

The output pins `outp` and `outn` modelled with a resistive output impedance as:

```
ioutp := (outp.v - youtp)/rout;
outp.i %= ioutp;
```

Finally, a power supply stage (pins `vdd` and `vss`) is implemented as explained in § 3.1.1.

Analogue Core

The time representation used in this analogue representation is continuous but the algorithm to implement—the behaviour of the model—is intrinsically based on a sampled-time representation. Therefore, we resort to the sampling and shifting methods developed in sect. 3.3.

In order to synchronize the computation of the algorithm with the clock signal V_{clk}, we trigger off the execution of the corresponding code block on the rising edge of the clock.

```
IF RISING(vclk,clk_thr) THEN
```

First, the sampling period `tdel` is computed,

```
tmem := current_time;
tsam := (current_time - previous(tmem));
tdel := real(tsam);
```

then, the intermediate state variable `vxin` is loaded either with the value of the input signal—to compute a new input sample—or with the last computed value of V_x,

```
IF (previous(count)=n-1.0) THEN
    count := 0.0;
    vxin := previous(vx,tdel);
ELSIF (previous(count)=0.0) THEN
    count := 1.0;
    vxin := vin;
ELSE
    count := previous(count) + 1.0;
```

```
        vxin := previous(vx,tdel);
    END IF;
```

then, V_x is calculated according to the actual algorithm; the two outputs V_{outp} and V_{outn} are set to VDD to code a 1 and to VSS to code a 0.

```
    IF (vxin>v_thr) THEN
        vx := 2.0*vxin - v_ref;
        voutp := vvdd;
        voutn := vvss;
    ELSIF (vxin<-v_thr) THEN
        vx := 2.0*vxin + v_ref;
        voutp := vvss;
        voutn := vvdd;
    ELSE
        vx := 2.0*vxin;
        voutp := vvss;
        voutn := vvss;
    END IF;
END IF;
```

Finally, the two outputs are set to a non-working value `v_rest` during the second half clock period.

```
IF FALLING(vclk,clk_thr) THEN
    voutp := v_rest;
    voutn := v_rest;
END IF;
```

To implement this algorithm and the interface stages, several `generic` parameters were necessary: the reference voltage, the threshold voltage, the number of conversion cycles, the non-working value of the output signals and various parameters related to the interface blocks.

7.2.2 Mixed-Mode HDL-A Coding

We coded two mixed-mode implementations. The core of the algorithm is coded in models II and III as a digital `process` statement using the mechanism described in sect. 3.3 for the coding of a difference equation. The analogue interface of model II was already described in

the previous section. The digital interface used in model III, however, has been highly simplified as we will see below.

Digital Interface

In this implementation, the output signals are connected to digital *bit* output ports, the clock is a *bit* input port, the power supply has been removed since it is not relevant anymore at this level of abstraction. Only the input remains an analogue pin associated with a capacitive input stage.

```
PORT (
     CLK : IN bit;
     OUTP : OUT bit;
     OUTN : OUT bit
     );
PIN (
    ANA_IN : electrical
    );
```

Digital Core

The digital core is the same for the two variants of the mixed-mode converter model. Again, the computation is triggered off by the rising edge of the clock signal, which is now digital.

```
WAIT ON clk UNTIL event(clk) AND clk='1';
```

The variable vxin now has to carry either a sample of the input or the previous value of V_x, which is automatically shifted.

```
IF (count=n-1.0) THEN
    count <= 0.0;
    vxin := vx;
ELSIF (count=0.0) THEN
    count <= 1.0;
    vxin := vin;
ELSE
    count <= count + 1.0;
    vxin := vx;
END IF;
```

Finally, V_x is computed according to the actual algorithm; the two digital outputs `outp` and `outn` are set to 1 or 0. This value is now maintained during the whole clock cycle because it is not possible to define an intermediate non-working value.

```
    IF (vxin>v_thr) THEN
        vx <= 2.0*vxin - v_ref;
        outp <= '1';
        outn <= '0';
    ELSIF (vxin<-v_thr) THEN
        vx <= 2.0*vxin + v_ref;
        outp <= '0';
        outn <= '1';
    ELSE
        vx <= 2.0*vxin;
        outp <= '0';
        outn <= '0';
    END IF;
END LOOP;
...
END PROCESS;
```

7.2.3 ABSynth Modelling

The RSD converter has also been modelled using ABSynth (model IV). The behaviour considered here is exactly the same as the behaviour coded in the analogue model I. We first give the corresponding functional diagram, then we describe two specific GBSs in detail.

Functional Diagram

The functional diagram of figure 7.2 describes the behaviour of the RSD converter. The input/output interface is modelled using usual hierarchical GBSs. The power supply part graphically symbolizes the current balance-sheet already mentioned in sect. 3.2. The RSD algorithm, however, required more attention. As it is intrinsically a procedural behaviour it could not be described conveniently using the standard GBS library and, therefore, a new dedicated GBS had to be defined. Its ternary output signal is transmitted to another new GBS,

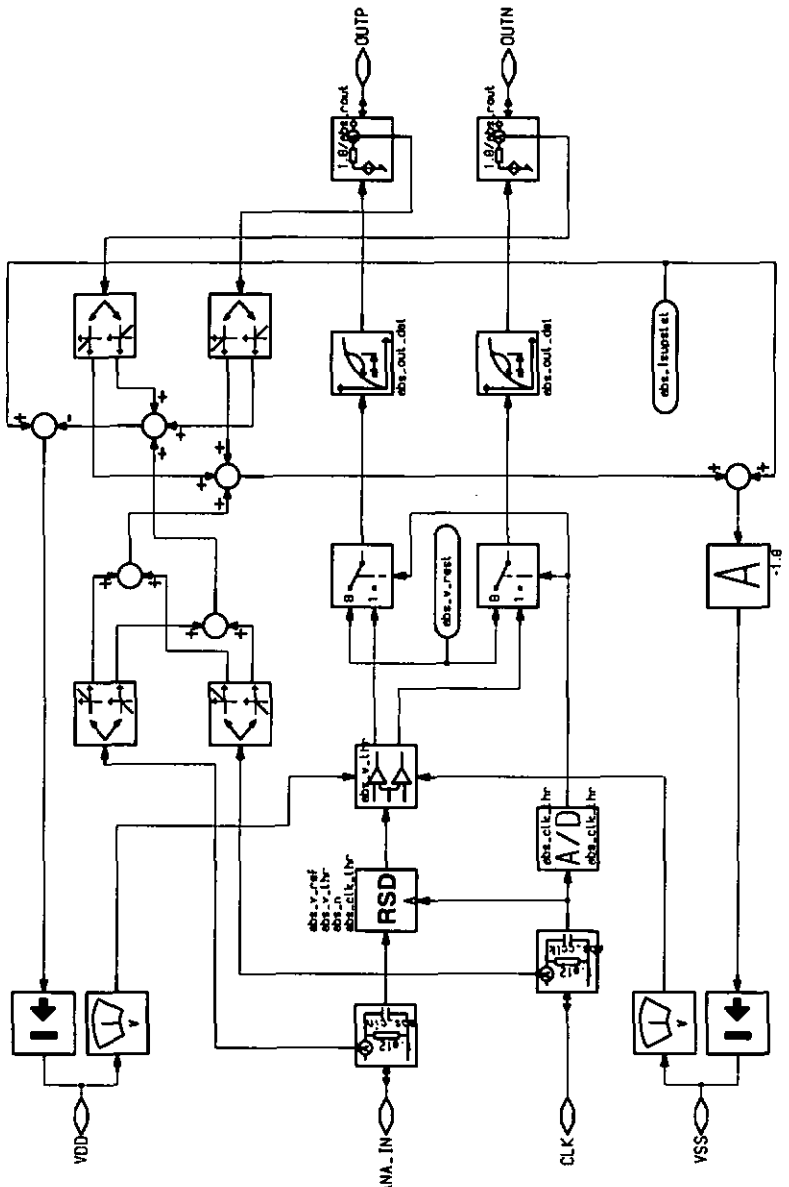


Fig. 7.2 Functional diagram of the RSD converter.

which performs a double comparison and generates both positive and negative signals. The clock signal is first used to trigger off the RSD GBS. Then, it is transformed into a binary signal that controls a pair of multiplexers so that either the result of the conversion or the non-working value is transmitted to the output stage. Finally, a constant delay GBS is added.

