4

Modelling in Hardware Description Languages

4.1 Introduction

For hardware description languages (HDL) — as for every other method of describing a system — the following two questions are raised:

- What can be modelled using this description method?
- What can be achieved using this description?

This is illustrated on the basis of Figure 4.1. On the left-hand side we see the domains that are significant in our context, which are to be modelled in hardware description languages. Digital and analogue electronics should be unproblematic because hardware description languages were originally developed for precisely this purpose. Question marks stand next to the domains of multibody mechanics, continuum mechanics and software; the modelling of these domains using hardware description languages is investigated in this book. Furthermore, some approaches should be mentioned at this point that attempt to automatically translate further description forms into hardware description languages. The work of Maillot and Wendling [246], in which state diagrams are depicted in VHDL, is worth mentioning here. Sax *et al*. [359] transfer MATRIX_X descriptions from classical control technology into VHDL-AMS. Overall, hardware description languages, and in particular VHDL-AMS, appear to be capable of serving as a general exchange format for models.

The question remains of what we can undertake using a system model in a hardware description language. This is shown on the right-hand side of Figure 4.1. Initially it is possible to specify and design using hardware description languages with the resulting models being available for documentation purposes in both cases. Furthermore, such a description can be directly simulated without any intermediate

Mechatronic Systems Georg Pelz
 ISBN: 0-470-84979-7

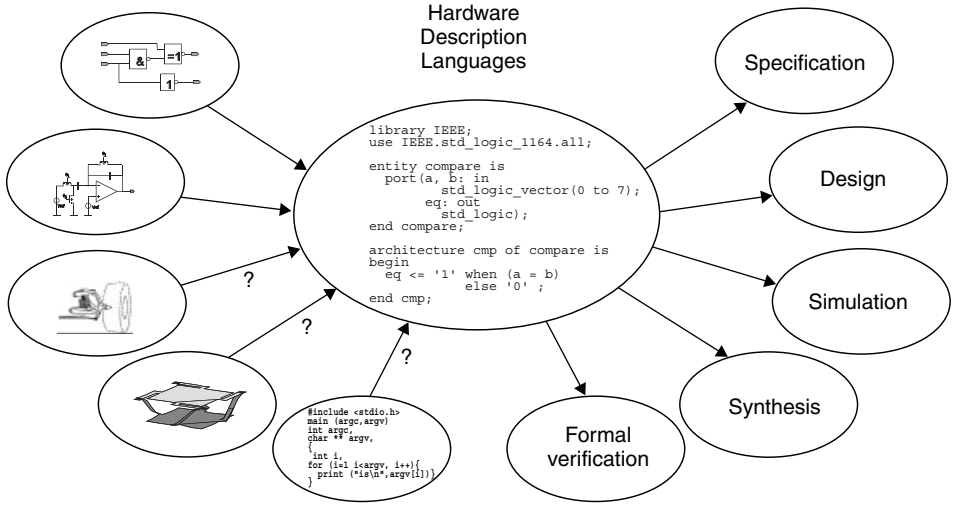

Figure 4.1 Fields of application of hardware description languages

stages, which facilitates the validation of the specifications and the verification of the designs. In the medium term formal verification or automatic synthesis of designs may also be possible, both of which currently tend to be the exclusive preserve of digital electronics.

Hardware description languages offer a whole range of advantages in relation to other approaches. For example, the problem of simulating mixed systems is moved from the simulator or programming level to the modelling level. It is thus no longer a question of implementing a tool that can execute an appropriate simulation. Instead, models have to be developed that describe the components of the system. The great advantage of this is that tried and tested simulators are available. This means that the corresponding functionalities, such as the building up and solving of equation systems, the co-simulation of digital and analogue system components or the representation of the results do not need to be re-implemented.

The second great advantage of hardware description languages lies in the fact that both the behaviour and the structure of a system or component can be formulated. Furthermore, this can occur on extremely different levels of abstraction. This allows hardware description languages to be implemented very flexibly. In particular, entire design sequences can be executed almost entirely using hardware description languages. This means that each design step primarily represents the transformation of one hardware description into another hardware description. This avoids undesirable losses due to the need to support various data formats. Furthermore, it is possible to simulate on all levels at any time and thus immediately investigate the correctness of a design step.

The most important fields of application of hardware description languages will be outlined in the sections that follow. These fields are specification, documentation, design, simulation, formal verification and synthesis. Furthermore, the syntax

and semantics of hardware description languages will be represented based upon the example of the IEEE standard 1076.1 (VHDL-AMS) passed in March 1999. This lays the foundation for the subsequent chapter on modelling.