Redundant Signed Digit GBS

As explained above, the RSD algorithm has been coded as a new GBS. The icon has three pins—a continuous input for V_{in}, a ternary output for V_x and an input for the clock signal—and four parameters—the reference voltage, the threshold voltage, the number of conversion cycles and the clock edge detection threshold. The behaviour is expressed as a code template according to the syntax given in sect. 5.4. Obviously, this GBS code template is very similar to the code commented in § 7.2.1.

Double Comparator GBS

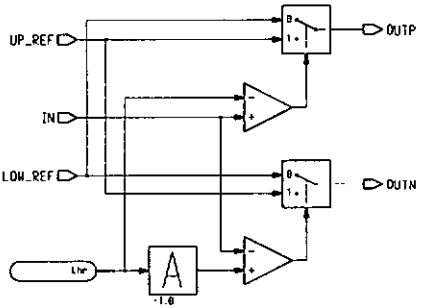


Fig. 7.3 Sub- functional diagram of the double comparator GBS.

The double comparison function can be easily realized using standard GBSs. However, for the sake of readability, a new hierarchical GBS has been defined. It has five pins—one input for the ternary signal, two outputs for the final binary signals and two other inputs for the reference output levels—and one parameter—a

threshold vth used to interpret the value of the ternary input signal. The functionality is simple, as can be seen in the corresponding sub-FD of figure 7.3.

The positive output is set to 1—i.e., it takes the value of the positive reference—when the input signal is 1. It is set to 0—i.e., the value of the negative reference—otherwise. Similarly, the negative output is set to 1 when the input signal is –1 and to 0 otherwise.

7.2.4 Code Size

Before going towards simulation of the models, we give in table 7.1 some interesting figures about code size and variable number.

Model	# Lines	# State	# Signals	# Variables
I (analogue)	200	11	0	13
II (mixed-mode)	212	8	5	14
III (digital)	82	1	3	1
IV (ABSynth)	906	126	0	28

Table 7.1 Simulation time (CPU) needed for the conversion of one sample with ELDO on Sun Sparc 10/30.

The analogue and mixed-mode models are quite similar in size but the digital one is much smaller because the interface part is reduced. The ABSynth model, however, is much bigger, which leaves room for optimization, as explained in chapter 6.

7.3 Simulation and Comparison

The different models have been simulated and the results are compared with the simulation of an 8-bit CMOS circuit implementation (model V) developed by Heubi [Heu96] and used by Grisoni as the core of a relative precision A to D converter [Gri95]. The conditions of the experiment are summarized in table 7.2.

Sample frequency	16 kHz
Number of bits n	8
Clock frequency	128 kHz
Reference voltage V_r	1.3 V
Supply voltage	±1.3 V
Non-working output value	0.5736 V
Output delay	50 ns

Table 7.2 Parameter settings for the RSD converter simulation.

We first show in figure 7.4 the shape of the curves obtained for one conversion on the analogue outputs (model I) and on the digital outputs (model III).

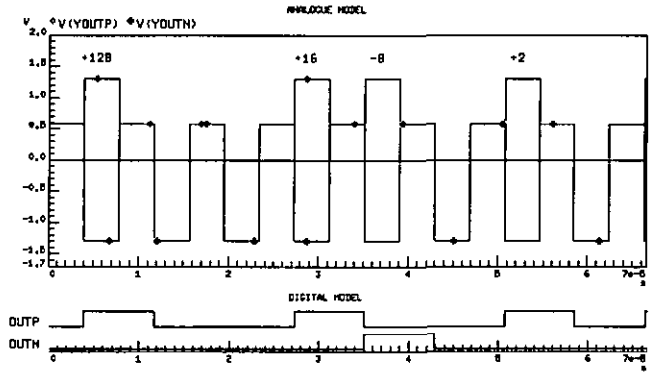


Fig. 7.4 Analogue and digital output signals of the RSD converter modela.

The value of the output signals can be interpreted as

$$\frac{128+16-8+2}{256} \cong 0.54 \cong \frac{0.7}{1.3}$$

In order to compare the behaviour of the different models, we plot—in figure 7.5—the positive analogue output of models I (analogue), II (mixed) and IV (ABSynth).

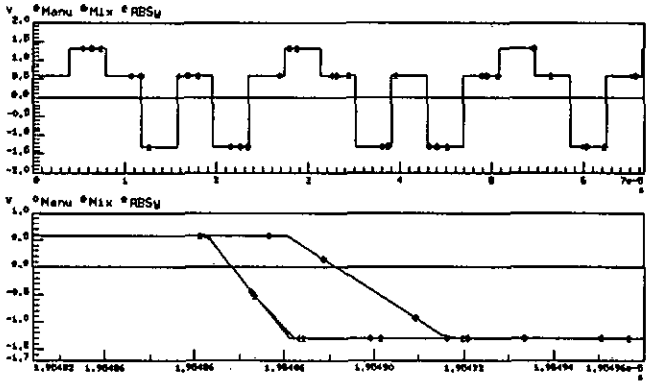


Fig. 7.5 Analogue output signals of different model variants.

Obviously, the primary behaviour—the result of the comparison—is the same. When we zoom in on a falling edge of the curve, however, small differences appear. In this case, the mixed model exhibits a negligible delay of a few hundreds of picoseconds.

Finally, we compare (figure 7.6) the positive output signal of the analogue model I with the positive output signal of the simulated CMOS circuit.

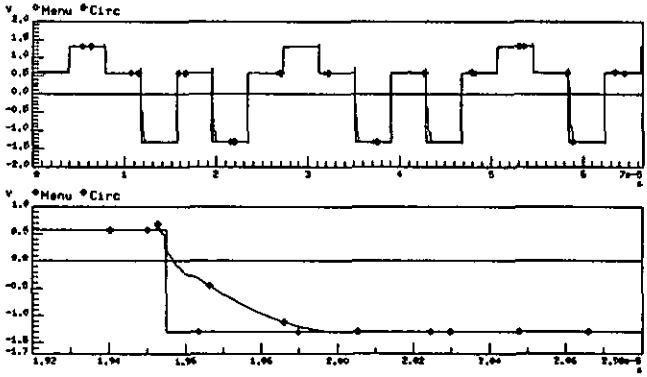


Fig. 7.6 Output signals of the analogue model and of the circuit.

Again, at first sight, the two curves match, but the circuit shows an additional secondary effect: an exponential delay appears at a

falling edge of the signal. As a first solution towards modelling this effect, an exponential delay—as described in sect. 3.2—could be added to the model. However, as the delay is not the same at the rising edge of the signal as at the falling edge, the model would become more complicated. It always remains the designer's choice to decide which behaviour must be included in the model and which aspect can be left aside.

Beside curve shape, another important criterion for model performance comparison is the simulation time needed to convert one input sample. Table 7.3 gives the values measured for the various models.

Model	Simulation time
I (analogue)	12s 916ms
II (mixed-mode)	15s 133ms
III (digital)	0s 250ms
IV (ABSynth)	50s 800ms
V (CMOS)	6mn 28s 483ms

Table 7.3 Simulation time (CPU) needed for the conversion of one sample with ELDO on Sun Sparc 10/30.

The simulation time of the analogue model (I) and of the mixed model (II) are very close. However, model III, which is almost completely digital, is much faster. This is partly due to the loss of detail in the digital interface representation. The ABSynth model (IV) is 4 to 5 times slower than the analogue and mixed models. This drawback comes from the fact that the code generated by ABSynth is not optimal—i.e., many intermediate variables are present. Lastly, we note that the behavioural models are at least 7 times faster than the CMOS circuit representation. Therefore, any of them could be used advantageously in a system-level simulation.

7.4 Conclusions

A successive approximation redundant signed digit analogue to digital converter has been modelled in HDL-A in several different ways. The

comparison of the different models variants with a CMOS circuit representation of the same system shows that a good approximation of the behaviour can be achieved and that the simulation performance increases significantly using behavioural modelling.