4.2 Fields of Application

4.2.1 Formulation of specification and design

A formalised circuit description on a behavioural level, such as that provided by a hardware description language, represents the precise specification and documentation of a circuit. In many cases informal paper specifications are associated with problems, for example, if certain operating states are not predicted and are thus not specified. These difficulties are avoided by using a formal, programme-like specification. With such a specification it is generally immediately clear if a system is incompletely or even contradictorily specified. Furthermore, the hardware description language is available for reference in all cases of dispute. In such a case a simulation should be capable of clearing up all doubt. Furthermore, this route automatically provides an entry into a universal design sequence. On the basis of abstract descriptions, increasingly detailed representations are developed or generated, descriptions which can be verified against one another. In this manner both the actual design problem and the problem of consistency between the textual specifications of a performance specification and the developed system can be addressed.

4.2.2 Validation of specifications and verification of designs

The use of simulations for the validation of specifications and for the verification of designs of mechatronic and micromechatronic systems is the main theme of this work. A simulator exists for virtually all hardware description languages and, for some, several simulators are even available. The simulation of digital hardware description languages has developed from logic simulation, whilst the simulation of analogue hardware description languages has developed from circuit simulation. Hardware description languages that include both digital and analogue components are represented on an appropriate ‘mixed mode’ simulator, which spares the user from having to think about the coupling between digital and analogue simulator cores. Nevertheless, this interface is indispensable because the simulation procedures for digital and analogue fields are very different, see Sections 2.7.2 and 2.7.3.

As an alternative to simulation we can also use the methods of formal verification in the digital field. In general, the motivation for this is that the simulation of systems almost always remains incomplete because it is not possible to play through

all combinations of input values in a simulation, for reasons of running time. Formal verification makes it possible to mathematically prove the equivalence between two descriptions or the existence of certain circuit properties.

Both simulation and formal verification are normally tied to a system description in a given hardware description language. Conversely, the formulation of a system in a hardware description language often facilitates the use of appropriate tools.

4.2.3 Automatic synthesis

As indicated above, the design of a circuit often consists of an incremental refinement of a hardware description language. The theory and corresponding software tools are well developed in this field, particularly for the digital hardware descriptions. The transition from the register-transfer level to a gate net list in particular is automated as standard even now. In general, further synthesis tools are connected with this, which convert the gate net list into a standard cell layout, a gate array layout, or a programme description for a FPGA. Thus the manual part of the design sequence for digital circuits is often completed as early as the register transfer level.

4.3 Characterisation of Hardware Description Languages

At first glance a hardware description language is similar to a programming language such as C or Pascal. Models are formulated as text in a hardware description language, with a range of key words being attributed special importance. Furthermore, a predefined syntax must be adhered to. After parsing, syntactically correct models are translated into an intermediate format upon which the simulation can then be run. However, there are also important differences between hardware description languages and programming languages. For example, a programme normally runs sequentially, i.e. only one instruction is ever processed before a certain point in time. This is not acceptable for the description of hardware. All gates of a logic circuit in principle work in parallel. In hardware description languages this state is accounted for by the fact that instructions are normally processed in parallel. Certain areas of a hardware description that are reserved for sequential instructions represent the exception to this rule. In this area the typical instruments of a procedural programming language are available, such as ‘if-then-else’ constructs, loops or ‘case’ instructions.

As mentioned above, a hardware description language provides the option of describing both the behaviour and the structure of a circuit. The main difference between behaviour and structure will be explained briefly in what follows. The

addition of four numbers can be unambiguously described in terms of their function as follows:

$$y = a + b + c + d; \tag{4.1}$$

The order in which the expression is evaluated is unimportant here since the commutative law for addition applies. However, if the addition is considered on the structural level then the sequence can no longer be neglected. For example, the following two alternatives exist:

$$y = (a + b) + (c + d); \tag{4.2}$$

$$y = ((a + b) + c) + d; \tag{4.3}$$

Corresponding realisations by adders are shown in Figure 4.2. It turns out that the realisation of the expression shown on the left is completed more quickly than that on the right since only two adding stages have to be run in this case.

Formulation on a behavioural level can thus significantly reduce the complexity of a circuit description. Higher operations such as addition, subtraction, multiplication, represent a few hundreds or even a few thousands of gates. Thus the readability of such a description is significantly greater than that of other circuit descriptions. Furthermore, the reuse of descriptions that were originally developed in a different context is made easier.

Finally, hardware description languages generally open up the option of considering the individual parts of a circuit in different abstractions, see Figure 4.3. Thus circuits or systems can be fully simulated if each of their modules possesses an abstract behavioural description. This initially offers an efficiency gain compared to a complete simulation of the finished design. Furthermore, as time goes on the individual blocks can be refined during the design process, until the design has achieved the required level of abstraction for the individual parts. In particular, refinements by several circuit developers can be implemented independently

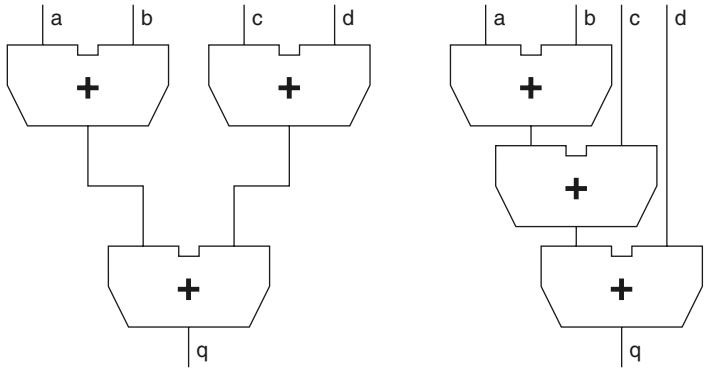

Figure 4.2 Two versions of an adder for four numbers

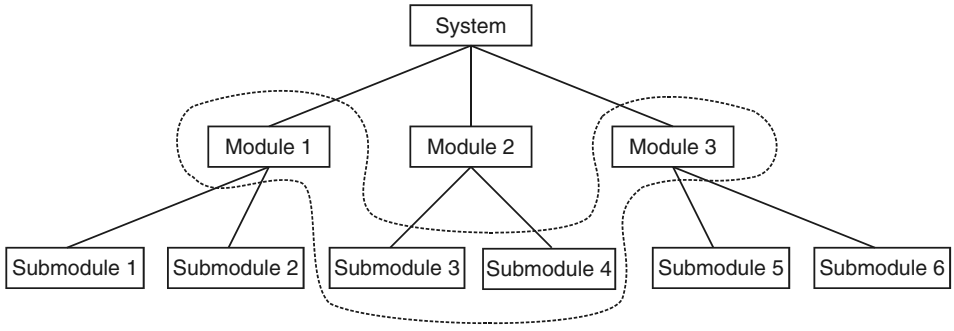

Figure 4.3 Simulation on a mixed abstraction level

of one another. Due to this 'interlacing' of the engineering work in the sense of simultaneous engineering, the design time for more complex systems can be kept within reasonable limits. Methods for the partitioning of engineering work will become increasingly important in the future because the organisational management of more complex, strongly coupled systems will increasingly be the factor that limits feasibility.

4.4 Languages

Many hardware description languages have been defined in recent years. Some of the more widespread languages were introduced by providers of design automation software. 'M-HDL' by Mentor Graphics or 'Verilog-HDL' by Cadence Design Systems are typical representatives of this group. In the analogue field the languages 'MAST' from Avant!, 'HDL-A' from Mentor Graphics, 'SpectreHDL' or 'Verilog-A' from Cadence Design Systems and 'ABCD' from Dolphin Integration S.A., are particularly worth mentioning. All these languages should be classified as proprietary hardware description languages since the associated tools could initially only be obtained from the companies in question.

A further group of hardware description languages originated from the university sector, such as 'BDS' from the University of California, Berkeley, or 'daCapo' from the University of Dortmund. However, these languages have only become widespread in the academic field. Nevertheless, because of their innovative ideas they often form the basis for commercial description languages.

A third group of languages is represented by VHDL,[2] which was initially the product of an American research programme and later became the IEEE standard 1076 as part of an expensive standardisation. The American Department of Defense, by far the biggest user in the North American area, helped the standard to make a breakthrough by making adherence to this standard a prerequisite for the

[2] VHSIC Hardware Description Language. VHSIC = very high speed integrated circuits, American promotional program for the development of particularly powerful integrated circuits.

placement of orders. Thus all CAE providers were forced to support VHDL. Other languages were also standardised such as, for example, Verilog-HDL, which was initially designed as a proprietary language. The great advantage of such standards is that they promote the exchange of circuit descriptions and furthermore make it possible for the providers of CAE tools to exchange simulators, for example, without the reformulation of the models into another language and the significant costs associated with this. Since 1987 VHDL has been a standard for the development of digital circuits and systems, which is being continuously improved and expanded. A significant aspect of this is the expansion around analogue and mixed analogue-digital constructs. In 1999 the IEEE standard 1076.1 (VHDL-AMS[3]) was passed, which covers the full language scope of VHDL and additional constructs for the modelling of analogue processes. For an introduction to VHDL the reader is referred to the books of Ashenden [15], Pellerin and Taylor [319] and Perry [334]. With regard to VHDL-AMS, as yet there is only the provisional version of the IEEE standard 1076.1 [160] and an associated tutorial [16].

As early as 1993 VHDL and Verilog-HDL enjoyed a clear predominance in the digital field compared to other languages, see Carrol [61]. Today hardly any other languages are used in the digital field. A similar concentration will presumably also take place in the field of analogue hardware description languages.