One of the behavioural models was coded with ABSynth. The functional diagram we obtained provides a useful insight in the modelled behaviour, which can be considered as an advantage of our graphical approach versus manual HDL-A coding. The FD has been drawn using many standard GBSs to model the interface of the model and some classical functions—e.g., delay, clock signal handling. For the typical behaviour or the RSD converter, however, two new GBSs were defined. A double comparison GBS was defined hierarchically for the sake of clarity. The second one, which stands for the actual conversion algorithm, could not be described using general purpose GBSs because our graphical method is not well-suited to describe algorithms. Therefore, we had to give the corresponding semantics directly as a new code template. This coding task requires some experience in HDL-A and is thus reserved to an expert user. Therefore, we consider this point as a weakness of ABSynth. To improve the tool, a code template editor should be developed in order to ensure the matching between the GBS interface and the code template declarations.

The results obtained with the different behavioural models are good and the simulation time has been reduced. However, the ABSynth model is slower than the semantically equivalent analogue model. We claim that this is due to the non-optimal generated code.

7.5 References

[Gin92] B. Ginetti et al., "A CMOS 13-b Cyclic RSD A/D Converter", *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 7, pp. 957-964, 1992.

[Gri95] L. Grisoni et al., "Micro Power 'Relative Precision' 15-bit A/D Converter", *The International Conference on Signal Processing Applications & Technology*, pp. 420-424, 1995.

[Heu96] A. Heubi et al., "Micro Power 'Relative Precision' 13 bits Cyclic RSD A/D Converter", *IEEE International Symposium on Low Power Electronics and Design*, 1996.

Chapter 8
Conclusions

In this dissertation, we first introduced analogue behavioural modelling, then we searched a way to support designers in modelling analogue systems. The solution we propose is twofold. First, the users have to graphically describe the behaviour of the system they want to model. Second, this graphical description is translated into an analogue hardware description language.

8.1 Main Contributions

A first contribution of this research is a method for the graphical description of analogue behaviour. We introduce the *functional diagram,* an extended block diagram, which allows the users to build complex behavioural descriptions using behavioural building blocks called *graphical building symbols.* The users can clearly define the boundary between the internal behavioural description based on signal flow variable processing and the external interface of the system based on the exchange of physical quantities.

A second contribution is a code generation strategy developed to translate the semantics of a functional diagram into behavioural HDL code. This strategy includes the handling of hierarchical and analysis-specific functional diagrams.

Lastly, the graphical description method and the code generation strategy have been implemented as a software tool called ABSynth (Analogue Behavioural model Synthesizer). ABSynth is the first wide-purpose analogue behavioural model generator based on a dedicated

analogue HDL. It takes care of the target HDL syntax, it is user-friendly, easy to extend towards new analogue functions, and integrated in a commercial design framework. However, it is still modular and would be easy to port to another target language or to another design environment. ABSynth has been validated using workbench examples—an analogue filter and an A/D converter. Simulation results of ABSynth models were compared with simulation results of "hand-typed" HDL models containing the same semantics. This showed that the approach is valid—the resulting curves are identical—but also that the generated code should be improved to achieve better simulation speed. For instance, a reduction of the number of state variables could be considered.

8.2 Fundamental Limitations

Several limitations have been mentioned in the preceding chapters. Some of them are linked to the modelling method itself and can therefore be considered as fundamental limitations.

First of all, even if we investigated mixed-mode modelling, the methods we developed are limited to analogue models.

Besides, the whole modelling method proposed here is based on a "divide-and-conquer" strategy: the users have to divide the behaviour of the component to model into elementary pieces in order to describe it in the form of a functional diagram. Unfortunately, this method cannot be applied to systems which exhibit a tightly coupled behaviour. This intrinsic limitation is probably the main drawback of this method.

Additionally, even if the graphical description method is quite general, the code generation strategies we developed are linked to the target language HDL-A. For this reason, the structure of the tool ABSynth reflects the structure of HDL-A. We were also limited to purely behavioural code. However, we believe that it will be possible to update ABSynth towards VHDL-AMS.

Finally, some problems were not investigated further. For instance, algebraic loops drawn in the functional diagram are kept unchanged in the generated model code and may therefore lead to erroneous models if the associated simulator cannot solve them.

8.3 Future Work

Some other limitations are related to the implementation and could therefore be pushed away in the future.

- An IEEE 1076.1 VHDL-AMS generator should be developed when this language is available.
- A more complete sub-set of the final language could also be implemented, including, e.g., *couplings* and wide pins (buses).
- An interface towards digital modelling could be provided based, for instance, on a digital black-box used to pack existing digital HDL code.
- Similarly, another description method should be available to generate complex algorithmic code.
- An assisted editor should also be provided to the users who still need to type some new GBS code templates.
- Lastly, the tool ABSynth could be optimized in order to reduce code generation time and generated code size.

8.4 Final Remarks

Finally, it seems important to emphasize that, whatever the modelling tools available, engineers first need to have a good understanding of the system they want to model. Secondly, they must know which effects have to be included in the model. Then, and only then, computer-aided modelling tools can be used efficiently.

In this context, we hope that the proposed modelling method and the tool ABSynth, despite the limitations mentioned above, can be a step towards automated analogue modelling and consequently a contribution, albeit a modest one, to the palette of analogue CAD tools.

Appendix A
Program Usage and Options

We detail here the command line syntax and the meaning of the various options of the executable programs which compose the tool ABSynth.

A.1 Code Synthesizer Script

```
hsynth <design> [-e] [-f] [-s] [-t]
        [ [-A <AC view>] [-D <DC view>] [-T <transient view>]
        | [-V <view>] ] [-O <output file name>]
        [-S <system library path>] [-U <user library path>]
        [-W <work dir>] [-X <HDL extension>]
```

```
or   hsynth -help
```

The various options mean:

<design> Path name to a Mentor Graphics component.

–e Entity only; no architecture description will be generated. Default: architecture description is generated.

–f Flat; do not go down into the description of hierarchical GBSs. Default: hierarchical descriptions are used.

–s Save new primitive models of hierarchical GBSs in the user library (see the –U option). Default: new primitive models are not saved.

–t Test the matching between EDIF cells and the corresponding primitive HDL-A model. Default: no test.

–A Set multiple FD mode; name of the AC view to process. Default: single FD mode.

–D Set multiple FD mode; name of the DC view to process. Default: single FD mode.

–O Path name to the output files; the HDL-A source file gets the extension *.absynth_source*, the back-annotation file gets the extension *.absynth_back*. Default: *./name of the design.*

–S Path to the system primitive HDL-A model library. Default: path set in the environment variable *ABS_SYS_LIB* if defined, otherwise current directory.

–T Set multiple FD mode; name of the transient view to process. Default: single FD mode.

–U Path to the user primitive HDL-A model library. Default: path set in the environment variable *ABS_USR_LIB* if defined, otherwise current directory.

–V Set single FD mode; name of the view to process. Default: single FD mode; default schematic of the component.

–W Path to the working directory. Default: path set in the environment variable *ABS_WORK_DIR* if defined, otherwise */tmp.*

–X Extension of the generic HDL-A code templates. Default: *.hdla.*

–help Display this message only.

A.2 Entity Generation Program

```
hdent < <file> [-h] [-t] [-S <system library path>]
               [-U <user library path>] [-W <work dir>]
               [-X <HDL extension>]
```

The various options mean:

<file> Path name to the EDIF input file.

–h Display this message only.

–t Tests the matching between EDIF cells and the corresponding primitive HDL-A model. Default: no test.

–S Path to the system primitive HDL-A model library. Default: path set in the environment variable *ABS_SYS_LIB* if defined, otherwise current directory.

–U Path to the user primitive HDL-A model library. Default: path set in the environment variable *ABS_USR_LIB* if defined, otherwise current directory.

–W Path to the working directory. Default: path set in the environment variable *ABS_WORK_DIR* if defined, otherwise */tmp*.

–X Extension of the generic HDL-A code templates. Default: *.hdla*.

A.3 Architecture Generation Program

```
hdarc < <file> [-f] [-h] [-s] [-t] [[-A] | [-D] | [-T]]
               [-S <system library path>]
               [-U <user library path>] [-V <architecture>]
               [-W <work dir>] [-X <HDL extension>]
```

The various options mean:

<file> Path name to the EDIF input file.

–f Flat; doesn't go down into the description of hierarchical cells. Default: hierarchical descriptions are used.

–h Display this message only.

–s Save new primitive models of hierarchical cells in the user library (see the –U option). Default: new primitive models are not saved.

–t Tests the matching between EDIF cells and the corresponding primitive HDL-A model. Default: no test.

–A Partial code generation: only AC and common blocks. Default: complete code generation.

-D Partial code generation: only DC and common blocks. Default: complete code generation.

-S Path to the system primitive HDL-A model library. Default: path set in the environment variable *ABS_SYS_LIB* if defined, otherwise current directory.

-T Partial code generation: only transient and common blocks. Default: complete code generation.

-U Path to the user primitive HDL-A model library. Default: path set in the environment variable *ABS_USR_LIB* if defined, otherwise current directory.

-V Name of the architecture. Default: name of the first view of the design in the EDIF file.

-W Path to the working directory. Default: path set in the environment variable *ABS_WORK_DIR* if defined, otherwise */tmp*.

-X Extension of the generic HDL-A code templates. Default: *.hdla*.

A.4 Code Customization Program

```
hdlagen < <GBS code template> "<instance name>"
          [<keyword>=<identifier>]
          [<in pin>,"<prev inst name>",<prev out pin>]
          [<property>="<value>"] [-o]
```

The various options mean:

<GBS code template>
Name of the code template file which correspond to the current GBS.

<keyword>=<identifier>
Generic keyword to replace, replacement name. Used to replace a keyword of the current GBS code by a particular identifier.

<instance name>
Name of the current GBS instance. Used to customize the names of the identifiers.

<in pin>,"<prev inst name>",<prev out pin>
: Name of an input pin of the current instance, name of the previous instance, name of an output pin of the previous instance. Used to connect an input pin of the current instance with an output pin of the previous instance.

<property>="<value>"
: Name of a property, expression. Used to attribute a value to a property of the current GBS.

–o
: Override pin ignorance. Handle pin names. Default: Do not handle pins.

A.5 Code Selection Program

```
hdlaselect [-K<keyword>] [-S["<separator>"]]
```

–K
: Keyword that represents the model part which must be selected. Authorized keywords are *GENERIC, PIN, DECLARE, init, pr_dc, pr_ac, pr_tr, ivar_dc, ivar_ac, ivar_tr, eq_dc, eq_ac, eq_tr.*

–S
: Character that must be appended to the selected string. Authorized separators are ";" and ",". If no separator character is given, a new line is added.

Appendix B
HDL-A Code Example

We give here different variants of the HDL-A code of the fifth order elliptic low-pass filter model used as an example in chapters 3 to 6. First the model has been written by hand (sect. 3.4), then it has been generated automatically starting from a flat single-FD description (§ 5.4.4).

B.1 Manual Coding

```
ENTITY ellip5 IS
    GENERIC (
            cin : real;
            isupstat : real;
            omega1 : real;
            omega2 : real;
            omega3 : real;
            omega4 : real;
            rout : real;
            sigma0 : real;
            sigma1 : real;
            sigma3 : real
            );
    PIN (
        SIG_IN : electrical;
        SIG_OUT : electrical;
        VDD : electrical;
        VSS : electrical
        );
END ENTITY ellip5;
ARCHITECTURE manual OF ellip5 IS
    STATE vin : analog;
    STATE int1, int2, int3, int4 : analog;
    STATE dint2, dint3, dint4 : analog;
```

```
    VARIABLE iin, iinn, iinp : analog;
    VARIABLE iout, ioutn, ioutp : analog;
    VARIABLE int5 : analog;
    VARIABLE b11, b12, b22, b13, b23, a24, a25 : real;
BEGIN
    RELATION
        PROCEDURAL FOR init =>
            cin := 1.0e-14;
            isupstat := 0.0;
            omega1 := 1.0630;
            omega2 := 2.1187;
            omega3 := 0.8012;
            omega4 := 1.4396;
            rout := 100.0;
            sigma0 := -0.68099;
            sigma1 := -0.10398;
            sigma3 := -0.41063;

            b11 := (-1.0)/(twopi*sigma0);
            b12 := (-2.0*sigma3)/ (twopi*(sigma3*sigma3+omega3*omega3));
            b22 := 1.0/ (twopi*twopi*(sigma3*sigma3+omega3*omega3));
            b13 := (-2.0*sigma1)/ (twopi*(sigma1*sigma1+omega1*omega1));
            b23 := 1.0/ (twopi*twopi*(sigma1*sigma1+omega1*omega1));
            a24 := 1.0/(twopi*twopi*omega2*omega2);
            a25 := 1.0/(twopi*twopi*omega4*omega4);

            vin := 0.0;
            int1 := 0.0;
            int2 := 0.0;
            int3 := 0.0;
            int4 := 0.0;
            dint2 := 0.0;
            dint3 := 0.0;
            dint4 := 0.0;
            iin := 0.0;
            iinn := 0.0;
            iinp := 0.0;
            iout := 0.0;
            ioutn := 0.0;
            ioutp := 0.0;
            int5 := 0.0;
        PROCEDURAL FOR dc, ac, transient =>
            vin := sig_in.v;
            iin := cin*ddt(vin) + 1.0e-12*vin;
            sig_in.i %= iin;
            dint2 := ddt(int2);
            dint3 := ddt(int3);
            int4 := int3 + a24 * ddt(dint3);
            dint4 := ddt(int4);
            int5 := int4 + a25 * ddt(dint4);
            iout := (sig_out.v - int5)/rout;
            sig_out.i %= iout;
            IP (iin>0.0) THEN
                iinp := iin;
                iinn := 0.0;
            ELSE
                iinp := 0.0;
                iinn := iin;
```

```
            END IF;
            IF (iout>0.0) THEN
                ioutp := iout;
                ioutn := 0.0;
            ELSE
                ioutp := 0.0;
                ioutn := iout;
            END IF;
            vdd.i %= isupstat - iinn - ioutn;
            vss.i %= - (isupstat + iinp + ioutp);
        EQUATION (
                 int1,
                 int2,
                 int3
                 ) FOR dc, ac, transient =>
            vin - b11 * ddt(int1) - int1 == 0.0;
            int1 - b12 * dint2 - b22 * ddt(dint2) - int2 == 0.0;
            int2 - b13 * dint3 - b23 * ddt(dint3) - int3 == 0.0;
    END RELATION;
END ARCHITECTURE manual;
```