4.5 Modelling Paradigms

4.5.1 Introduction

In the following, the most important techniques of digital and analogue modelling in hardware description languages will be described. For example, the language VHDL-AMS, which covers the most important constructs of other hardware description languages, will be considered in this connection. The aim of the descriptions that follow is to convey an impression of the modelling possibilities available using hardware description languages. However, they are not a substitute for the corresponding literature. In the following, the key words in hardware description languages are written in upper case letters and all identifiers in lower case letters. In principle this makes no difference, since in VHDL and VHDL-AMS, no differentiation is made between upper and lower case.

A VHDL model is organised into various descriptions. Every module has precisely one interface description, which in principle specifies the corresponding interface signals and their type and direction. Such a description is also called an `ENTITY`. For each `ENTITY` there is one or more `ARCHITECTURE` descriptions that contains the different variations of the modelling of the module. For example, in the following section three architectures will be listed for a module. For frequently used constructs it is possible to define packages, which are themselves split into

[3] VHDL analogue and mixed signal extensions.

an interface section (PACKAGE) and an implementation section (PACKAGE BODY). A fifth group of descriptions specifies which architectures should form the basis for a simulation. These are also called configurations (CONFIGURATION).

4.5.2 Structural and behaviour-oriented modelling

Structural modelling formulates the submodules from which a module is composed. In contrast to this, behaviour-oriented modelling describes the function and timing of the module. Let us clarify this using the example of a full adder. Hardware description 4.1 shows the interface description of a fictitious full adder in VHDL. Comments for the rest of the lines are preceded by a double minus sign. Using the LIBRARY and USE instructions a PACKAGE is first referenced, which includes the necessary types for the digital signals, e.g. std_logic. The ENTITY description mainly consists of a PORT instruction, which declares the inputs and outputs of the full adder.

```
LIBRARY IEEE;
-- IEEE Package for logic types
USE IEEE.std_logic_1164.all;

ENTITY full_adder IS
 -- two sum inputs, one Carry-In
 -- one sum output, one Carry-Out ...
 PORT (i1, i2, ci:  IN std_logic;
       sum, co:  OUT std_logic);
END full_adder;
```

Hardware description 4.1 Interface description of a full adder

The first possibility is represented by structural modelling, in which the full adder is made up of a half adder and an Or gate, see Hardware description 4.2. The timing is taken from the timing of the underlying modules.

```
ARCHITECTURE structure OF full_adder IS
...
BEGIN
...
  inst1 : half_adder(i1 ,i2 ,tc1 ,ts1);   -- Instantiation HA
  inst2 : half_adder(cin ,ts1 ,tc2 ,sum); -- Instantiation HA
  inst3 : or_gate (tc1 ,tc2 ,co);         -- Instantiation OR
END structure;
```

Hardware description 4.2 Structural description of a full adder

The simplest form of behavioural modelling is the data flow description, in which the underlying Boolean function is merely assembled from basic functions and the calculation of the results performed after a delay. This is shown in Hardware description 4.3.

```
ARCHITECTURE data_flow OF full_adder IS
BEGIN -- Signal assignment according to Boolean function...
 sum <= i1 xor i2 xor ci AFTER 3 ns;
 co  <= (i1 and i2) or (i1 and ci) or (i2 and ci) AFTER 2 ns;
END data_flow;
```

Hardware description 4.3 Data flow description of a full adder

However, not all functions that are possible are predefined. It can also be tiresome to fully prepare the Boolean functions. In such cases it is also possible to provide a purely behavioural description, which relates input and output assignment to each other in tabular form, see Hardware description 4.4. This is based upon a so-called process, the body of which includes sequential instructions.

```
ARCHITECTURE behaviour OF full_adder IS
BEGIN
 PROCESS                                    -- Process head ...
   VARIABLE tmp : std_logic_vector(2 DOWNTO 0);
 BEGIN         -- Process body with sequential instructions ...
  WAIT ON i1, i2, ci;                  -- Wait for signal change
  tmp(2) := i1; tmp(1) := i2; tmp(0) := ci;    --Store in vect.
  CASE tmp IS                         -- Case differentiation ...
  WHEN "000" =>
   sum <= '0' AFTER 3 ns;               -- Signal assignment sum
   co <= '0' AFTER 2 ns;          -- Signal assignment Carry-Out
  WHEN "001" =>
   sum <= '1' AFTER 3 ns;               -- Signal assignment sum
   co <= '0' AFTER 2 ns;          -- Signal assignment Carry-Out
  WHEN ...
 END CASE;
 END PROCESS;
END behaviour;
```