B.2 Flat Single-FD Generated Code

```
-- Cell name :     el15
-- Architecture :  flat
-- Language :      HDL-A
-- Date :          Fri Feb  2 17:39:46 1996
-- Author :        IMT UniNE & CSEM SA, Neuchatel, Switzerland
-- Program :       ABSynth: Analogue Behavioural model Synthesizer

ENTITY el15 IS
    GENERIC (
            abs_cin : real;
            abs_isupstat : real;
            abs_omega1 : real;
            abs_omega2 : real;
            abs_omega3 : real;
            abs_omega4 : real;
            abs_rout : real;
            abs_sigma0 : real;
            abs_sigma1 : real;
            abs_sigma3 : real
            );
    PIN (
        SIG_IN : electrical;
        SIG_OUT : electrical;
        VDD : electrical;
        VSS : electrical
        );
END ENTITY el15;
ARCHITECTURE flat OF el15 IS
    STATE I690_out : analog;
    VARIABLE I690_parameter : real;
    STATE I705_out : analog;
    STATE I712_out : analog;
    STATE I693_in, I693_out : analog;
```

```
    STATE I697_in, I697_out : analog;
    VARIABLE I697_gain : real;
    STATE I706_in, I706_out : analog;
    VARIABLE I706_p : real;
    STATE I696_in, I696_out : analog;
    VARIABLE I696_gain : real;
    STATE I707_in, I707_out, I707_dout : analog;
    VARIABLE I707_rp, I707_ip : resl;
    VARIABLE I707_b1, I707_b2 : real;
    STATE I704_in1, I704_in2, I704_out : analog;
    STATE I708_in, I708_out, I708_dout : analog;
    VARIABLE I708_rp, I708_ip : real;
    VARIABLE I708_b1, I708_b2 : real;
    STATE I695_in : analog;
    STATE I703_in, I703_out1, I703_out2 : analog;
    STATE I709_in, I709_din, I709_out : analog;
    VARIABLE I709_rz, I709_iz : resl;
    VARIABLE I709_a1, I709_a2 : real;
    STATE I710_in, I710_din, I710_out : analog;
    VARIABLE I710_rz, I710_iz : real;
    VARIABLE I710_a1, I710_a2 : real;
    STATE I714_in1, I714_in2, I714_out : snalog;
    STATE I711_in, I711_out : analog;
    VARIABLE I711_gain : real;
    STATE I702_in, I702_out1, I702_out2 : analog;
    STATE I713_in : analog;
    STATE I688_in1, I688_in2, I688_out : analog;
    STATE I691_in1, I691_in2, I691_out : analog;
    STATE I700_in1, I700_in2, I700_out : analog;
    STATE I701_in1, I701_in2, I701_out : analog;
    STATE I698_in : analog;
    STATE I699_in, I699_out : analog;
    VARIABLE I699_gain : real;
    STATE I692_in : analog;
BEGIN
    RELATION
        PROCEDURAL FOR init =>
            abs_cin := 1.0e-14;
            abs_isupatat := 0.0;
            abs_omega1 := 1.0630;
            abs_omega2 := 2.1187;
            abs_omega3 := 0.8012;
            abs_omega4 := 1.4396;
            abs_rout := 100.0;
            abs_sigma0 := -0.68099;
            abs_sigma1 := -0.10398;
            abs_sigma3 := -0.41063;
            I690_out := 0.0;
            I690_parameter := abs_Isupstat;
            I705_out := 0.0;
            I712_out := 0.0;
            I693_in := 0.0;
            I693_out := 0.0;
            I697_in := 0.0;
            I697_out := 0.0;
            I697_gain := 1.0e-12;
            I706_in := 0.0;
            I706_out := 0.0;
```

```
I706_p := abs_sigma0;
I696_in := 0.0;
I696_out := 0.0;
I696_gain := abs_cin;
I707_in := 0.0;
I707_out := 0.0;
I707_dout := 0.0;
I707_rp := abs_sigma3;
I707_ip := abs_omega3;
I707_b1 :=(-2.0*I707_rp)/
          (twopi*(I707_rp*I707_rp+I707_ip*I707_ip));
I707_b2 := 1.0/ (twopi*twopi*(I707_rp*I707_rp+I707_ip*I707_ip));
I704_in1 := 0.0;
I704_in2 := 0.0;
I704_out := 0.0;
I708_in := 0.0;
I708_out := 0.0;
I708_dout := 0.0;
I708_rp := abs_sigma1;
I708_ip := abs_omega1;
I708_b1 := (-2.0*I708_rp)/
           (twopi*(I708_rp*I708_rp+I708_ip*I708_ip));
I708_b2 := 1.0/(twopi*twopi*(I708_rp*I708_rp+I708_ip*I708_ip));
I695_in := 0.0;
I703_in := 0.0;
I703_out1 := 0.0;
I703_out2 := 0.0;
I709_in := 0.0;
I709_din := 0.0;
I709_out := 0.0;
I709_rz := 0.0;
I709_iz := abs_omega2;
I709_a1 := (-2.0*I709_rz)/
             (twopi*(I709_rz*I709_rz+I709_iz*I709_iz));
I709_a2 := 1.0/(twopi*twopi*(I709_rz*I709_rz+I709_iz*I709_iz));
I710_in := 0.0;
I710_din := 0.0;
I710_out := 0.0;
I710_rz := 0.0;
I710_iz := abs_omega4;
I710_a1 := (-2.0*I710_rz)/
             (twopi*(I710_rz*I710_rz+I710_iz*I710_iz));
I710_a2 := 1.0/(twopi*twopi*(I710_rz*I710_rz+I710_iz*I710_iz));
I714_in1 := 0.0;
I714_in2 := 0.0;
I714_out := 0.0;
I711_in := 0.0;
I711_out := 0.0;
I711_gain := 1.0/abs_rout;
I702_in := 0.0;
I702_out1 := 0.0;
I702_out2 := 0.0;
I713_in := 0.0;
I688_in1 := 0.0;
I688_in2 := 0.0;
I688_out := 0.0;
I691_in1 := 0.0;
I691_in2 := 0.0;
```

```
        I691_out := 0.0;
        I700_in1 := 0.0;
        I700_in2 := 0.0;
        I700_out := 0.0;
        I701_in1 := 0.0;
        I701_in2 := 0.0;
        I701_out := 0.0;
        I698_in := 0.0;
        I699_in := 0.0;
        I699_out := 0.0;
        I699_gain := -1.0;
        I692_in := 0.0;
    PROCEDURAL FOR dc =>
        I690_out := I690_parameter;
        I705_out := SIG_IN.v;
        I712_out := SIG_OUT.v;
        I693_in := I705_OUT;

        I693_out := ddt(I693_in);
        I697_in := I705_OUT;

        I697_out := I697_gain * I697_in;
        I706_in := I705_OUT;

        I696_in := I693_OUT;

        I696_out := I696_gain * I696_in;
        I707_in := I706_OUT;

        I707_dout := ddt(I707_out);
        I704_in1 := I696_OUT;
        I704_in2 := I697_OUT;

        I704_out := I704_in1 + I704_in2 ;
        I708_in := I707_OUT;

        I708_dout := ddt(I708_out);
        I695_in := I704_OUT;

        SIG_IN.i %= I695_in;
        I703_in := I704_OUT;

        IF (I703_in > 0.0) THEN
            I703_out1 := I703_in;
            I703_out2 := 0.0;
        ELSE
            I703_out1 := 0.0;
            I703_out2 := I703_in;
        END IF;
        I709_in := I708_OUT;

        I709_din := ddt(I709_in);
        I709_out := I709_in + I709_a1 * I709_din +
                    I709_a2 * ddt(I709_din);
        I710_in := I709_OUT;

        I710_din := ddt(I710_in);
        I710_out := I710_in + I710_a1 * I710_din +
```

```
                    I710_a2 * ddt(I710_din);
    I714_in1 := I712_OUT;
    I714_in2 := I710_OUT;

    I714_out := I714_in1 - I714_in2 ;
    I711_in := I714_OUT;

    I711_out := I711_gain * I711_in;
    I702_in := I711_OUT;

    IF (I702_in > 0.0) THEN
        I702_out1 := I702_in;
        I702_out2 := 0.0;
    ELSE
        I702_out1 := 0.0;
        I702_out2 := I702_in;
    END IF;
    I713_in := I711_OUT;

    SIG_OUT.i %= I713_in;
    I688_in1 := I703_OUT1;
    I688_in2 := I702_OUT1;

    I688_out := I688_in1 + I688_in2 ;
    I691_in1 := I703_OUT2;
    I691_in2 := I702_OUT2;

    I691_out := I691_in1 + I691_in2 ;
    I700_in1 := I688_OUT;
    I700_in2 := I690_OUT;

    I700_out := I700_in1 + I700_in2 ;
    I701_in1 := I690_OUT;
    I701_in2 := I691_OUT;

    I701_out := I701_in1 - I701_in2 ;
    I698_in := I701_OUT;

    VDD.i %= I698_in;
    I699_in := I700_OUT;

    I699_out := I699_gain * I699_in;
    I692_in := I699_OUT;

    VSS.i %= I692_in;
PROCEDURAL FOR ac =>
    I690_out := I690_parameter;
    I705_out := SIG_IN.v;
    I712_out := SIG_OUT.v;
    I693_in := I705_OUT;

    I693_out := ddt(I693_in);
    I697_in := I705_OUT;

    I697_out := I697_gain * I697_in;
    I706_in := I705_OUT;

    I696_in := I693_OUT;
```

```
I696_out := I696_gain * I696_in;
I707_in := I706_OUT;

I707_dout := ddt(I707_out);
I704_in1 := I696_OUT;
I704_in2 := I697_OUT;

I704_out := I704_in1 + I704_in2 ;
I708_in := I707_OUT;

I708_dout := ddt(I708_out);
I695_in := I704_OUT;

SIG_IN.i %= I695_in;
I703_in := I704_OUT;

IF (I703_in > 0.0) THEN
    I703_out1 := I703_in;
    I703_out2 := 0.0;
ELSE
    I703_out1 := 0.0;
    I703_out2 := I703_in;
END IF;
I709_in := I708_OUT;

I709_din := ddt(I709_in);
I709_out := I709_in + I709_a1 * I709_din +
              I709_a2 * ddt(I709_din);
I710_in := I709_OUT;

I710_din := ddt(I710_in);
I710_out := I710_in + I710_a1 * I710_din +
              I710_a2 * ddt(I710_din);
I714_in1 := I712_OUT;
I714_in2 := I710_OUT;

I714_out := I714_in1 - I714_in2 ;
I711_in := I714_OUT;

I711_out := I711_gain * I711_in;
I702_in := I711_OUT;

IF (I702_in > 0.0) THEN
    I702_out1 := I702_in;
    I702_out2 := 0.0;
ELSE
    I702_out1 := 0.0;
    I702_out2 := I702_in;
END IF;
I713_in := I711_OUT;

SIG_OUT.i %= I713_in;
I688_in1 := I703_OUT1;
I688_in2 := I702_OUT1;

I688_out := I688_in1 + I688_in2 ;
I691_in1 := I703_OUT2;
```

```
    I691_in2 := I702_OUT2;

    I691_out := I691_in1 + I691_in2 ;
    I700_in1 := I688_OUT;
    I700_in2 := I690_OUT;

    I700_out := I700_in1 + I700_in2 ;
    I701_in1 := I690_OUT;
    I701_in2 := I691_OUT;

    I701_out := I701_in1 - I701_in2 ;
    I698_in := I701_OUT;

    VDD.i %= I698_in;
    I699_in := I700_OUT;

    I699_out := I699_gain * I699_in;
    I692_in := I699_OUT;

    VSS.i %= I692_in;
PROCEDURAL FOR transient =>
    I690_out := I690_parameter;
    I705_out := SIG_IN.v;
    I712_out := SIG_OUT.v;
    I693_in := I705_OUT;

    I693_out := ddt(I693_in);
    I697_in := I705_OUT;

    I697_out := I697_gain * I697_in;
    I706_in := I705_OUT;

    I696_in := I693_OUT;

    I696_out := I696_gain * I696_in;
    I707_in := I706_OUT;

    I707_dout := ddt(I707_out);
    I704_in1 := I696_OUT;
    I704_in2 := I697_OUT;

    I704_out := I704_in1 + I704_in2 ;
    I708_in := I707_OUT;

    I708_dout := ddt(I708_out);
    I695_in := I704_OUT;

    SIG_IN.i %= I695_in;
    I703_in := I704_OUT;

    IF (I703_in > 0.0) THEN
        I703_out1 := I703_in;
        I703_out2 := 0.0;
    ELSE
        I703_out1 := 0.0;
        I703_out2 := I703_in;
    END IF;
    I709_in := I708_OUT;
```

```
    I709_din := ddt(I709_in);
    I709_out := I709_in + I709_a1 * I709_din +
                I709_a2 * ddt(I709_din);
    I710_in := I709_OUT;

    I710_din := ddt(I710_in);
    I710_out := I710_in + I710_a1 * I710_din +
                I710_a2 * ddt(I710_din);
    I714_in1 := I712_OUT;
    I714_in2 := I710_OUT;

    I714_out := I714_in1 - I714_in2 ;
    I711_in := I714_OUT;

    I711_out := I711_gain * I711_in;
    I702_in := I711_OUT;

    IF (I702_in > 0.0) THEN
        I702_out1 := I702_in;
        I702_out2 := 0.0;
    ELSE
        I702_out1 := 0.0;
        I702_out2 := I702_in;
    END IF;
    I713_in := I711_OUT;

    SIG_OUT.i %= I713_in;
    I688_in1 := I703_OUT1;
    I688_in2 := I702_OUT1;

    I688_out := I688_in1 + I688_in2 ;
    I691_in1 := I703_OUT2;
    I691_in2 := I702_OUT2;

    I691_out := I691_in1 + I691_in2 ;
    I700_in1 := I688_OUT;
    I700_in2 := I690_OUT;

    I700_out := I700_in1 + I700_in2 ;
    I701_in1 := I690_OUT;
    I701_in2 := I691_OUT;

    I701_out := I701_in1 - I701_in2 ;
    I698_in := I701_OUT;

    VDD.i %= I698_in;
    I699_in := I700_OUT;

    I699_out := I699_gain * I699_in;
    I692_in := I699_OUT;

    VSS.i %= I692_in;
EQUATION (

        I706_out,
        I707_out,
        I708_out
```

```
            ) FOR dc =>
          I706_in - (-1.0/(twopi*I706_p)) * ddt(I706_out) -
            I706_out == 0.0;
          I707_in - I707_b1 * I707_dout -
            I707_b2 * ddt(I707_dout) - I707_out == 0.0;
          I708_in - I708_b1 * I708_dout -
            I708_b2 * ddt(I708_dout) - I708_out == 0.0;
      EQUATION (

            I706_out,
            I707_out,
            I708_out
            ) FOR ac =>
          I706_in - (-1.0/(twopi*I706_p)) * ddt(I706_out) -
            I706_out == 0.0;
          I707_in - I707_b1 * I707_dout -
            I707_b2 * ddt(I707_dout) - I707_out == 0.0;
          I708_in - I708_b1 * I708_dout -
            I708_b2 * ddt(I708_dout) - I708_out == 0.0;
      EQUATION (

            I706_out,
            I707_out,
            I708_out
            ) FOR transient =>
          I706_in - (-1.0/(twopi*I706_p)) * ddt(I706_out) -
            I706_out == 0.0;
          I707_in - I707_b1 * I707_dout -
            I707_b2 * ddt(I707_dout) - I707_out == 0.0;
          I708_in - I708_b1 * I708_dout -
            I708_b2 * ddt(I708_dout) - I708_out == 0.0;
  END RELATION;
END ARCHITECTURE flat;
```

Appendix C
HDL-A Description Summary

We briefly describe here some HDL-A concepts and constructs in order to allow the user to understand the HDL-A examples given in the different chapters. A mode complete description of HDL-A can be found in the User's Manual [ANA94a] and in the Language Reference Manual [ANA94b]. Furthermore, as some digital modelling concepts are inherited from VHDL, more explanations can be found in the VHDL Language Reference Manual [IEE93].

An HDL-A model—like a VHDL model—is composed of two parts: an *entity declaration* and an *architecture body*. The entity gives an external view of the model and is associated to various architecture to describe several internal views—i.e., implementations, behavioural models, etc.—of the design. This global HDL-A model structure can be expressed as

```
ENTITY entity_name IS
   external view description
END [ENTITY] [entity_name];

ARCHITECTURE architecture_name OF entity_name IS
   internal view description
END [ARCHITECTURE] [architecture_name];
```

The entity declaration and the associated interface objects are addressed first, followed by a description of the architecture part.

C.1 Entity Declaration

The entity declaration gives the name of the design and the various connection objects which compose its interface. These objects are all optional and can be classified in four categories:

- *Pins* are bi-dimensional bi-directional analogue connection points used to connect HDL-A models to each other or to other analogue components like transistors or resiators. A pin has an associated *nature* property which indicates the nature of the through and across quantities it carries. Typical pin natures are *electrical*—the across quantity is a voltage, the through quantity is a current—, *fluid*—the across quantity is a pressure, the through quantity is a flow rate—or *mechanical1*—the across quantity is a translational velocity, the through quantity ia a force. As pins carry two quantities at a time, specific constructs are necessary to access each of them. For example `inp.i` designate the current which flowa through pin `inp` and `inp.v` designate the voltage on pin `inp`.
- *Couplings* are uni-dimensional analogue connection points. In other words they can only carry either an across quantity or a through quantity. They are used to access directly an analogue quantity of another model. The connection is expressed by explicit reference to another coupling.
- *Ports* are digital connection points. They are uni-dimensional and can be either uni-directional or bi-directional. They carry signals which can be of various type ranging from *bit* to *analog*.
- *Generics* are used to set up a model to a particular application. The user can set the value of each generic at the beginning of a simulation. These values remain then constant during the whole simulation. Each generic has a type ranging from *bit* to *analog*.

An entity declaration with all kinds of connection objects would look like

```
ENTITY entity_name IS
   GENERIC ( generic_name : generic_type ;
             ...
             generic_name : generic_type ) ;
   PORT ( port_name : port_mode port_type ;
          ...
```

```
        port_name : port_mode port_type ) ;
  COUPLING ( coupling_name : coupling_type ;
             ...
             coupling_name : coupling_type ) ;
  PIN ( pin_name : pin_nature ;
        ...
        pin_name : pin_nature ) ;
END [ENTITY] [entity_name];
```

All connection objects of the same subtype can be put in lists as

```
  GENERIC ( generic_name1, generic_name2, ... : generic_type ) ;
```

Furthermore, connection objects can also be grouped in order to form vectors or matrixes as, for instance

```
  COUPLING ( coupling_name : coupling_type_vector ( a to b ) ;
  PORT ( port_name : port_type_matrix ( a to b , c downto d ) ;
```

C.2 Architecture Body

The architecture body is composed of three different parts. In the first one—the *declaration* block—all the analogue and digital variables are declared. The second one—the optional *relation* block—is used to describe analogue behaviour. The third one—the optional *process* block—is used to describe digital behaviour. HDL-A does not include any structural coding facility.