Hardware description 4.4 Behaviour-oriented description of a full adder

4.5.3 Digital modelling

The process (`PROCESS`) will be explained in more detail in the following. It forms the work-horse of digital modelling. Virtually all digital relationships are modelled either directly as a process or in a form that is easy to convert into a process. The process is attributed to the parallel instructions. Thus it is processed in parallel to the other processes and the remaining parallel instructions. The body of a process contains sequential commands that are thus processed one after the other. When the processing reaches the end of the body, it jumps back to the start and thus executes an endless loop. To prevent this from causing the simulation to hang, each body must contain at least one synchronisation point in the form of an explicit or implicit WAIT instruction. Its task is to delay progress in the body of the process

by an amount that depends upon its parameter. This may be based, for example, upon a fixed time period or the occurrence of a certain event. The process is executed accordingly by the performance of a sequence of instructions between two synchronisation points. Sequential instructions in VHDL are comparable to the instructions of procedural programming languages. In the following, a few processes will be described as examples.

Example: multiplexer

The first example is a multiplexer that is formulated in Hardware description 4.5 as `ENTITY` and `ARCHITECTURE`. Synchronisation takes place by means of the `WAIT` instruction, which interrupts the execution of the process until at least one of the signals `a`, `b` or `sel` has changed. Thus the body of the architecture proceeds as soon as there is a change at the inputs of the multiplexer. Then and only then can a change at the outputs be expected.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;    -- IEEE package for logic types
ENTITY mux IS       -- Interface description of multiplexer ...
 PORT(a, b, sel: IN std_logic;
     q         : OUT std_logic);
END mux;

ARCHITECTURE behaviour OF mux IS -- Architecture description...
BEGIN
 PROCESS                                          -- Process ...
 BEGIN
  WAIT ON sel, a, b;                 -- Wait for signal changes
  if sel = '1' then                     -- Case differentiation
   q <= a;                                 -- Signal allocation
  else
   q <= b;                                 -- Signal allocation
  END IF;
 END PROCESS;
END behaviour;
```

Hardware description 4.5 Behaviour-oriented modelling of a multiplexer

Example: multiplier

The next example is used to explain in more detail the various abstractions of modelling in a design sequence. The example relates to a multiplier. In its simplest form this can be described by a times sign, see Hardware description 4.6. This form of description is extremely compact, although a realisation of the circuit can consist of thousands of gates. In a second description, multiplication can be traced back to shifting and adding, as we learned multiplication at school, see

Hardware description 4.7. This corresponds with the first step in the direction of implementation. Most synthesis tools are able to translate this description into a gate circuit, which could be followed up by representation on a FPGA.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;    -- IEEE package for logic types
USE IEEE.std_logic_arith.all;    -- IEEE package for associated
                                      arith.

ENTITY multiplier IS
 PORT(clk: in std logic;
     a, b : IN std_logic_vector(3 DOWNTO 0);
     q    : OUT std_logic_vector(7 DOWNTO 0));
END multiplier;

ARCHITECTURE behaviour1 OF multiplier IS
BEGIN PROCESS
 BEGIN
  WAIT UNTIL rising_edge(clk);          -- Wait for rising edge
  q <= a*b;                   -- Multiplier and assign result to
 END PROCESS;
END behaviour1;
```

Hardware description 4.6 Behavioural description of a multiplier on the basis of a multiplication operation

At this point we should highlight a further point. The `WAIT` instructions delay the sequence up to the next active clock-pulse edge. For the architecture `behaviour1` this means that the multiplication must be completed within one clock cycle. The realisation `behaviour2`, however, unrolls the loop over time and not spatially. Thus the calculation of the product requires at least as many clock cycles as the number of bits of the operands. In the VHDL formulation this is achieved by the fact that the loop contains a `WAIT` instruction.

```
ARCHITECTURE behaviour2 OF multiplier IS
BEGIN
 PROCESS
  VARIABLE pp, res : std_logic_vector(7 DOWNTO 0);
 BEGIN
  WAIT UNTIL rising_edge(clk);          -- Wait for rising edge
  res := "00000000";                 -- Initialise variable res
  FOR index IN 0 TO 3 LOOP              -- Loop index := 0 .. 3
    WAIT UNTIL rising_edge(clk);        -- Wait for rising edge
    pp := "00000000";                  -- Initialise variable pp
    IF b(index) = '1' THEN           -- If bit index of b set ...
      pp((index + 3) DOWNTO index) := a;         -- Adder moved
    END IF;
    res := res + pp;                        -- Accumulate result
```

```
        END LOOP;
   WAIT UNTIL rising_edge(clk);          -- Wait for rising edge
   q <= res;                      -- Signal assignment for output
 END PROCESS;
END behaviour2;
```

Hardware description 4.7 Behavioural description of a multiplier on the basis of moving and adding

Digital signal assignment

Up until now we have based our description of a signal assignment upon an intuitive understanding, which in some cases can be deceptive. This can be clarified by looking at a simple inverter gate. The function of the inverter is quickly described. However, in some cases this does not achieve the desired result. The inverter may have a delay time of 100 picoseconds. If a pulse of one picosecond occurs at its input then we would assume in the first approximation that this pulse would be observed in the opposite polarity at the output 100 picoseconds later. However, this is not physically correct because the pulse is much too short to effect a change at the output. Before this has moved to a significant degree, the cause has disappeared again. In order to bring about this 'inert' behaviour it is necessary for each signal assignment to evaluate the right-hand side correctly and to draw up a list of current and future events. If necessary, the future events may have to be deleted again before they are realised. This is also the case, for example, if the right-hand side always produces an assignment with the same value, so that a formal assignment yields no new information for the signal. In this case we can postpone the assignment, so that no events without information content are produced. This task and others are undertaken by the so-called signal driver.

4.5.4 Analogue modelling

Introduction

We can differentiate between three classical applications of analogue modelling, see Vachoux and Berge [406]. Firstly, and self-evidently, it is implemented when the system under investigation consists wholly or partially of analogue components. But even when looking at digital systems, the consideration of an analogue environment of the circuit may still be necessary. Finally, analogue effects, such as signal delays or coupling capacitances, often cannot be disregarded especially for digital high-speed circuits.

Again, the extremely different levels of abstraction can be represented. Thus, on the purely behavioural level we can provide models based upon transfer functions or differential equations. At a lower level of abstraction, so-called macromodels are often used, which may represent the standard blocks of analogue circuit design, e.g.

operational amplifiers, comparators, etc. Such macro-models describe behaviour at the terminals, for example, in the form of a characteristic. Finally, we can also model components such as transistors, diodes, etc. using analogue hardware description languages.

Furthermore, the methodology of analogue modelling is in line with the following strategies:

Structural definition Analogue hardware description languages permit the formulation of a component as an interconnection of its subcomponents.

Behavioural definition The description of the terminal behaviour of components on the basis of mathematical equations is one of the main properties of analogue hardware description languages.

Conservative modelling Analogue hardware description languages permit the formulation of models on the basis of potential (across) and flow (through) variables, e.g. voltage and current or velocity and force, meaning that Kirchhoff's laws apply. The product of potential and flow variables is normally represented by energy. So this formulation is set up to describe energy flows.

Non-conservative modelling Non-conservative quantities can also be described, allowing block or signal flow diagrams to be formulated using hardware description languages. Often the description of an information or control flow predominates.

Table model Table models are normally based upon a piece-wise linear description, which may be smoothed for numerical reasons. These models can also be unproblematically formulated into an analogue hardware description language.

Arbitrary mixed forms Analogue hardware description languages permit the use of arbitrary mixed forms of these modelling strategies.

Using the above-mentioned modelling strategies, analogue hardware description languages thus permit the formulation of structural, physical and experimental models, so that the fundamental approaches to modelling from Chapter 2 are fully represented. The use of mathematical equations in the description of the models allows the addition of various fields to the discussion. The fields listed in Table 4.1 are particularly pertinent here, see Antao [12].

Table 4.1 Model formulation in analogue hardware description languages

Description	Field	Representation
Discrete	Time	Differential equations and algebraic equations
Continuous	Time	Differential equations and algebraic equations
Discrete	Frequency	Z-transformation
Continuous	Frequency	Laplace transformation

Now, if analogous behaviour is to be formulated in a hardware description language this normally occurs in the form of mathematical equations. In VHDL-AMS these equations are also termed simultaneous instructions. Both sides of the equation must have real values. The equations are symmetrical in the sense that swapping the left and right-hand side leads to the same results. The analogue solver is responsible for the fact that these equations are approximately fulfilled. In addition to the equations there are also the simultaneous versions of the `IF`, `CASE` and `PROCEDURAL` instructions, which facilitates sequential notation. Let us now clarify this using the example of a diode model.