An architecture body statement can then look like

```
ARCHITECTURE architecture_name OF entity_name IS
   declaration block
BEGIN
   RELATION
      relation block
   END RELATION
   PROCESS
      process block
   END PROCESS
END [ARCHITECTURE] [architecture_name];
```

C.2.1 **Declaration Block**

Several objects can be used to describe behaviour in HDL-A. They are all optional and can be of various subtypes. They all have to be declared in the *declaration* block.

- A *state*—also called state variable—is an analogue object. It can be written in the analogue part of the model (*relation* block) only but its value can be accessed in the analogue part and in the digital part (*process* block). As a *state* has a history, it is possible to access its past values as well as its first time derivative and time integral.
- A *signal* is the same digital object as the VHDL signal. It can be written in the digital part of the model only but its value can be accessed in the analogue part and in the digital part.
- A *constant* is a neutral object which value cannot change during a simulation.
- A *variable* is a neutral object which can be written and accessed in the digital part and in the analogue part. It is commonly used to store intermediate results or quantities which do not need to have a history.
- Finally, C sub-programs can also be called from within the model. We do not explain this feature further here.

A typical declaration block could be expressed as

```
ARCHITECTURE architecture_name OF entity_name IS
   CONSTANT constant_name : constant_subtype ;
   VARIABLE variable_name : variable_subtype ;
   SIGNAL signal_name : signal_subtype ;
   STATE state_name : state_subtype ;
...
```

Again these objects can also be declared as lists, vectors or matrixes.

C.2.2 **Analogue Behaviour**

Analogue behaviour is coded in a *relation* block which begins with an *initialization* part followed by *explicit* and *implicit* blocks, each of

which is valid for one or more analysis domains—i.e., DC, AC or transient analysis.

The *initialization* is a sequence of assignment statements. Initial values can be set to any kind of variable. Default values are assigned to generics.

An *explicit* block—also called *procedural* block—is a sequence of analogue statements which can be simple assignment statements, if-then-else constructs or iterative loops. Expression can be formed using mathematical operators (+, -, *, /, `ddt`, `integ`, etc.), value access function (`current_time`, `temperature`, `previous`, etc.) and triggering functions (`rising`, `falling`). When a characteristic equation of a model can be coded as a direct assignment, it is called an explicit equation and it can be coded in the explicit block. The operator which symbolizes a direct assignment is := as in

```
y := x1 + x2 ;
```

An additional assignment operator (`%=`) has been defined to symbolize a contribution to a pin quantity.

```
inp.i %= x1 + x2 ;
```

In this example the sum of `x1` and `x2` is a contribution to the current which flows through pin `inp`. At simulation time, the simulator sums all the internal and external contributions in order to determine the actual current that flows through each pin and branch of a circuit.

An *implicit* block—also called *equation* block—is a set of characteristic equations coded with yet another equality operator (==) which indicates that the left-hand side of the equation is equal to the right-hand side, each side of the equation being possibly a very complex expression. For instance

```
a*y + b*ddt(y) == c*x1 + d*x2 ;
```

When a characteristic equation of a model cannot be coded as a direct assignment, it is called an implicit equation and it must therefore be coded in the implicit block. Explicit equations, however, can be coded either in the explicit block or the implicit block.

A *relation* block can then look like

```
RELATION
   PROCEDURAL FOR init =>
      generic_name := default_value ;
      state_name := initial_value ;
```

```
    ...
  PROCEDURAL FOR dc, ac, transient =>
    procedural analogue statements
    ...
  EQUATION (
            unknown_state, unknown_state, ...
          ) FOR dc, ac, transient => .
    equation statements
    ...
END RELATION
```

In case of analysis specific description—e.g., one description valid for DC analysis, one description valid for AC and one description valid for transient analysis—different procedural blocks and equation blocks are defined. However, all equation blocks must address the same list of unknowns.

C.2.3 Digital Behaviour

Digital behaviour is coded in a single *process* block which can be divided into an *initialization* part and a *simulation* part. The initialization part is executed only once at the beginning of the simulation. The simulation part is placed in an infinite loop and its execution is triggered off by a single *wait* statement. HDL-A uses the VHDL delta-delay to control the execution of simulation: when a process block is triggered off, the new signal values are computed according to signal values at the previous delta delay. Each signal is updated to its new value after the loop has been completely executed. The process block can then be executed again until all signals are stable. The operator <= is used for signal assignment while instantaneous variable assignment (:=) is also available. A *process* block can finally look like

```
PROCESS
BEGIN
   variable_name := initial_value ;
   signal_name <= initial_value ;
   LOOP
      WAIT ... ;
         digital statements
```

```
    END LOOP;
END PROCESS;
```

C.3 References

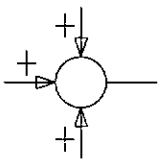
[ANA94a] *HDL-A User's Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

[ANA94b] *HDL-A Language Reference Manual,* Issue 1.0, ANACAD Electrical Engineering Software, 1994.

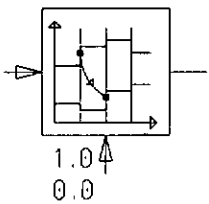
[IEE93] *IEEE Standard VHDL Language Reference Manual,* IEEE Std 1076-1993, The Institute of Electrical and Electronics Engineers Inc., 1993.

Appendix D
Standard GBS Library

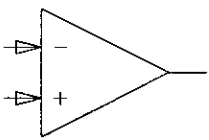
We give here a list of standard graphical building symbols which are included in ABSynth V1.1. Interface GBSs are marked with an I, hierarchical GBSs are marked with an H.



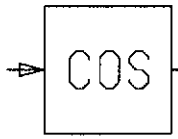
adder
3-input addition
Pins: IN1 (in), IN2 (in), IN3 (in), OUT (out)



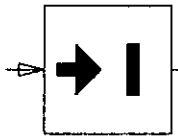
clk_delay
In sampled-data modelling, shift by one clock period
Parameters: threshold, clock_delays
Pins: IN (in), CLK (in), OUT (out)



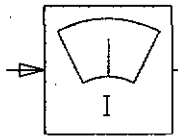
comparator
If IN1>IN2 then OUT:=1 else OUT:=0
Pins: IN1 (in), IN2 (in), OUT (out)



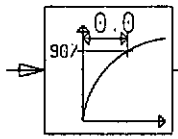
cosine
OUT:=cos(IN)
Pins: IN (in), OUT (out)



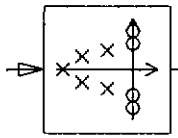
current_gen (I)
Writes the value of IN as a contribution to the current on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: IN (in), PIN_NAME (out)



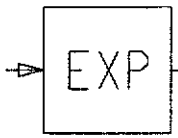
current_probe (I)
Assigns to OUT the value of the current read on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: PIN_NAME (in), OUT (out)



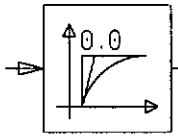
delay90
Delay until the exponential output is at 90% of the final value
Paramster: delay
Pins: IN (in), OUT (out)



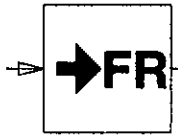
ell5_transf (H)
Fifth-order low-pass elliptic transfer function
Parameters: omega1, omega2, omega3, omega4, sigma0, sigma1, sigma3
Pins: IN (in), OUT (out)



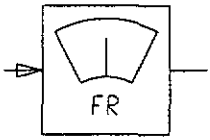
exp
OUT:=exp(IN)
Pins: IN (in), OUT (out)



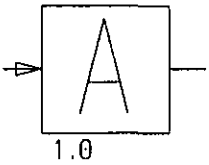
exp_delay
Exponential delay
Parameter: delay
Pins: IN (in), OUT (out)