Hardware description 4.8 shows a simple diode model in VHDL-AMS, see [160]. The division into interface and implementation, i.e. into `ENTITY` and `ARCHITECTURE`, also applies for the analogue model. In addition to the anode and cathode electrical connections the interface now includes a `GENERIC` instruction that permits the named parameter to set when the model is instanced. Furthermore, standard values are specified that are used if no further specifications are encountered during instancing. Then some electrical quantities are initially declared in the architecture such as, for example, the diode current `id` and the voltage `ud` across the diode. The threshold voltage `ut` is finally declared as a constant. The actual equations define the diode current `id`, the charge of the diode `q` and an additional current `ic`, which is found from the derivative of charge with respect to time `q'DOT`. Furthermore, the fact is worthy of special mention that individual equations can be allocated to a predefined accuracy group by means of the `TOLERANCE` instruction, so that different accuracies can be set for various equations. However, this means that no decision is anticipated regarding which criteria the simulator is to use for the evaluation of accuracy and how this is to be calculated.

```
ENTITY diode IS
 -- Parameter declaration with default values ...
 GENERIC (is0:real := 1.0e-14; tau, rd : real := 0.0);
 -- Inputs/outputs
 PORT     (TERMINAL anode, cathode: electrical);
END ENTITY diode;

ARCHITECTURE simultaneous OF diode IS
 -- Declaration of variables and constants ...
 QUANTITY ud ACROSS id, ic THROUGH anode TO cathode;
 QUANTITY q: real;
 CONSTANT ut: voltage := 0.0258;
BEGIN -- Defining equations ...
 id == is0*(exp((ud-rd*id)/ut)-1.0);
 q  == tau*id TOLERANCE "Charge";
 ic == q'DOT;
END ARCHITECTURE simultaneous;
```

Hardware description 4.8 Simultaneous behavioural description of a diode

Alternatively, a sequential description can also be provided, see Hardware description 4.9. Here the causality is specified by the assignments. However, some possibilities for sequential modelling exist such as the use of IF-THEN-ELSE constructs, CASE instructions or loops, meaning that this form of modelling also has its attraction. However formulated, the two descriptions should, however, supply the same outputs.

```
ARCHITECTURE procedural OF diode IS
 QUANTITY ud ACROSS id, ic THROUGH anode TO cathode;
 QUANTITY q: real;
 CONSTANT ut: voltage := 0.0258;
BEGIN
 p1: PROCEDURAL BEGIN -- defining assignments
   id := is0*(exp((ud-rd*id)/ut)-1.0);
   q  := tau*id TOLERANCE "charge";
   ic := q'DOT;
 END PROCEDURAL;
END ARCHITECTURE procedural;
```

Hardware description 4.9 Sequential behavioural description of a diode

Physical domains and associated quantities

When describing analogue relationships in VHDL-AMS the physical domains that can be described are not specified in advance. Rather, it is even possible to declare domains with their associated quantities subsequently. Here a differentiation is made between potentials and flows, which are declared by the keywords ACROSS and THROUGH. For electronics these may be voltage and current. Hardware description 4.10 shows the corresponding declaration as PACKAGE.

```
PACKAGE electrical_system IS
 SUBTYPE voltage IS real TOLERANCE "low_voltage";
 SUBTYPE current IS real TOLERANCE "low_current";
 NATURE electrical IS
  voltage ACROSS;   -- Potential
  current THROUGH;  -- Flow
 ALIAS ground IS electrical'reference;
 ...
END PACKAGE electrical_system;
```

Hardware description 4.10 Declaration of electrical potentials and flows

In the same manner, potentials and flows can be declared to arbitrary other domains. For translational mechanics these might be velocity and force; for rotational mechanics, rotational velocity and torque.

In a model the quantities used can be declared as either a THROUGH or an ACROSS QUANTITY. This is a real number that describes a continuous variable. Kirchhoff's

voltage law is applied for potential quantities, which means that all ACROSS quantities in a closed loop add up to zero. For the flow quantities, Kirchhoff's current law applies. Thus all THROUGH quantities at a node add up to zero. In addition to the declared quantities others are implicitly defined such as, for example, q'DOT, q'INTEG and q'DELAYED(t). These denote the derivative of the quantity q with respect to time, the integral of the quantity q with respect to time and a quantity q delayed by time t. In addition to the potentials and flows it is sometimes worthwhile considering quantities that are not subject to Kirchhoff's laws. For example, in control technology signal flow diagrams or block diagrams are often considered, in which the individual quantities do not occur in pairs and furthermore have a direction. Kirchhoff's laws in particular do not apply to these quantities. In VHDL-AMS such quantities can also be used, as is demonstrated by the following example of a combined adder/integrator, see Hardware description 4.11 and [16].

```
ENTITY adder_integrator IS
 GENERIC (k1,k2: real);
 PORT    (QUANTITY in1, in2: IN real;
          QUANTITY outp: OUT real);
END ENTITY adder_integrator;

ARCHITECTURE signal_flow OF adder_integrator IS
 QUANTITY qint: real;
BEGIN -- defining equations ...
 qint == k1*in1 + k2*in2;
 outp == qint'INTEG; -- Integration
END ARCHITECTURE signal_flow;
```

Hardware description 4.11 Signal flow modelling of a combined adder/integrator

Discontinuities

In the case of mechanical models in particular, non-continuous relationships also often have to be modelled. These are illustrated in what follows based upon the example of a bouncing ball, see Hardware description 4.12 and Bakalar *et al*. [16]. Two discontinuities are considered in this model. The first of these is the start of the simulation at which the initial state is set at the first BREAK command. The second discontinuity consists of the fact that the bouncing ball reverses its velocity when the it hits a surface, i.e. at $s \leq 0$. This corresponds with an elastic impact. Furthermore, the IF instruction ensures that the braking effect of air resistance acts with gravity when rising and against gravity when falling.

```
LIBRARY disciplines;              -- Reference to a package with
USE disciplines.mechanical.all;   -- the mechanical declarations
ENTITY ball IS                    -- Autonomous model,
END ENTITY ball;                  -- no connections
ARCHITECTURE simple OF ball IS
```

```
 QUANTITY v : velocity;          -- Velocity
 QUANTITY s : displacement;      -- Relative position
 CONSTANT g : real := 9.81;      -- Gravity
 CONSTANT lw: real := 0.1;       -- Air resistance
BEGIN
 -- Initial conditions ...
 BREAK v => 0.0, s => 10.0;
 -- Detect discontinuity and invert velocity...
 BREAK v => -v WHEN NOT s'ABOVE(0.0);
 s'DOT == v;                  -- v = ds/dt
 IF v > 0.0 USE
      v'DOT == -g - v**2*lw; -- Accel. = -Gravity - Air resist.
 ELSE
      v'DOT == -g + v**2*lw; -- Accel. = -Gravity + Air resist.
 END USE;
END ARCHITECTURE simple;
```

Hardware description 4.12 Modelling of discontinuities using the example of a bouncing ball

Modelling in the frequency range

In addition to modelling in the time range we can also provide a description in the frequency range. This is based upon a small-signal model, which arises as a result of the linearisation of the equations around the working point. In this model it is possible to define quantities based upon their spectra. Furthermore, predefined functions are available that effect either a Laplace or a Z-transformation. In this manner filters, for example, can be described in a very simple way.

4.6 Simulation of Models in Hardware Description Languages

In what follows the focus will again lie on the consideration of VHDL-AMS, which provides a good example of a hardware description language with digital and analogue components. Thus, we are automatically considering a mixed digital-analogue simulation. The first step is the performance of the so-called elaboration, which includes the evaluation of structural sections of the model and thus builds up a complete system model from the module instantiations. The digital section consists of a number of processes and the digital simulator core. The analogue section consists of a number of equations and the analogue solver. A necessary prerequisite for analogue solvability is that the number of equations and the number of (analogue) unknowns in the model are equal. For VHDL-AMS this is the number of `THROUGH` quantities, free quantities and interface quantities with the direction `OUT`. The actual simulation then runs in two phases. In the first phase

the operating point of the system is determined. There then follows a simulation in the time, small-signal or noise range. If a model contains no quantities, then the simulation is reduced to a pure logic simulation, which corresponds with the predetermined simulation cycle in the VHDL 1076 standard, see [158] and [159]. If, on the other hand, a model does not include a digital signal, then the simulation is exclusively analogue.

The simulation cycle of VHDL-AMS should be described based upon Algorithm 4.1 below, which is formulated in pseudocode. The representation is somewhat simplified, for a complete version refer to [160].

```
Loop {
  Call to the analogue solver;
  Set current time Tc to Tn;
  If maximum time reached or no active
      processes present then simulation end;
  Bring digital signal to newest state;
  Execute active, not delayed processes
      up to the next synchronisation point (= WAIT);
  Calculate next time point of digital activity Tn;
  If Tn = Tc              -- delta time interval
      then proceed to the start of the loop;
  Execute active, delayed processes
      up to the next synchronisation point;
  Calculate next time point of digital activity Tn;
}
```

Algorithm 4.1 Simplified simulation cycle of VHDL-AMS

The simulation cycle of VHDL-AMS includes the combined simulation of analogue and digital processes and thus requires a corresponding linking of the digital and analogue solution strategies. Initially the analogue solver is called up, which in general calculates a solution up to time point T_n. However, it may be necessary for T_n to be set back to T_n' ($T_n' < T_n$), if the analogue world has produced a digital event at time point T_n'. The current time T_c is then set to T_n or possibly to T_n'. If the maximum representable time has now been reached by the time variables, or there are no longer any active processes, the simulation is ended. Otherwise the digital signals are set to the latest state and the active processes before the next synchronisation point executed. However, the execution of some of these processes is delayed. Then the next time of digital activity T_n is calculated. If T_n is equal to T_c then it is a time increment that elapses in zero time, i.e. a delta time increment. In this case execution is restarted at the start of the loop. Otherwise the delayed processes are executed and a new T_n calculated. This completes the circle and execution is recommenced at the start of the loop.

4.7 Summary

This chapter has described the opportunities of modelling in hardware description languages. It thus provides the basis for the investigation of the inclusion of software and mechanics using hardware description languages covered in the next chapter.