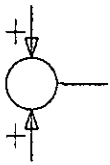
flow_rate_gen (I)
Writes the value of IN as a contribution to the flow rate on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: IN (in), PIN_NAME (out)



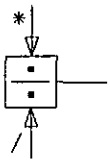
flow_rate_probe (I)
Assigns to OUT the value of the flow rate read on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: PIN_NAME (in), OUT (out)



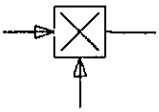
gain
Parameter: gain
Pins: IN (in), OUT (out)



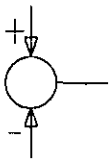
in2_adder
Pins: IN1 (in), IN2 (in), OUT (out)



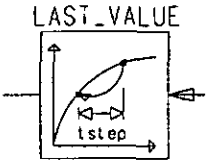
in2_divider
Pins: IN1 (in), IN2 (in), OUT (out)



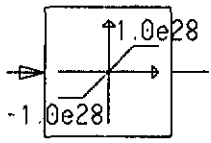
in2_multiplier
Pins: IN1 (in), IN2 (in), OUT (out)



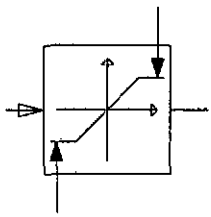
in2_sub
Pins: IN1 (in), IN2 (in), OUT (out)



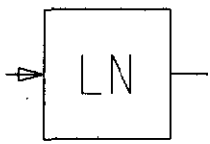
last_value
Outputs the value of a state variable one simulation time step earlier
Pins: IN (in), OUT (out)



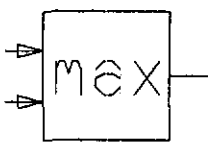
limiter
Linear with unity slope between two saturation limits controlled by static parameters
Parameters: sup, inf
Pins: IN (in), OUT (out)



limiter2
Linear with unity slope between two saturation limits controlled by external state variables
Pins: IN (in), SUP (in), INF (in), OUT (out)

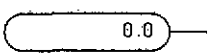


logn
OUT:=ln(IN)
Pins: IN (in), OUT (out)

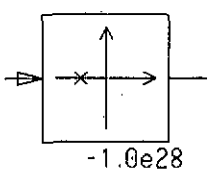


max
Pins: IN1 (in), IN2 (in), OUT (out)

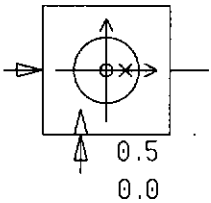
min
Pins: IN1 (in), IN2 (in), OUT (out)



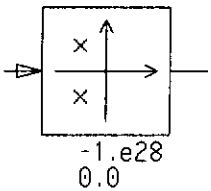
parameter
Parameter: parameter
Pin: OUT (out)



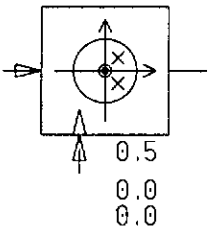
pole1
Single pole on the real axis
Parameter: p
Pins: IN (in), OUT (out)



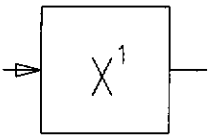
pole1z
In sampled-data modelling, single pole on the real axis combined with a zero on the origin
Parameters: p, threshold
Pins: CLK (in), IN (in), OUT (out)



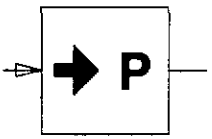
pole2
Pair of complex conjugate poles
Parameters: rp, ip
Pins: IN (in), OUT (out)



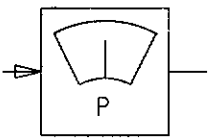
pole2z
In sampled-data modelling, pair of complex conjugate poles combined with a doubls zero on the origin
Parameters: rp, ip, threshold
Pins: CLK (in), IN (in), OUT (out)



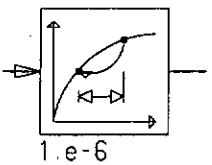
power
Parameter: exponent
Pins: IN (in), OUT (out)



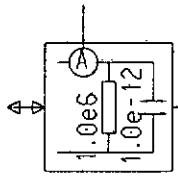
pressure_gen (I)
Writes the value of IN as a contribution to the pressure on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: IN (in), PIN_NAME (out)



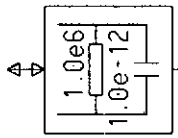
pressure_probe (I)
Assigns to OUT the value of the pressure read on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: PIN_NAME (in), OUT (out)



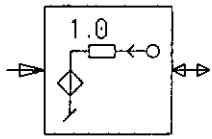
previous
Delays the input state variable of a given time interval
Parameter: delay
Pins: IN (in), OUT (out)



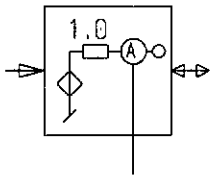
rcin_iin (H) (I)
RC-parallel input stage; the value of the input current contribution is available
Parameters: r, c
Pins: PIN_NAME (io), IIN (out), OUT (out)



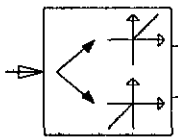
rin_cin (H) (I)
RC-parallel input stage
Parameters: r, c
Pins: PIN_NAME (io), OUT (out)



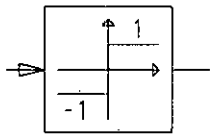
rout (H) (I)
Resistive output stage
Parameter: r
Pins: IN (in), PIN_NAME (io)



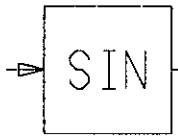
rout_iout (H) (I)
Resistive output stage; the value of the output current contribution is availsble
Parameter: r
Pins: IN (in), PIN_NAME (io), IOUT (out)



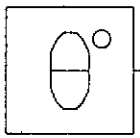
separator
If IN>0 then OUT1:=IN, OUT2:=0 else OUT1:=0, OUT2:=IN
Pins: IN (in), OUT1 (out), OUT2 (out)



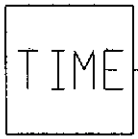
sign
Pins: IN (in), OUT (out)



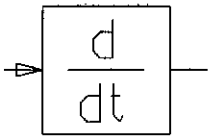
sine
OUT:=sin(IN)
Pins: IN (in), OUT (out)



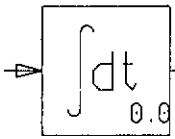
temperature
Current temperature of the simulation
Pin: OUT (out)



time
Current simulation time
Pin: OUT (out)

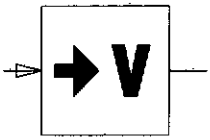


time_derivator
Pins: IN (in), OUT (out)

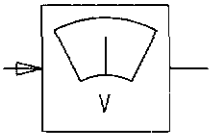


time_integrator
parameter: y0
Pins: IN (in), OUT (out)

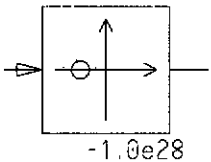
tstep
Outputs the value of the current simulation time step
Pin: OUT (out)



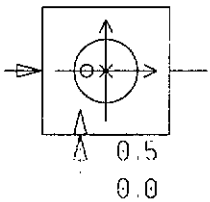
voltage_gen (I)
Writes the value of IN as a contribution to the voltage on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: IN (in), PIN_NAME (out)



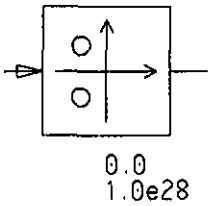
voltage_probe (I)
Assigns to OUT the value of the voltage read on PIN_NAME; PIN_NAME must be connected to a physical pin
Pins: PIN_NAME (in), OUT (out)



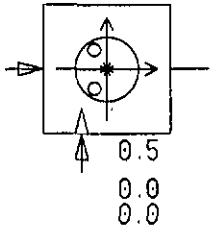
zero1
Single zero on the real axis
Parameter: z
Pins: IN (in), OUT (out)



zero1z
In sampled-data modelling, single zero on the real axis combined with a pole on the origin
Parameters: z, threshold
Pins: CLK (in), IN (in), OUT (out)



zero2
Pair of complex conjugate zeros
Parameters: rz, iz
Pins: IN (in), OUT (out)



zero2z
In sampled-data modelling, pair of complex conjugate zeros combined witb a double pole on the origin
Parameters: rz, iz, threshold
Pins: CLK (in), IN (in), OUT (out)